Monitors and Blame Assignment for Higher-Order Session Types

Limin Jia, Hannah Gommerstadt, and Frank Pfenning

November 6, 2015

CMU-CyLab-15-004

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213
Monitors and Blame Assignment for Higher-Order Session Types

Limin Jia  Hannah Gommerstadt  Frank Pfenning
Carnegie Mellon University
liminjia@cmu.edu  hgommers@cs.cmu.edu  fp@cs.cmu.edu

Abstract

Session types provide a means to prescribe the communication behavior between concurrent message-passing processes. However, in a distributed setting, some processes may be written in languages that do not support static typing of sessions or may be compromised by a malicious intruder, violating invariants of the session types. In such a setting, dynamically monitoring communication between processes becomes a necessity for identifying undesirable actions. In this paper, we show how to dynamically monitor communication to enforce adherence to session types in a higher-order setting. We present a system of blame assignment in the case when the monitor detects an undesirable action and an alarm is raised. We prove that dynamic monitoring does not change system behavior for well-typed processes, and that one of an indicated set of possible culprits must have been compromised in case of an alarm.

1. Introduction

Session types [15, 16] provide a means to prescribe the communication behavior between concurrent message-passing processes. A variant of previously defined session type systems has been discovered to be in a Curry-Howard correspondence with linear logic [3, 5, 29], where session types correspond to linear propositions, process expressions to sequent proofs, and communication to cut reduction. This observation has formed the basis for SILL [28], a programming language integrating ordinary functional with message-passing concurrent computation. Separately, it has been shown that the logical foundation supports not only the synchronous communication model of Caires and Pfenning [3] but also asynchronous communication with message queues [8]. Recently, these two styles of communication have been unified using the logical concept of polarization [21]. In this formulation, communication is a priori asynchronous, but synchronization can be specified via so-called shift operators in the type.

All of the above mentioned systems and languages satisfy session fidelity (a generalization of type preservation) and global progress (a generalization of ordinary progress). These properties are inherited directly from cut elimination in linear logic, since the operational semantics is based on cut reduction. They continue to hold in the presence of recursive session types when other properties such as productivity and termination fail. Despite exhibiting significant confluence in the operational semantics, the languages also have strong confluence properties [19], again inherited from the confluence of cut reduction in linear logic. They therefore occupy a kind of middle ground between functional languages and foundational calculi such as the π-calculus with much inherent nondeterminism.

At this point one might consider the basic foundational questions regarding logic-based session-typed languages to be solved. Usually, static typing together with a type preservation theorem is seen as license to erase types at runtime. In the setting of message-passing concurrency, however, there are two important reasons to also consider dynamic monitoring of communication. The first is that when spawning a new process, part of the execution of a program now escapes immediate control of the original process. If the new process is compromised by a malicious intruder, then incorrect yet unchecked messages can wreak havoc on the original process, inducing undefined behavior such as the infamous buffer overruns. A second reason is that session types are explicitly designed to abstract away from local computation. This means we can use them to safely connect communicating processes written in a variety of different and even incompatible languages, as long as they (dynamically) adhere to the session protocol and basic data formats. But there is often no reason to trust remote processes, or believe in the correctness of the code even for trusted processes, so dynamically monitoring communication becomes a necessity.

However, designing monitoring infrastructure that allows precise blame assignment in the presence of higher-order processes is challenging. In such settings, channels of arbitrary type can be passed along other channels. When this occurs, the runtime monitor cannot immediately determine if the process communicating over the channel being passed along satisfies session fidelity, but must monitor further communication over this channel.

This paper makes the following main contributions:

- We define a powerful but intuitive adversary model for session-typed communicating processes.
- We show how to dynamically monitor communication to enforce adherence to session types in a higher-order setting.
- We present a precise system of blame assignment in the case an alarm is raised.
- We prove that dynamic monitoring does not change system behavior for well-typed processes, and that one of an indicated set of possible culprits must have been compromised in case of an alarm.

For simplicity, the formal development omits affine and shared channels present in Pfenning and Griffith [21], but our techniques are general enough to also encompass them. Of course, intruders could fly under the rader of the dynamic monitor and send incorrect values of the correct types. To capture such violations, additional measures such as contracts or dependent types [4, 14, 22, 23, 26] need to be considered. We conjecture that the framework presented in this paper is sufficient to permit generalizations to these more precise properties.

2. Polarized Session Typed Processes

The first fundamental idea behind our system of session types is that every linear channel has two endpoints: a provider and a client.
A process always provides along a single channel, but it may be
the client of multiple channels. This identification is quite strong:
each channel is provided by a unique process, and each process
provides along a unique channel which never changes. Our basic
typing judgment is therefore

\[ x_1 : A_1, \ldots, x_n : A_n \vdash P : (x : A) \]

where \( P \) is a process providing \( x : A \) and using \( x_i : A_i \). The
session types \( A \) and \( A_i \) prescribe the communication behavior
along channels \( x \) and \( x_i \), respectively.

We first explain the session types \( A \) and their intuitive interpretation
in terms of process behavior, followed by the process expression
and the typing rules.

### 2.1 Polarized Session Types

The distinction between provider and client is orthogonal to the
direction of communication: a process can both send and receive
along the channel it provides and any channel that it uses. Of
course, the session type of the channel will tell us which.

We explicitly polarize the types: any change in the direction of
communication must be explicitly denoted by a shift operator [18].
The polarized formulation of session types supports synchronous
and asynchronous communication in the same language, while at
the same time providing a uniform integration of linear, affine, and
shared channels [21]. It also simplifies monitoring since message
queues always have a definitive direction. From the perspective of
monitoring and blame assignment, under our assumptions, there
is no significant difference between linear and affine channels, so
for simplicity we treat only linear ones. Shared channels can be
handled straightforwardly with our techniques, so we omit them
from the development in order to streamline the presentation.

From the perspective of the provider, positive types correspond
to sending a message and negative types correspond to receiving
a message. The kinds of messages we consider here are basic data
values \( v \) (like integers or booleans), labels \( \text{lab} \) that indicate an
internal or external choice between alternative ways to continue
a session, channels \( a \), and special tokens ‘end’, to indicate the
end of a session and ‘shift’, to indicate a change in direction of
communications.

As an example for the use of session types, we consider the
specification of the behavior of a priority queue. A priority queue
offers the client a choice between inserting an element into the
queue, or deleting the element of the currently highest priority. The
elements themselves are channels of some arbitrary positive type
\( A \). This is represented as an external choice:

\[
\text{pq}^- A^+ = \\&\{ \text{ins} : \ldots, \\
\quad \text{del} : \ldots \} 
\]

The type \( \text{pq}^- A^+ \) is negative, since the process providing a priority
queue has to start by receiving, either the label \( \text{ins} \) or the label \( \text{del} \).

When the client elects to insert, it first has to send the priority as
an integer, and then the channel. After that, the provider has to once
again behave as a priority queue which means the type is recursive.

\[
\text{pq}^- A^+ = \\&\{ \text{ins} : \text{int} \to A^+ \to \text{pq}^- A^+, \\
\quad \text{del} : \ldots \} 
\]

When the client elects to delete the element of highest priority,
the provider sends back either the label \( \text{none} \) (indicating that the
priority queue is empty) and terminates, or the label \( \text{some} \), followed
by the priority and then element itself. This is an example of an
internal choice, since the provider has to send the label in this case.

\[
\text{pq}^- A^+ = \\&\{ \text{ins} : \text{int} \to A^+ \to \text{pq}^- A^+, \\
\quad \text{del} : \uparrow\{ \text{none} : 1, \text{some} : \text{int} \land A^+ \to \bot \text{pq}^- A^+ \} 
\]

Note that we needed to insert some shifts to indicate the change of
direction in communication: after receiving label \( \text{del} \), the provider
will then have to send a shift followed by another label (either \( \text{none} \)
or \( \text{some} \)). Before we recurse, we have to shift once again to be ready
to receive the next insert or delete.

The type \( \text{pq}^- A^+ \) is an example of a recursive session type. Re-
cursive types are equal to their unfolding, i.e., equirecursive, so
there are no special type constructors for them. However, we allow
matching recursive process definitions. Similarly, we allow types
and process definitions to be (implicitly) polymorphic. Type vari-
ables, however, must have a well-defined polarity and can only be
instantiated with types of that polarity. For example, the definition
of \( \text{pq}^- A^+ \) is parameterized by a positive type \( A^+ \). If we want to
instantiate it with a negative type \( B^- \), we have to use \( \bot B^- \).

### 2.2 Process Expressions

The syntactic constructs for writing processes are relatively straight-
forward. We indicate dependence on a bound variable \( x \) (for channels)
or \( n \) (a value) using a subscript that we may substitute for.

Returning to our example, we now give some (recursive) proc-
tess definitions. We write \( c \leftarrow p \ n_1 \cdots n_k \leftarrow d_1, \ldots, d_m \) for
a process that provides a session along channel \( c \) and uses channels
\( d_1, \ldots, d_m \). It is parameterized by data values \( n_1, \ldots, n_k \) which
constitute a form of local state. We write the type of such as func-
tion as \( \tau_1 \to \cdots \to \tau_k \to \{ A \leftarrow A_1, \ldots, A_m \} \) as in SILL [28].

\[
\text{empty} : \{ \text{pq}^- A^+ \} \\
\quad c \leftarrow \text{empty} = \begin{cases} \text{ins} \to p \leftarrow \text{recv} c ; & \% \text{receive priority } p \\
\quad \text{case } c \text{ of } \quad \text{send end and terminate} \\
\quad \quad \text{close } c & \quad \text{send end and terminate} \\
\quad \quad \text{send } c (x \leftarrow P_c) ; Q & \quad \text{send new } a \text{ along } c, \\
\quad \quad \text{send } c (\text{lab}_i) ; Q & \quad \text{spawn } P_a \text{ and cont. as } Q, \\
\quad \quad \text{case } c \text{ of } \{ \text{lab}_i \Rightarrow Q_i \} & \quad \text{receive } a \text{ along } c, \text{ cont. as } Q_a, \\
\quad \quad \text{send } c (\text{v}) ; Q & \quad \text{send lab}_i \text{ along } c, \text{ cont. as } Q, \\
\quad \quad \text{send } c (\text{shift}) ; Q & \quad \text{send shift along } c, \text{ cont. as } Q, \\
\quad \quad \text{shift} \leftarrow \text{recv } c ; Q & \quad \text{receive shift along } c, \text{ cont. as } Q, \\
\quad \quad x \leftarrow P_a ; Q & \quad \text{create new } a, \text{ spawn } P_a, \text{ cont. as } Q_a, \\
\quad \quad c \leftarrow d & \quad \text{connect } c \text{ with } d \text{ and terminate} \\
\end{cases}
\]

![Figure 1: Polarized linear session types](image1)

![Figure 2: Linear process expressions](image2)
Next, the code which implements a non-empty priority queue. The elem is decreasing priority. We maintain the priority queue as a linked collection of processes with all the elements of lower priority. In other words, we send cp to all the elements of lower priority. In Figure 3, an empty priority queue is indicated as q connecting a providing process along c with its client along c. It also connects along d to all the elements of lower priority. In other words, we maintain the priority queue as a linked collection of processes with decreasing priority.

```
| del → shift ← recv c ; | % shift direction to send
| c.none ; | % send lab none (queue is empty)
| close c | % close channel c and terminate
```

```
{pc^-A^+ ← A^+, pc^-A^+}
```

c ← elem p ← x, d,

case c of

| ins → q ← recv c ; | % receive channel x to be stored
| y ← recv c ; | % spawn new empty pri. queue
| case (p > q) of
| true → d.ins ; | % pass y with priority q to d
| send d q ; | send d y ;
| c ← elem p ← x, d | c ← elem q ← y, c
| false → e ← elem p ← x, d | e ← elem q ← y, c
| del → shift ← recv c ; | % shift c to send
| c.some ; | % send label some
| send c p ; | % send priority p
| send c x ; | % send highest priority channel x
| send c shift ; | % shift c to recv
| c.d ← forward d as c
```

The typing rules for process expressions, defining the judgment $\Delta \vdash P :: (x : A)$ are given in Figure 3, omitting only the simple cases of sending and receiving basic values (types $\tau \land B^+$, $\tau \to B^+$). Here, $\Delta$ is a sequence of channel declarations $x:A_i$ for pairwise distinct $x$, whose order is irrelevant.

### 2.3 Operational Semantics

The computational meaning of process expressions is given in the form of a substructural operational semantics [20, 25] which relies on multiset rewriting [6]. The rules can be found in Figure 4, again omitting the sending and receiving of basic values.

Because communication is asynchronous, a configuration of multiple interacting processes is described by a multiset of propositions of the form $q_{c', q, c}$ and $pro(c, P)$. Here, $pro(c, P)$ means that a process providing a service along $c$ is executing process expression $P$. We do not explicitly record the channels that are used by $P$, which are simply its free names $fn(P) - \{c\}$. The predicate queue($c', q, c$) describes the state of a message queue $q$ connecting a providing process along $c$ with its client along $c'$. Message queues have a definitive direction which is needed so processes don’t accidentally receive their own messages. The direction of a queue is indicated as $p$ (the provider sends) or $p$ (the client sends). Receipt of a shift will change the direction of the queue. We often elide this direction information when it can be inferred or is insignificant. This bare form of the propositions does not track the types which are necessary for monitoring. Therefore, the queue propositions in the monitored semantics in Figure 6 contain some additional information.

We use $\Omega$ to refer to a multiset of proc and queue propositions, which describe the state of a concurrent computation. Each rule of the operational semantics has the form

$$F_1 \otimes \ldots \otimes F_m \rightsquigarrow \{\exists x_1, \ldots, \exists x_k. G_1 \otimes \ldots \otimes G_n\}$$

which represents a transition from one multiset to another:

$$\Omega, F_1, \ldots, F_m \rightarrow \Omega, [x_1/x_1, \ldots, x_k/x_k](G_1, \ldots G_n)$$
were $a_1, \ldots, a_k$ are freshly chosen names. Persistent propositions which can not be rewritten are preceded by an exponential modality $\exists \alpha$. On the left-hand side, such propositions can represent a condition to be checked.

3. Monitoring Untrusted Processes

In a distributed setting, processes can be compromised by an attacker, may be untrusted, or may be written in a language that cannot statically enforce session types. They may therefore deviate from their prescribed session types. For simplicity, we sometimes refer to all these situations as an "attack" compromising a process.

We use runtime monitors to detect such deviations and attribute blame to rogue processes. In this section, we discuss the adversary and trust model and explain the monitor design. We then formally define the operational semantics for the monitoring mechanism and for the blame assignment.
3.1 System Assumptions

**Adversary and trust models** We assume that processes are distributed across the network and communicate with each other using the message queues. We assume that there is a secure (trusted) network layer which maintains the message queues. In other words, we do not consider network attackers that eavesdrop or intercept messages, or tamper with the message queue. We say that all message queues are trusted. In contrast, all processes are untrusted; any process could be compromised by an attacker. We consider this a conservative assumption—we could easily refine our system to distinguish between trusted and untrusted processes which would lead to more precise blame assignments.

**Monitor capabilities** We assume that the monitor can inspect communications between processes to check session fidelity, but it cannot observe internal operations of the executing processes. Only send, receive, spawn (cut), and forward (identity) requests can be seen by the monitor. This design decision is important because it allows our monitoring techniques to be applied in the situation where we make no assumptions about the internal structure of the communicating processes. The monitor is also trusted.

Monitors can raise alarms and assign blame when messages sent to queues are of the wrong type, which we explain in detail in Section 3.2. In this paper we do not consider error recovery or network failure, which are important, but belong to a different level of abstraction. If a protocol violation is detected and alarm is raised, we assume the whole distributed computation will be aborted.

We have a one-to-one correspondence between channels and processes, since every channel is provided by exactly one process, and each process provides exactly one channel. When a process wants to spawn a new process, it makes a corresponding request to the network layer, which also creates a new message queue of specified type. This type must be taken as prescriptive; violating it will raise an alarm. If we had access to the source of the code executed in the new process, we could type-check it against the given type, and absolve the spawning process. However, this may be difficult to do when executing a new binary, or cloning the current process, so we do not assume we can type-check processes before they start executing.

**Adversary capabilities** Our adversary model has to describe adversary capabilities: what an attacker might do with an untrusted process. The first possibility would be simply to replace the executing code with arbitrary other code. However, this model is too strong: under this attacker model, the adversary can gain send and receive access to any channel in the current system of processes. Under those circumstances, any hope at precise blame assignment has to be abandoned. Instead, we assume that channels are private in that only the processes at the two endpoints of a channel can send to or receive from it. Further, channel names are capabilities that are hard to forge. An attacker only knows the channel names that are given to it by the trusted runtime (e.g., through spawning a new process).

We define the following transition rule (named havoc) to represent an attacker’s action of taking control of a process. The attacker replaces the original process with one of the attacker’s choice. However, the attacker cannot forge channel names, and therefore, the set of channel names in Q is a subset of that in P.

\[
\text{havoc} : \text{proc}(c, P) \rightarrow! (\text{fn}(P) \supseteq \text{fn}(Q)) \sim \text{proc}(c, Q)
\]

Finally, because processes are untrusted, they cannot raise an alarm.

3.2 Monitoring and Blame Assignment

**Placement of the monitor** First, we examine several possible design choices for the monitor and explain our chosen design. Figure 5a shows two processes that offer along channels a and b respectively. Trusted components have a grey background. We further assume that process b uses the client channel a’ (Since processes are uniquely identified by the channel they provide, we use the channel name as a process id.) We link a process to its providing channel at the right end of the queue with a line labeled p. We link a process to the client channels that it uses with lines labeled c. These client channels are the left ends of the queues. Let us assume that process a is a newly spawned process offering a service of type A. Processes a and b can communicate by sending and receiving messages through the queue \(\text{queue}(a', a)\). Because both a and b can be compromised by an attacker, the task of the monitor is to mediate their communications through queue \(\text{queue}(a', a)\): initially, a is supposed to offer a service of type A and b is supposed to use a service of type A through a’.

Figures 5b to 5d show three different monitor designs for mediating communication between processes a and b. Figure 5b illustrates the design where a partial identity process acts as the monitor and mediates communications between a and b. Here, the monitor process m relays messages between a and b and makes sure that the types of the messages entering the queue \(\text{queue}(m', m)\) and the messages entering the queue \(\text{queue}(a', a)\) are consistent. Initially, m and a’ are of type A, which is the type of the service offered by process a. The process \(\text{proc}(m, M)\) is a partial identity process from A to A. In this design, every spawned process (by means of the cut, tensor, or lolli rules), has an accompanying monitoring process generated at the time when it is spawned. The advantage is that the monitor is a process defined within the same language as the rest of the system, so implementing the monitors is straightforward. The drawback is that blame assignment becomes difficult. When a process c sends a message of the wrong type to a client channel m’, the monitor can only raise an alarm after the message reaches the other end of the queue and is observed by the monitor. At this point, the monitor does not know where the message came from, because several processes could legitimately have access to m’. To move forward with this approach, additional bookkeeping would be needed to mark the provenance of messages. A second approach is to place the monitors directly at the ends of the message queues (Figure 5c). The monitor then keeps track of the types of the ends of the queues and checks the message pattern before a message is placed into the queue. The monitor does not need to check the messages leaving the queue, because they will have already been checked. The advantage of this approach is that when a process attempts to send messages of the wrong type, the monitor will raise an alarm before the message is added to the queue. This leads to more precise blame attribution than the first approach.

Finally, in a mutually untrusting environment, it may make sense to place the monitor locally at each process node (Figure 5d). When a process contacts the queue to receive a message, the monitor is contacted instead, and the monitor checks the message before handing it to the process. This is necessary because these messages are sent by other processes. The monitor trusts its local process; when a process sends messages to queues, its local monitor does not check the messages. This approach fits the distributed model more naturally, as the monitor is local to each process. However, this approach suffers from a similar problem as the first approach: the blame assignment is difficult. Moreover, since all processes are a priori untrusted, a separate mechanism would be needed to somehow verify alarms and blame processes.

To help solve the provenance issue mentioned earlier, one can imagine requiring processes to always send a message together with its signature which is generated by the private key of the process. Unfortunately, the signature alone is not enough; the system still suffers from replay attacks. Consider the following scenario: a process b receives a message of type int from process c via chan-
nel $a'$, with c's signature, then later, b sends this message back to $a'$, but now $a'$ expects a channel name. When the monitor eventually raises an alarm, the signature from c is valid, but c is not to be blamed. We either need to record which processes have access to which channels, or need to associate each message with more detailed provenance information. In either case, this design is significantly more complicated than the approach of placing the monitors at the ends of the queues (shown in Figure 5c).

After considering these different approaches, we chose the approach that places monitors at the ends of the queues because it is simple and provides relatively precise blame assignment. Next, we define the formal monitor semantics.

**Monitor Semantics** We augment the operational semantics in Figure 4 to include monitor actions. To assign blame, the monitor maintains a graph data structure that records process spawns throughout the execution of the entire system. We write $G$ to denote the graph (defined below). The nodes in the graph, denoted $N$, are provider channel names. After a process offering along channel $a$ spawns a process offering along channel $b$, and edge $a \rightarrow_{sp} b$ is added to the graph. $G$ is a set of trees.

\[
\begin{align*}
\text{Process graph} & \quad G \ ::= \ (N,E) \\
\text{Edges} & \quad E \ ::= \ \cdot \ | \ E, a \rightarrow_{sp} b \\
\text{messages} & \quad m \ ::= \ n \ | \ \text{end} \ | \ \text{ch}(a) \ | \ \text{lab}(l) \ | \ \text{shift} \\
\end{align*}
\]

We write $m \triangleright A$ to mean that a message pattern $m$ is compatible with type $A$. It describes the pattern of the message that a channel of type $A$ expects to receive (defined below). The monitor uses message patterns to decide whether a message is allowed to be enqueued; the monitor raises an alarm if the message is not compatible with the type expected by the queue.

\[
\begin{align*}
\text{ch}(a) & \triangleright A^+ \otimes B^- \\
\text{lab}(lab_1) & \triangleright \{ lab_1 : A_1^- \} \\
\text{lab}(lab_2) & \triangleright \& \{ lab_2 : A_2^- \} \\
\text{shift} & \triangleright \downarrow A^- \\
\text{end} & \triangleright 1 \\
\text{s} & \triangleright \sigma_{K, \tau} \text{ if verify}(s, \tau, \text{pub Key}(K))
\end{align*}
\]

The last pattern matches a signed value $s$ against the type $\sigma_{K, \tau}$, which requires the monitor to use cryptographic primitives to check whether $s$ is a validly signed value of type $\tau$ using $K$’s public key. This signature is only valid if it is generated using K’s private key. This type is explained in detail in [22]. Our monitoring framework is extensible, so the details of typechecking whether the first-order value (e.g., $\sigma_{K, \tau}$) are a matter of additional theory.

We show the key semantic rules in Figure 6. Most of the rules in Figure 6 involve actions by the monitor. We write monitor$(t)$ to denote a check on condition $t$ by the monitor. The conditions highlighted in gray are only used to prove properties of the monitor; they are not needed to understand the semantics of the monitor nor could they actually be checked by a monitor. For example, an application of the havoc rule is by design an internal transition and not observable. For the rest of this section, conditions highlighted in gray can be safety ignored. We will come back to these rules and explain these conditions in detail in Section 4.

The first five rules define the semantics for a forwarding process where the first two rules apply to both positive and negative types. The only differences between these two rules are the highlighted gray boxes which we will explain in detail in Section 4.2. A forwarding process is only allowed to concatenate the two message queues if their corresponding types match. Rule id$_p$ raises an alarm when a process tries to connect two queues that are expecting messages of different types. The argument of the alarm is the provider channel of the process that forwards. The equality check of the types also ensures that the directions of the two queues are the same, which is an invariant maintained by the queue.

Rule id$_p$ raises an alarm because the forwarding process tries to connect two wrong ends of the queues which would incorrectly identify two providers. Rule id$_p$ raises an alarm because the process $c$ tries to forward to a queue with the wrong provider channel. Well-typed processes will not trigger the last three rules, as these processes cannot be typed using rules in Figure 3.

The rules for cut are augmented, so that new edges are added persistently to the graph $G$. The monitor also records the types of the ends of the newly generated message queue. The monitor will check session fidelity based on these types. These two rules only differ for proof purposes. We omit the cut rules for negative types.

The rules for waiting for a channel to terminate only differ for proof purposes and are the same as rule one_$c$. If a process waits on the wrong channel, the process gets stuck and the monitor does not take any action. This is because the monitor does not inspect any internal actions of the process. Rule one_$sa$ raises an alarm if a process tries to close a channel that either is not the process’ provider channel or is not of type 1.

The two alarm rules in Figure 6 summarize common cases where the monitor raises an alarm. Rule alarm$_s$ raises an alarm when the message sent to the queue is not compatible with the type expected by the queue. This rule also covers the case when a process sends a message to a queue that has the wrong direction. Rule alarm$_p$ raises an alarm when a process uses a provider channel that is not its own provider channel.

We omit additional rules as they are similar to the ones in Figure 6. For the send and receive cases where the monitor’s checks pass, the monitor additionally is in charge of changing the types of the channels recorded at the end of the message queue.

**Blame assignment** When an alarm (alarm$(a)$) is raised, the monitor assigns blame to all the direct ancestors of $a$ in the graph $G$. That is, we find the tree in $G$ that contains $a$, and let $a_1 \rightarrow_{sp} c_2 \cdots c_n \rightarrow_{sp} a$ be the path in that tree from the root to $a$. Then, the processes in the set $\{c_1, \cdots, c_n, a\}$ are jointly blamed. Informally, at least one of the processes in that set must have “havoced”; otherwise, type preservation will ensure that no alarm is raised. We formally show the correctness of the blame assignment in Section 4.
3.3 Motivating Examples

In this example, we illustrate monitoring with a mobile photosharing application, Snapchat, that takes and shares a user’s photos and sends them to some remote entity. To take photos, Snapchat needs to operate the camera. To prevent the Snapchat application from continuously taking and sharing the user’s photos, the camera requires that the user grant Snapchat permission every time Snapchat wants to take and share a photo.

This example contains three main processes: the Snapchat application process, the camera process, and the user process. The monitor checks messages when they are enqueued by a sender. If the Snapchat process deviates from its prescribed session types, for instance, if it tries to send an invalid permission to the camera process, the monitor should raise an alarm.

3.3.1 Types and Encoding

We encode the expected behavior of each process as a session type declaration below.

\[
\text{stype Cam}^+ = \& \{ \text{take : photoPerm}^+ \rightarrow \{ \text{picHandle}^+ \odot \downarrow \text{Cam}^- \} \}
\]

\[
\text{stype User}^- = \& \{ \text{picPerm} : \{ \text{fail : User} \}; \text{succ : photoPerm}^+ \odot \downarrow \text{User} \} \}
\]

\[
\text{stype photoPerm}^+ = \{ \text{once : } \odot \text{ok} \land 1 \}
\]

\[
\text{stype Snap}^+ = \{ \text{share : } \odot \text{pic} \land \downarrow \{ \text{Snap}^+ \} \}
\]

**Camera** The camera process offers a service of type Cam, which is an external choice with a single option: take. After the client selects take, the camera process requires the client to send the channel of a photo permission process of type photoPerm before sending a handle to a picture to the client. Once the permission is received and checked and picture handle is sent, the camera process continues to offer a service of type Cam.

The camera application runs the following CameraFun process. We assume that the takePic function returns a picture handle. The code also details the communication pattern between the camera process and the photo permission process (lines 5–7). Upon receiving a channel pm, the camera process receives a signature from the channel pm. In this example, the camera process does not validate the signature of the permission by itself, but instead relies on the monitor to check the signature, which we will explain later when we discuss the monitoring scenarios.

1 CameraFun : \{ \text{Cam}^- \} =
2 c \leftarrow \text{CameraFun} =
3 \text{case } c \text{ of}
4 \quad \mid \text{take } \rightarrow \text{pm } \leftarrow \text{recv } c ;
5 \quad \quad \text{case } \text{pm of}
6 \quad \quad \quad \mid \text{once } \rightarrow x \leftarrow \text{recv } pm ;
7 \quad \quad \quad \quad \text{wait } pm ;
8 \quad \quad \quad \quad \text{shift } \leftarrow \text{recv } c ;
9 \quad \quad \quad \quad \text{picH } \leftarrow \text{takePic} ;
10 \quad \quad \quad \quad \text{send } c \text{ picH} ;
11 \quad \quad \quad \quad \text{send } c \text{ shift} ;
12 \quad \quad \quad \quad \text{c } \leftarrow \text{CameraFun}

**User** The user process offers a service of type User, which is an external choice with a single option picPerm. When a process needs permission to access the camera, it communicates with the
user process and selects picPerm. The user process then waits to receive a shift before it can send an internal choice label of either fail or succ. After the user sends the fail label, the user process continues to offer a service of type User, without granting its client permission to use the camera. After the user process sends a succ label, it then spawns a new process that provides a service of type photoPerm and sends the new process' channel to its client. The type photoPerm is an internal choice, labeled once. The newly spawned process first sends the label once, then sends a digital signature of a token ok (of type ok) using the camera’s private key, before it terminates. The digital signature serves as an unforgeable authentication token for a permission to access the camera once.

The code snippet that corresponds to the permission process spawned by the user is given below.

1. send pm once;
2. send (sign K,priv(U) ok);
3. close pm;

The function sign will use the user’s private key to sign the abstract type ok. We assume that this permission process has access to the user’s private key as it is spawned by the user process.

**Snapchat** Finally, the Snapchat application offers a service of type Snap, which states that the application offers a single internal choice with label that first sends a picture, then continues behaving as Snap. The double shifts forces a synchronization between Snapchat and its client; Snapchat will not send a second picture before the client has received the first picture in the message queue. The Snapchat application uses services offered by the camera process and the user process. These communications are not specified in the Snap type, but in the Cam and User type.

Next, we define the process for the Snapchat application. The ToSnap process uses c to communicate with the camera process c and u to communicate with the user process and offers the picture sharing service along channel s. We note that convert is a function that converts a picture handle to a string of type pic.

1. ToSnap : {Snaptop ← User, Cam}
2. s ← ToSnap ← u, c =
3. c.take;
4. u.picPerm;
5. send u shift;
6. case u of
7. | fail → shift ← recv c;
8. shift ← recv u;
9. s ← ToSnap ← u, c
10. | succ → s.share;
11. perm ← recv u;
12. send c (y ← (y ← perm)) ;
13. send c shift;
14. picH ← recv c;
15. send s convert(picH);
16. send s shift;
17. shift ← recv c;
18. shift ← recv u;
19. shift ← recv s;
20. s ← ToSnap ← u, c

The Snapchat application first instructs the camera via channel c to take a picture. The ToSnap process then asks the user process for permission and cases on the response from the user. If the user does not grant the permission (line 7), then no picture is sent and the Snapchat process continues to try and send a picture. If the user grants the permission (line 10), the Snapchat application sends its client the label share, receives a channel connecting to a permission process from the user, then forwards the channel to the camera (line 12). It then receives a picture handle from the camera, which is converted to a picture and sent to Snapchat’s client.

### 3.3.2 Monitoring Scenarios

We show three monitoring scenarios to demonstrate how our monitor can detect violations of invariants specified by the session types.

In the scenarios, an attacker tries to take pictures without being granted permissions required by the camera.

**Scenario 1** The Snapchat process is compromised by an attacker. The havocd process does not ask for permission from the user and instead of sending a permission to the camera, sends an integer value (i.e. replacing lines 6-20 of the ToSnap process with (send c n)). Right before this send, the queue associated with the camera process is:

\[ \text{queue}(\text{cm'}, \text{photoPerm}^{+} \rightarrow \{\text{picHandle} \otimes \downarrow \text{Cam}^{-}\}, \rightarrow, \text{cm'}, \text{photoPerm}^{-} \rightarrow \{\text{picHandle} \otimes \downarrow \text{Cam}^{-}\}) \]

The Snapchat process is proc(s, send cm' n ; \cdots). Here, the program variable c is substituted by the concrete channel cm'. The queue is expecting a value of type PhotoPerm which should be a channel. The monitor checks n \(\text{send} \text{photoPerm}^{+} \rightarrow \{\text{picHandle} \otimes \downarrow \text{Cam}^{-}\} \)

The check fails, causing the monitor to raise an alarm (alarm(s)). In this case, s is a node in the monitor’s graph G with no edges attached. Here, blame is assigned to one process, Snapchat (offering along channel s).

**Scenario 2** Snapchat is again compromised. Instead of asking for permission, it tries to spawn a permission process using a fake signature. That is, replacing lines 6-20 of the ToSnap process with

send c (y ← y.once);
send y (sign K,priv(A) ok);
close y

The above send will succeed and spawn a new process:

\[ \text{proc}(d, \text{send} d \text{ once}; \text{send} d (\text{sign} ok); \text{close} d) \]

At this point, the graph G is augmented with \(s \rightarrow_{sp} d\). After the label once is sent, the queue associated with process d is:

\[ \text{queue}(d' : \text{photoPerm}^{+}. \text{once}, d : \text{cm} \otimes \text{ok} \wedge 1) \]

When the process d tries to send a signature, it does not have the user’s key, it cannot generate a value v such that v \(\text{send} \text{cm} \otimes \text{ok}\). When the process d sends (sign ok) to d, the monitor’s check fails and raises an alarm (alarm(d)). The blame is assigned to the set \(\{s, d\}\) as G includes \(s \rightarrow_{sp} d\), and s is the root.

**Scenario 3** The Snapchat process is working appropriately and has gotten a legitimate photo permission from the user. When the Snapchat process spawns a process on line 12 to send the permission to the camera, the spawned process is taken over by an attacker. Instead of proc(d, d \(\rightarrow \text{cm}\)), where the process offering along \text{cm} is spawned by the user, the attacker changes it to proc(d, send d n). The monitor will raise an alarm (alarm(d)) when the above compromised process tries to send an integer value to d, because the monitor is expecting the label once. The graph G includes \(s \rightarrow_{sp} d\), so the blame is assigned to \(\{s, d\}\). This scenario has the same blame assignment as the previous scenario, even though two different processes are compromised. One interesting point is that in this scenario, Snapchat actually has the right permission. The attacker effectively launched a denial of service attack.

### 4. Metatheory

We first define several properties of the monitor, then we prove that our monitor satisfies those properties. The main challenge in constructing these proofs lies in identifying the invariants maintained at runtime in the presence of havocd processes. Formalizing these invariants and proving that the invariants hold at runtime is essential for proving the blame correctness theorem.
4.1 Properties of the Monitor

We identify four high-level properties that the monitor should satisfy: correctness of the blame assignment, minimality of the blame assignment, the fact that well-typed processes are not blamed, and transparency of the monitor.

The correctness of the blame assignment is defined as follows. Recall that $\Omega$ is the multiset of processes and queues describing the current state of computation. We say that it is correct to blame a set of processes if at least one of the processes in the set has made a havoc transition. As some of the examples below show, we cannot in general narrow it down to a single process because the monitors can only observe messages and not anything regarding the internal state of process. We write $\vdash \Omega : wf$ to denote that the state $\Omega$ is well-typed, formally defined in Section 4.2. This well-typeness requires that all processes in $\Omega$ be typed using typing rules in Figure 3 and that the use of the channel names in $\Omega$ satisfy linearity constraints.

Definition 1 (Correctness of blame). A set of processes $N$ is correct to be blamed w.r.t. the execution trace $T = \Omega, G \rightarrow^* \Omega'$ with $\vdash \Omega : wf$ if there is a $b \in N$ such that $b$ has made a havoc transition in $T$.

Second, if all processes are well-typed to begin with and no process is compromised at runtime, then the monitor should not raise an alarm. This property shows that a havoc transition is necessary for the monitor to halt the execution and assign blame.

Definition 2 (Well-typed configurations do not raise alarms). Given any $T = \Omega, G \rightarrow^* \Omega', G'$ such that $\vdash \Omega : wf$ and $T$ does not contain any havoc transitions, there does not exist an $a$ such that $\text{alarm}(a) \in \Omega'$.

Third, the set of processes that the monitor assigns blame to should be as small as possible. An algorithm that always blames all processes in the case of an alarm is correct (according to our definition), but not minimal. Formalizing minimality requires a theory of observational equivalence with respect to a class of monitors and is beyond the scope of this paper. However, as some of our examples illustrate, including ancestors of a for $\text{alarm}(a)$ in the blame set is required. Since we do not include any processes beyond these, we conjecture that our blame sets are indeed minimal.

Finally, the monitor should not change the behavior of well-typed processes. We write $\rightarrow^* \rightarrow^-$ to denote the operational semantics without the monitor. If the initial configuration is well-typed and no process is compromised, then executing the configuration with and without the monitor should yield the same result.

Definition 3 (Monitor transparency). Given any $T = \Omega, G \rightarrow^* \Omega', G'$ such that $\vdash \Omega : wf$ and $T$ does not contain any havoc transitions. Then $\Omega(\rightarrow^*) = \Omega''$, where $\Omega''$ is obtained from $\Omega'$ by removing typing information from queues.

4.2 Validating Properties of our Monitor

We prove that our monitor assigns blame correctly, does not blame well-typed processes, and is transparent. We illustrate the minimalty of the blame assignment of our monitor through examples. For the rest of this section, we call a process that has made a havoc transition, or is a descendant of a process that has made a havoc transition a havocd process.

Augmenting the configuration To formally define and analyze the properties of our monitor, we augment the operational semantics with more information about the actions of havocd processes.

We write $\Omega_Q$ to denote a list of queues, and $\Omega_P$ and $\Omega_H$ to denote a list of unhavoced and havocd processes respectively.

First, we use an additional graph $H$ to keep track of havocd processes. Graph $H$ should be a sub-graph of $G$ and each tree in $H$ is rooted at a provider channel name of a process that has made a havoc transition. When a process makes a havoc transition, a new node is added to $H$, denoted by $!(\text{ havoc}(a))$ in the operational semantic rule for havoc (highlighted in gray in Figure 6). When a process offering along $a$ spawns a new process $b$ and $a$ is a node in $H$, then an edge from $a$ to $b$ is added to $H$, denoted by $!(H(a \rightarrow_{sp} b)))$ (e.g., in cut* rule in Figure 6). When $b$ is a node in $H$ (denoted $b \in H$), $b$ is a havocd process.

Second, because havocd processes may violate linearity constraints, we define $\Theta$ to denote the set of channel names that are used by havocd processes. Channels in $\Theta$ could potentially have aliases or be dangling references to closed or forwarded channels.

The small-step operational semantic rules are either of the form $\Omega, G, H, \Theta \rightarrow \Omega', G', H', \Theta'$ or $\Omega, G, H, \Theta \rightarrow \_, H, \text{alarm}(a)$.

Since we assume a global abort, we do not write out the full configuration in the second kind of transition. In these rules, $H$ and $\Theta$ are present purely for the purpose of proving metatheorems. The set of channel names in $\Theta$ may increase because compromised processes generate dangling pointers via closing a channel or forwarding a channel. In Figure 6, rules id, one, and two, augment $\Theta$ and generate two dangling processes names. In Figure 6, rule id is different from id, in that the forwarding process is a havocd process in the latter case. Similarly, the two cut rules differ in whether the process that spawns a new process is a havocd process.

It is easy to show that $\Omega, G, H, \Theta \rightarrow \Omega', G', H', \Theta'$ implies $\Omega, G \rightarrow \Omega'', G''$ and $\Omega, G, H, \Theta \rightarrow \_, H', \text{alarm}(a)$ implies $\Omega, G \rightarrow \_, \text{alarm}(a)$.

Typing the configurations The typing rules for states use several contexts. The context $\Psi$ maps channel names to their types. We note that all channels in $\Psi$ are linear. The queue channel typing context $\Gamma$ contains type bindings for both ends of a queue. For instance, $d : A \Leftarrow d : B$ denotes that the provider channel of the queue has type $B$ and the client channel of the queue has type $A$.

We define two functions client($\Gamma$) and provider($\Gamma$). The function client($\Gamma$) returns the channel typing context that contains only client channels in $\Gamma$. That is, for every $d : A \Leftarrow R$ in $\Gamma$, client($\Gamma$) returns $d : A$. Similarly, we define provider($\Gamma$) to return the context that contains only provider channels in $\Gamma$: for every $d : A \Leftarrow R$ in $\Gamma$, provider($\Gamma$) returns $R$. We write $\omega$ to denote that two contexts are disjoint in their domains. We write $\Psi \not\subeq \Psi'_1 \cup \cdots \cup \Psi'_{n}$ to mean that any two contexts $\Psi_i$ and $\Psi_j$ have disjoint domains, where $i, j \in [1, n]$. The typing rules for queues, processes, and configurations are summarized in Figure 7. The judgment $\Psi \vdash \Lambda, \Omega_Q : \Gamma$ means that queues in $\Omega_Q$ are well-typed under $\Psi$. The context $\Gamma$ contains the provider/client channel types of all the queues in $\Omega_Q$. The rule for checking a single queue uses the judgment $\Psi \vdash \text{queue}(a, A, q, R)$ (shown in Figure 8) to check that the elements in $q$ are exactly the channel names in the domain of $\Psi$ and that the types of elements in $q$, as specified in $\Psi$, respect the types of the two ends of the queue. The last rule ensures channels names are used linearly.
The typing judgment for unhavoced processes is \( H; \Theta \models \Omega_H \). The rules for this judgment check that processes in \( \Omega_H \) are not in \( H \) and that the processes use channels in \( \Theta \), and provide services on channels in \( \Psi_o \). The rule is similar to the last rule for queue typing: it ensures that processes only use channel names linearly.

The typing judgment for havoced processes is \( H; \Theta \models \Omega_H \). As before, \( H \) is the graph tracking havoc transitions and the context \( \Psi_o \) maps channel names to their types. The context \( \Theta \) contains all the channels used by processes in \( \Omega_H \) (not necessarily linearly). The rule for checking a single havoced process \( \text{proc}(c, P) \) checks that a free name used in the process is either in \( \Theta \) or the provider channel of the process, or a direct ancestor of \( c \) in \( H \). The last rule type checks a set of havoced processes. Here, the variable contexts are shared across the premises of that rule. This means that havoced processes do not need to use channel names linearly.

The top-level typing rule for the entire state is \( H; \Theta \models \Omega \). We write \( \Psi^* \) to denote the context resulted from erasing types in \( \Psi \). The context \( \Psi^* \) only contains the channel names in \( \Psi \). The key points of the above typing rule are that: (1) the queues, unhavoced processes, and havoced processes use mutually disjoint sets of channels, (2) havoced processes additionally use a set of dangling channels (in \( \Theta \)), (3) the client channels, including their types, used by queues and processes should match exactly the set of client channels of the queues, and (4) the provider channels of the queues can be split into two disjoint sets, one that matches the set of provider channels, including their types, of unhavoced processes; the other that matches the provider channel names of havoced processes.

By restricting \( H \) and \( \Theta \) to be empty, we obtain the stricter definition of well-typedness of configurations used in our property definitions. We define \( \models \Omega \) as \( \models \Theta \). That is, the configuration \( \Omega \) can be typed under the assumption that no processes have havoced and all channel names are used linearly.

We make the implicit well-formedness assumption that the context \( \Gamma \)’s provider and client channels are disjoint from each other. That is client(\( \Gamma \)) \( \ominus \) provider(\( \Gamma \)). The queue typing rules only generate well-formed \( \Gamma \’s.

**Blame Correctness Theorem** The fact that our monitor assigns blame correctly (Definition 1) is a corollary of Theorem 1 (see below).

**Theorem 1** (Alarm).

1. If \( \Psi ; \models \Omega \) and \( \Psi ; \models \Omega , \emptyset , \emptyset \) then \( H; \Theta \models \Omega \).
2. If \( \Psi ; \models \Omega \) and \( \Psi , \emptyset \) then \( H; \Theta \models \Omega \).

The above theorem states that from an initial configuration, a well-typed configuration can make a series of transitions to either another well-formed configuration, or a state where an alarm is raised on a process \( a \) that is in the \( H \) graph. Based on the semantics, \( H \) only contains processes that either have made a havoc transition, or are descendants of one that has made a havoc transition, and \( H \)
is a subgraph of $G$, and therefore, the blame assigned to $a$ and $a$’s
direct ancestors in $G$ is correct.

Another corollary of Theorem 1 is that well-typed processes are
not blamed: if the configuration is well-formed and no process is
compromised, then the monitor will not raise any alarm. This is
easy to prove because if there is an alarm associated with $a$, then
$a$ must be in $H$. However, when no process havoces, $H$ remains
empty; a contradiction.

The correctness proof for the blame assignment is similar to that
of a preservation proof. The key lemma is Lemma 2, which states
that if a well-typed configuration makes a transition, then it either
takes steps to another well-formed configuration, or an alarm is raised on
a process that is in the $H$ graph.

Lemma 2 (Alrm (one step)). If $H, \Theta = \Omega$ and $\Omega, G, H, \Theta \rightarrow$
then
1. $\Omega, G, H, \Theta \rightarrow \Omega', G', H', \Theta'$ implies $H'; \Theta' = \Omega'$
2. $\Omega, G, H, \Theta \rightarrow \rightarrow H$, alarm$(a)$ implies $a \in H$.

Using the above lemma, we can prove Theorem 1 which
considers a sequence of transitions. The proof is done by induction on
the length of the trace.

Minimality We show an example scenario (Figure 9) where the
monitor cannot tell which process in the blame set is compromised.
Initially, no process is compromised. The initial configuration is
$\Omega_0$, which contains a process $b$ and a queue. We omit other pro-
cesses and queues, as they are not important for this example. The
process offering along $b$ spawns a process, which spawns another
process that sends an integer to $a$. We consider three execution
traces $T_1$, $T_2$, and $T_3$. We mark the havoc step using a superscript
$H(x)$, where $x$ is the channel name of the process that makes the
havoc step. The configurations $\Omega_1$ and $\Omega_2$ are the intermediary
configurations in those traces. We use the convention that $\Omega_1$ contains a
compromised process; instead of sending $1$ to $a$, the compromised
process tries to send a channel to $a$. When this send is executed, the
monitor will raise an alarm.

All of the three traces end up raising the same alarm. When the
alarm is raised, the graph $G$ that the monitor maintains contains a
chain denoting that $b$ has spawned $b_1$ and $b_1$ has spawned $b_2$. Here,
$T_1$, $T_2$, and $T_3$ are observationally equivalent to the monitor. We
define two traces to be observationally equivalent to the monitor if
the lengths of the traces are the same and for any two states at the
same position on the traces, all the channel names of the queues are the
same and the graphs $G$ are the same. This definition is reasonable because the monitor state consists entirely of the queue
typing and the graph. In all three scenarios, the monitor assigns
blame to the set $\{b, b_1, b_2\}$. This blame is minimal as the monitor
cannot distinguish which trace it is on when an alarm is raised.

Transparency Our monitor is transparent: it does not alter the be-
havior of well-formed configurations. The proof is done by exam-
ing how each monitor check is applied to well-formed configura-
tions. Since well-formed configurations do not have havoced pro-
cesses ($H$ is empty), all processes and queues are well-typed using
the stricter typing rules. The fact that the monitor checks never fail
can be obtained by inverting the typing judgments of the relevant
queues and processes.

Preservation and progress? The above theorem is not the same
as the preservation lemma proven in Pfenning and Griffith [21]. This
is because the configuration typing rules maintain looser invariants
than those in Pfenning and Griffith [21]. More concretely,
the preservation allows violation of linearity by havoced processes,
such as sharing channels, using dangling channels, and creating cir-
cular dependencies of queues. These violations are not possible if
no havoc transitions are made.

$P = \text{send } a' \text{ shift; wait } a'$
$\Omega_0 = \text{proc } b, x_1 \leftarrow (x_2 \leftarrow (\text{send } a' \cdot 1; P; \text{close } x_2);$
\hspace{1cm} wait $x_2; \text{close } x_1);
\hspace{1cm} \text{wait } x_1; \text{close } b)
\hspace{1cm} \bowtie \text{queue}(a' : \text{int } \rightarrow \uparrow 1, \cdot, a : \text{int } \rightarrow \uparrow 1)
\hspace{1cm} \Omega_0' = \text{proc } b, x_1 \leftarrow (x_2 \leftarrow (\text{send } a' (y \leftarrow \text{close } y);$
\hspace{1cm} \hspace{1cm} P; \text{close } x_2);$
\hspace{1cm} \hspace{1cm} \text{wait } x_2; \text{close } x_1);
\hspace{1cm} \hspace{1cm} \text{wait } x_1; \text{close } b)
\hspace{1cm} \bowtie \text{queue}(a' : \text{int } \rightarrow \uparrow 1, \cdot, a : \text{int } \rightarrow \uparrow 1)
\hspace{1cm} \Omega_1 = \text{proc } b, \text{wait } b_1; \text{close } b)
\hspace{1cm} \bowtie \text{queue}(b_1 : 1, \cdot, b_1 : 1)
\hspace{1cm} \text{proc } (b_1, x_2 \leftarrow (\text{send } a' (y \leftarrow \text{close } y); P; \text{close } x_2);$
\hspace{1cm} \hspace{1cm} \text{wait } x_2; \text{close } b_1);
\hspace{1cm} \bowtie \text{queue}(a' : \text{int } \rightarrow \uparrow 1, \cdot, a : \text{int } \rightarrow \uparrow 1)
\hspace{1cm} \Omega_1' = \text{proc } b, \text{wait } b_1; \text{close } b)
\hspace{1cm} \bowtie \text{queue}(b_1 : 1, \cdot, b_1 : 1)
\hspace{1cm} \text{proc } (b_1, x_2 \leftarrow (\text{send } a' (y \leftarrow \text{close } y); P; \text{close } x_2);$
\hspace{1cm} \hspace{1cm} \text{wait } x_2; \text{close } b_1);
\hspace{1cm} \bowtie \text{queue}(a' : \text{int } \rightarrow \uparrow 1, \cdot, a : \text{int } \rightarrow \uparrow 1)
\hspace{1cm} \Omega_2 = \text{proc } b, \text{wait } b_1; \text{close } b)
\hspace{1cm} \bowtie \text{queue}(b_1 : 1, \cdot, b_1 : 1)
\hspace{1cm} \text{proc } (b_1, \text{wait } b_2; \text{close } b_1);
\hspace{1cm} \text{queue}(b_2 : 1, \cdot, b_2 : 1)
\hspace{1cm} \text{proc } (b_2, \text{send } a' (y \leftarrow \text{close } y); P; \text{close } b_2);
\hspace{1cm} \text{queue}(a' : \text{int } \rightarrow \uparrow 1, \cdot, a : \text{int } \rightarrow \uparrow 1)
\hspace{1cm} G = b \rightarrow_{sp} b_1 \rightarrow_{sp} b_2
\hspace{1cm} \text{For simplicity, we omit G from the traces.}
\hspace{1cm} T_1 = \Omega_0 \rightarrow \rightarrow H(b)
\hspace{1cm} \Omega_0' \rightarrow \rightarrow \Omega_2' \rightarrow \rightarrow \text{alarm}(b_2)
\hspace{1cm} T_2 = \Omega_0 \rightarrow \rightarrow H(b_1)
\hspace{1cm} \Omega_1 \rightarrow \rightarrow \Omega_2 \rightarrow \rightarrow \text{alarm}(b_2)
\hspace{1cm} T_3 = \Omega_0 \rightarrow \rightarrow \Omega_1 \rightarrow \rightarrow \Omega_2 \rightarrow \rightarrow \text{alarm}(b_2)
\hspace{1cm} N = \{b, b_1, b_2\}

Figure 9: An example illustrating minimality of blame assignment.

For similar reasons, well-typed configurations may be stuck (do
not have progress). We show a few examples of violations that
result in stuck system states in Figure 10. In the first example,
process $c$ has havoced. Notice that $c$ uses the client channel $a'\cdot$
multiple times. In the first step, $c$ spawns a new process $d$ and
sends the client channel $a'\cdot$ of that new process to the queue $a'$. In
the second step, process $a$ receives the client channel $d'$, and
waits to receive from $d'$. At the same time process $d$ waits on
$a'$. Now processes $a$ and $d$ wait on each other and the system
is stuck. In the second example, both processes $a$ and $c$ have havoced.
Otherwise, they can’t both have access to the client channel $d'$.
(For $a$ and $c$ to share a channel, we can use the same trick used
in the previous example: $c$ can be spawned by $a$ and contains all
the channels that $a$ can access). Process $c$ forwards $d'$ to $c$, now $d'$
disappears. However, process $a$ can still attempt to operate on $d'$.
Since $d'$ doesn’t exist anymore, $a$ is stuck.
Circular dependency:
\[
\text{proc}(c, \text{send } a' \leftarrow (y \leftarrow \text{recv } a' : P_{(x,y)})) ; Q) \otimes
\]
\[
\text{queue}(c':A^+ \leftarrow B', c : A^+ \leftarrow B') \otimes
\]
\[
\text{proc}(a, z \leftarrow \text{recv } a ; z' \leftarrow \text{recv } z ; R_{z}) \otimes
\]
\[
\exists d, \exists x. \text{proc}(c, Q) \otimes \text{queue}(c':B', d, a : A^+ \leftarrow B') \otimes
\]
\[
\text{proc}(a, z \leftarrow \text{recv } a ; z' \leftarrow \text{recv } z ; R_{z}) \otimes
\]
\[
\text{queue}(c':A^+, c : A^+ \leftarrow d', \text{send } d' ; P) \otimes
\]
\[
\text{proc}(a, \text{send } d' ; P)
\]

Dangling channel:
\[
\text{queue}(c': A^+, c : A^+ \leftarrow P) \otimes \text{proc}(c, c \leftarrow d') \otimes
\]
\[
\text{queue}(c': A^+, c : A^+ \leftarrow P) \otimes \text{proc}(a, \text{send } d' ; P)
\]

Figure 10: Example configurations.

5. Related Work

There is a rich body of work on higher-order contracts and the correctness of blame assignments in the context of lambda calculus since Fung and Felleisen [13] first introduced higher-order contracts and the concept of blame. Wadler and Fung [31] defined the first blame calculus and established that blame always lies with the less-precisely-typed-code. More comprehensive theorems about the correctness of blame assignment have been proposed by Dimoulas et al. [9, 10].

Subsequent work on gradual typing that considers systems with both static and dynamic typing also uses “blame always lies with the less-precisely typed code” as a criteria for correctness. For instance, Ahmed et al. [1] developed a blame calculus for a language that integrates parametric polymorphism with static and dynamic typing. Fennell and Thiemann [12] proved a blame theorem for a linear lambda calculus with type Dynamic. Most recently, Wadler [30] surveys the history of the blame calculus and presents the latest developments. Keil and Thiemann [17] developing a blame assignment for higher order contracts that includes intersection and union contracts. Siek et al. [24] develop three calculi for gradual typing and relate them in an effort to unite the concepts of blame and coercion.

Compared to the body of work mentioned above, our work focuses on distributed systems, where processes communicate with each other via message queues. At a high-level, we can relate our adversary model to the work on blame assignment as follows. Each process can be viewed as a program written in dynamically typed language. Our monitor enforces the coercion of session types by observing the communications between the processes. Our blame assignment is the ancestor closure of the process that has directly caused an alarm to be raised. The set of processes that is blamed always includes a compromised process (all of its descendants are likely to have been compromised too). If we view the compromised process as a less-precisely-typed program, our correctness of blame property is similar to the notion proposed in Wadler and Fung [31]: blame always falls on less-precisely-typed programs.

Disney et al. investigated behavior contracts that enforce temporal properties for modules [11]. 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 are not as expressive as the temporal higher-order contracts proposed by Disney et al, but our system is concurrent, while their system does not consider concurrency.

The work most closely related to ours is on multi-party session types [2, 7, 27]. Bocchi et al. [2]'s work and Chen et al. [7]'s work assume a similar asynchronous message passing model as ours. Their monitor architecture is also similar to ours; monitors are placed at the ends of the communication channels and monitor communication patterns. One key difference is that their monitors do not raise alarms; instead, the monitors suppress “bad” messages and move on. Our monitors halt the execution and assign blame. Consequently, this work does not have theorems about blame assignment which are central to our work. Using global types, their monitors can additionally enforce global properties such as deadlock freeness, which our monitors cannot. Our work supports higher-order processes, while their work is strictly first-order. Recently, Thiemann [27] has taken steps towards integrating gradual typing and a two-party session-type system. There, the communications are synchronous. A proxy process acts as a monitor for a forked dynamically typed process, mediating the communications of the forked process. This proxy can be viewed as a partial identity process that eta-expands the coerced session type. As discussed in Section 3.2, we moved away from that design and chose a more local monitoring strategy to achieve more precise blame assignment. Neither our system nor their system prevents deadlock. We additionally proved both the correctness of blame and the monitor transparency properties of our monitor.

6. Conclusion

We have presented a system for monitoring and blame assignment for session types based on a Curry-Howard interpretation of linear logic. Communication is asynchronous, through message queues, and our monitors manage these queues based on a types-as-contracts interpretations of the session types. Monitored communications include spawning of processes (logically the cut rule), forwarding between processes (logically the identity rule), internal and external choice, as well as passing of channels along channels which gives the language some higher-order aspects. Our adversary or failure model allows a process to make arbitrary transitions to ill-typed code in a step we call havoc, subject to the constraint that a havocd process can not gain access to other private channels. Proving correctness of the blame assignment has been technically challenging because execution may continue with havoced processes for many steps before an observable type violation occurs (if at all). Process configurations with havoced processes and, among other invariants, may violate the linearity constraints which force channels to have exactly one provider and one client.

We have omitted shared channels from this presentation. An in-depth discussion of their typing and operational reading can be found in Pfennig and Griffith [21]. A shared channel is always provided by a unique persistent process and can be used by arbitrarily many other processes. We can only use a shared channel of type \(\uparrow A\) by spawning a new linear session of type \(A\), leaving the persistent process in place. This new linear session would be monitored with the techniques described here. Monitoring and blame assignment of the shared channels themselves is actually easier, since persistent message “queues” are always empty or singletons.

We do not consider the mechanics of how to connect to offered services, be they linear or persistent. As long as offered services come with a type (which is required in any case for session-typed communication), monitoring can then commence as described in this paper once a connection has been established.

One remaining interesting technical question is whether our blame assignment identifies a minimal set of potential culprits, given the kinds of observations we allow the monitor to make. We conjecture that this is so, but currently we lack the theory of observational equivalence in this setting to answer it definitively. Another interesting question is whether we can be more precise if we
allow more global observations about the state of the computation, for example, the geometry of the connections. Precision here can refer both to smaller blame sets and to discovering more violations which are undetected by simple type-checking. Along the latter dimension, it would also be interesting to add contracts or dependent types to the language so that more violations can be discovered (potentially both statically and dynamically).

Acknowledgement

This research was supported in part by NSF grant CNS1423168.

References


A. Summary of Monitoring Rules

Figure 11 and 12 show all the monitoring rules. The conditions in gray boxes are only used in the proofs of meta-theories. The monitor only takes effect when a process executes an instruction that sends a message. Rule alarmp and alarmm are two common rules for cases when the message being sent to a queue is not compatible with the type that the queue (monitor) is expecting. Under those situations, the monitor raises an alarm. Rule alarmm applies in situations when a process tries to send a message to a provider channel that is different from its own provider channel. The monitor raises an alarm because a well-behaved process does not have access to provider channel names for other processes.

Rules one_scp and plus_scp are similar to alarmm. Rules for sending channels to another process are split into two stages. Take tensor_s from Figure 4 as an example, this rule is split into tensor_s1 (or tensor_sh1) and tensor_s2 (or alarmm or alarmm). In the first stage, a new process is spawned, but the new channel is not sent to the queue of the recipient. The resulting sending process still needs to send the newly generate channel. Further, the newly generated queue does not have typing specifications at the end of this stage. In the second stage, the sending process sends the channel and the monitor checks the channel against the type of the
Lemma 8 holds.

**Proof.** By induction on the structure of the derivation $\psi \vdash \text{proc}(c, P) :: c:A$.

**Lemma 4 (Inversion).**

1. If $\psi \vdash \text{queue}(c':A, q, R) : \Gamma$ then $\psi = \psi_1, \psi_2 \vdash \text{queue}(c':A, q, R)$, and $\Gamma = \Gamma', \psi' = c:A \in \mathcal{R}$.
2. If $H; \psi \vdash \text{proc}(c, P) : \psi_3$ then $\psi = \psi_1, \psi_2 \vdash \text{proc}(c, P) :: (c:A), c \notin H$, and $\psi_0 = \psi$, $c:A$.
3. If $H; \psi \vdash \text{proc}(c, P) : \psi$ then $H; \psi \vdash \text{proc}(c, P) : \psi'$, $\text{fn}(P) \subseteq \Theta \cup \{c\} \cup \text{ancestors}(H, c)$, $c \in H$, and $\psi = \psi'$.

**Proof.** By induction on the derivations.

**Lemma 5 (Weakening H1).** If $H; \psi \vdash \text{proc}(c, P) : \psi'$ then given any $c.s.t. c \notin \text{dom}(\psi')$ $H[N:=N, c]; \psi \vdash \psi'$. 

**Proof.** By induction on the derivation $H; \psi \vdash \text{proc}(c, P)$.

**Lemma 6 (Weakening H2).** If $H; \psi \vdash \text{proc}(c, P) : \psi$ then given any $c. H[N:=N, c]; \psi \vdash \psi$. 

**Proof.** By induction on the derivation $H; \psi \vdash \text{proc}(c, P)$.

**Lemma 8 (Queue concatenation).**

1. If $\psi_2 \vdash \text{queue}(c':A_2, p, c: B), \psi_1 \vdash \text{queue}(d':B, q, R)$, and $\psi_1 \vdash \psi_2$ then $\psi_1, \psi_2 \vdash \text{queue}(c':A_2, p\cdot q, R)$.
2. If $\psi_2 \vdash \text{queue}(c':A_2, p, c: B), \psi_1 \vdash \text{queue}(d':B, q, R)$, and $\psi_1 \vdash \psi_2$ then $\psi_1, \psi_2 \vdash \text{queue}(c':A_2, p\cdot q, R)$.

**Proof.** By induction on the length of the queue $c$ in 1, and by induction on the length of the queue $d'$ in 2.

**Lemma 9 (Reverse queue typing).**

1. $\psi \vdash \text{queue}(c':A^+, q, c:B^-)$ implies $\psi \vdash \text{queue}(c':A^+, \text{shift}, q, c:B^-)$
2. $\psi \vdash \text{queue}(c':B^+, q, c:C^-)$ implies $\psi \vdash \text{queue}(c':B^+, \text{shift}, c:A^-)$
3. $\psi \vdash \text{queue}(c':B, q, c:1)$ implies $\psi \vdash \text{queue}(c':B, q, c: \text{end})$
4. $\psi \vdash \text{queue}(c':A^+, \text{end}, c:B^-, q, c:C^-)$ implies $\psi, b':A^+ \vdash \text{queue}(c':B^-, b', q, c:C^-)$

**Proof.** By inversion of (1), and inversion Lemma 4, we have the following. Each box is a typing derivation, read from top to bottom.

**C. Proof of Lemma 2**

To make the relations between contexts clearer, we use tables to represent the premises of the above rule as follows. Each box is a typing derivation, read from top to bottom. The derivations as well as the relations between contexts are labeled and can be referred to later in the proofs.

The typing judgment in Figure 7 is represented as follows.

![Diagram](image-url)
By inversion of (3) and (4) and equality constraints of contexts (5) and (11)

\[ \Psi \] by (13) and Lemma 8

By (5)

We can construct the typing derivation

\[ \Omega \]

By assumption

\[ \Omega = \Omega_Q, \text{queue}(c' : A_1, p, c : A_2), \Omega_P, \Omega_H \]

We can construct the typing derivation \( H ; \Theta \vdash \Omega' \) as follows

\[ c' : A_1 \leftarrow c : A_2 \]

By Lemma 4 on (7)

\[ \text{fn}(c \leftarrow d') \subseteq (\Psi_H)^* \]

\[ H ; (\Psi_H)^*, \Theta, c, \text{ancestors}(H, c) \]

\[ \text{provider}(\Gamma_Q), c' : A_1, d' : B_1 \]

By (5) and (11)

\[ \Psi_H = \Psi_H^1, d' : B_1 \]

By (5) and (12)

\[ \text{client}(\Gamma_Q), c' : A_1 \]

By (9), (10), and (11)

\[ \text{provider}(\Gamma_Q), c : A_2, d : B_2 \]

By inversion of (1) and inversion Lemma 4, we have the following.

Each box is a typing derivation, read from top to bottom.

\[ c : B \]

\[ \text{split into three contexts in gray (5)} \]

\[ \text{split into four contexts in gray (5)} \]

\[ \text{split into three contexts in gray (15)} \]

\[ \text{split into three contexts in gray (5)} \]

\[ \text{split into four contexts in gray (5)} \]
By inversion of (3) and (4)
\[ \Psi_1^d \vdash \text{queue}(c':A_1, p, c:A_2) \]
and \[ \Psi_2^d \vdash \text{queue}(d':B_1, q, d:B_2) \]  
By (15) and Lemma 8
\[ \Psi_1^d, \Psi_2^d \vdash \text{queue}(c':A_1, p \cdot q, d:B_2) \]  
By id_, rule  
\[ \Omega' = \Omega_Q, \text{queue}(c':A_1, p \cdot q, d:B_2), \Omega_P, \Omega_H \]  
\[ \Theta' = \Theta, d' \]  
We can construct the typing derivation \( H; \Theta \vdash \Omega' \) as follows

\[
\begin{array}{c}
\text{By inversion of (1), and inversion Lemma 4, we have the following.} \\
\text{split into three contexts in gray (13)} \\
\Phi_Q \vdash \Psi_1^d, \Psi_2^d \\
\Phi_Q \vdash \Psi_1^d, \Psi_2^d, \Psi_3^d \\
\Phi_Q \vdash \Psi_1^d, \Psi_2^d, \Psi_3^d, \Psi_4^d \\
\Phi_Q \vdash \Psi_1^d, \Psi_2^d, \Psi_3^d, \Psi_4^d, \Psi_5^d \\
\Phi_Q \vdash \Psi_1^d, \Psi_2^d, \Psi_3^d, \Psi_4^d, \Psi_5^d, \Psi_6^d \\
\text{case id_}. \\
\text{queue}(c':A_1, p, c:A_2) \otimes \text{proc}(c, c \leftarrow d') \otimes \text{queue}(d':B_1, q, d:B_2) \\
\oplus \text{monitor}(A_2 \neq B_1) \rightarrow \{\text{alarm}(c)\} \\
\text{By assumption} \\
\Omega = \Omega_Q, \text{queue}(c':A_1, p, c:A_2), \text{queue}(d':B_1, q, d:B_2), \\
\Omega_P, \text{proc}(c, c \leftarrow d'), \Omega_H \\
H; \Theta \vdash \Omega \\
\text{Let us assume that } c \notin H. \text{Next, we derive a contradiction.} \\
\text{By inversion of (1), and inversion Lemma 4, we have the following.} \\
\text{each box is a typing derivation, read from top to bottom.} \\
\text{case havoc.} \\
\text{proc}(c, P) \otimes ![\text{fn}(Q) \subseteq \text{fn}(P)] \rightarrow \text{proc}(c, Q) \otimes ![\text{havoc}(c)] \\
\text{By assumption} \\
\Omega = \Omega_Q, \Omega_P, \text{proc}(c, P), \Omega_H \\
H; \Theta \vdash \Omega \\
\text{By inversion of (1), and inversion Lemma 4, we have two cases.} \\
\text{one is } c \in H \text{ and the other is } c \notin H. \\
\text{Subcase } c \notin H \\
\Phi_Q \vdash \Omega_Q \\
\text{By equality constraints of contexts (5) and (11)} \\
B = B_1 \text{ and } B = A_2, \text{ which contradicts with } B_1 \neq A_2. \\
\text{case id_}. \\
\text{proc}(c, P) \otimes ![\text{fn}(Q) \subseteq \text{fn}(P)] \rightarrow \text{proc}(c, Q) \otimes ![\text{havoc}(c)] \\
\text{By assumption} \\
\Omega = \Omega_Q, \Omega_P, \text{proc}(c, P), \Omega_H \\
H; \Theta \vdash \Omega \\
\text{By inversion of (1), and inversion Lemma 4, we have the following.} \]

\[
\begin{array}{c}
\end{array}
\]
and

\[ \text{client}(\Gamma_Q^c) \]

split into four contexts in gray (3)

\[ \Psi_Q^c \vartriangleleft \Psi_U^c \vartriangleleft \Psi_\text{proc} \vartriangleleft \Psi_H^c \vartriangleleft \Theta \]

By (5) and Lemma 3
\[ \text{fn}(P) = \Psi_c^c \cup \{c\} \]

By (10), \( \text{fn}(Q) \subseteq \text{fn}(P) \), and apply the typing rule
\[ H[N := N, c]; (\Psi_c^c, \Psi_H^c)^* ; \Theta \vdash_h \text{proc}(c, Q) : c : \text{havoc} \]

By (6), Lemma 6 (11)
\[ H[N := N, c]; (\Psi_c^c, \Psi_H^c)^* ; \Theta \vdash_h \text{proc}(c, Q), \Omega_H : c : \text{havoc}, \Psi_H^c \]

By (7)
\[ \Psi_p^c = \Psi_Q^c : c : B \]

By inversion of typing rules,
\[ c \notin \text{dom}(\Psi_p^c) \]

By Lemma 5, (4), and (14)
\[ H[N := N, c]; \Psi_U \vdash_h \Omega_P : \Psi_p^c \]

By havoc rule,
\[ \Omega' = \Omega_Q, \Omega_P, \text{proc}(c, Q), \Omega_H \]

By type rules,
\[ \vdash_h \text{queue}(a, \Psi_c^c, a : A^+) : a : A^+ \]

By (10), substitution, and \( a \) and \( a' \) are fresh
\[ \Psi_1^c \vdash P_a : a : A^+ \]

By cut rule,
\[ \Omega' = \Omega_Q, \text{queue}(a, \Psi_c^c, a : A^+), \]

We can construct the typing derivation \( H' ; \Theta \vdash \Omega' \) as follows.

\[ \text{provider}(\Gamma_Q) \]

\[ \text{client}(\Gamma_Q^c) \]

split into four contexts in gray (3)

\[ \Psi_Q^c \vartriangleleft \Psi_U^c \vartriangleleft \Psi_\text{proc}^c \vartriangleleft \Psi_H^c \vartriangleleft \Theta \]

and

\[ \Psi_p^c : c : B \]

the union of the two gray contexts

equals the following (9)

\[ \text{provider}(\Gamma_Q) \]

Subcase \( c \in H \)

This case can be proven by using the transitivity of \( \subseteq \) to show that
\[ \text{proc}(c, Q) \] can be typed.

case \( \text{cut}^+ \).

By assumption
\[ \Omega = \Omega_Q, \Omega_P, \text{proc}(c, x:A^+) \leftarrow P_a, \Omega_H \]

By inversion of (1), and inversion Lemma 4.

\[ \text{provider}(\Gamma_Q^c) \]

By inversion of (5)
\[ \Psi_1^c \vdash P_a : x : A^+ \]

By type rules,
\[ \vdash_h \text{queue}(a, \Psi_c^c, a : A^+), a : A^+ \]

By (10), substitution, and \( a \) and \( a' \) are fresh
\[ \Psi_1^c \vdash P_a : a : A^+ \]

By cut rule,
\[ \Omega' = \Omega_Q, \text{queue}(a, \Psi_c^c, a : A^+) \]

We can construct the typing derivation \( H' ; \Theta \vdash \Omega' \) as follows.

\[ \text{provider}(\Gamma_Q) \]

17
By queue typing rules,
\[ \vdash_{q} \text{queue}(a':A^+, \cdot, a:A^+) : a':A^+ \Leftarrow a:A^+ \]  
(14)

By (a) is fresh and (7)
\[ \Psi_{H}, a:\text{havoc} \models_{\text{dom}} \Psi_{2}, a: A^+ \]  
(15)

By cut\(^{+}\) rule,
\[ \Omega' = \Omega_Q, \text{queue}(a':A^+, \cdot, a:A^+) \]  
\[ \Omega_P, \text{proc}(c, Q_{a'}) \]  
\[ \text{proc}(a, P_a), \Omega_H \]  
\[ H' = H[E := E, c \rightarrow_{sp} a] \]  
We can construct the typing derivation \( H' ; \Theta \models \Omega' \) as follows.

\[ \Psi_{Q} \models_{\text{q}} \Psi_{I} \models_{\text{q}} \Psi \models_{\text{q}} \Psi_{H} \]  
(2)
\[ \vdash_{h} \text{proc}(a, P_a), \Omega_H \]  
(16)
\[ \vdash_{h} \text{proc}(c, Q_{a'}) \]  
(17)
\[ \vdash_{h} \text{proc}(a, P_a), \Omega_H \]  
(18)
\[ a:\text{havoc}, \Psi'_{I} \]  
(19)
\[ \vdash_{q} \text{queue}(a':A^+, \cdot, a:A^+) \]  
(20)

\[ \vdash_{q} \text{queue}(a', A^+, \cdot, a:A^+) : a':A^+ \Leftarrow a:A^+ \]  
(21)

and

\[ \text{client}(\Gamma_{Q}^o), a':A^+ \]  
(22)

By weakening Lemma 6,7
\[ H[E := E, c \rightarrow_{sp} a]; \Psi_{H}, a':A^+, \Theta \models_{\text{h}} \Omega_H : \Psi^q_{H} \]  
(23)

By weakening Lemma 5 and (4)
\[ H[E := E, c \rightarrow_{sp} a]; \Psi_{H}, a':A^+, \Theta \models_{\text{h}} \Omega_H : \Psi^q_{H} \]  
(24)

\[ \vdash_{q} \text{queue}(a', A^+, \cdot, a:A^+) \]  
(25)

By cut\(^{+}\) rule,
\[ \Omega' = \Omega_Q, \text{queue}(a', A^+, \cdot, a:A^+) \]  
\[ \Omega_P, \text{proc}(c, Q_{a'}) \]  
\[ \text{proc}(a, P_a), \Omega_H \]  
\[ H' = H[E := E, c \rightarrow_{sp} a] \]  
We can construct the typing derivation \( H' ; \Theta \models \Omega' \) as follows.

\[ \Psi_{Q} \models_{\text{q}} \Psi_{I} \models_{\text{q}} \Psi \models_{\text{q}} \Psi_{H} \]  
(2)
\[ \vdash_{h} \text{proc}(a, P_a), \Omega_H \]  
(16)
\[ \vdash_{h} \text{proc}(c, Q_{a'}) \]  
(17)
\[ a:\text{havoc}, \Psi'_{I} \]  
(19)
\[ \vdash_{q} \text{queue}(a', A^+, \cdot, a:A^+) \]  
(20)

\[ \vdash_{q} \text{queue}(a', A^+, \cdot, a:A^+) \]  
(21)

and

\[ \text{client}(\Gamma_{Q}^o), a':A^+ \]  
(22)

By weakening Lemma 6,7
\[ H[E := E, c \rightarrow_{sp} a]; \Psi_{H}, a':A^+, \Theta \models_{\text{h}} \Omega_H : \Psi^q_{H} \]  
(23)

By weakening Lemma 5 and (4)
\[ H[E := E, c \rightarrow_{sp} a]; \Psi_{H}, a':A^+, \Theta \models_{\text{h}} \Omega_H : \Psi^q_{H} \]  
(24)

\[ \vdash_{q} \text{queue}(a', A^+, \cdot, a:A^+) \]  
(25)
By inversion of (6)

\[ \Psi_c^i \vdash Q : c : B \]  

and

\[ \Psi_p, c : B = \psi_2 \]

By (11) and (9)

\[ \Psi_H^i, c : \text{havoc} = \text{dom} \Psi_1 \]

By one-rh rule,

\[ \Omega' = \Omega_Q, \Omega_P, \text{proc}(c, Q), \Omega_H \]

We can construct the typing derivation \( H; \Theta \upharpoonright \Omega' \) as follows.

(2) \[ \Psi_Q \vdash \Omega_Q : \Gamma_Q \]

and

\[ \Psi_p^i, \Psi_H^i \]

the union of the two gray contexts equals the following (10)

\[ \text{provider}(\Gamma_Q) \]

\[ \text{case one_rh.} \]

\[ \text{proc}(c, \text{wait } a' : Q) \odot \text{queue}(a' : 1, \text{end, }.) \odot \text{!}(c \in H) \]

By assumption

\[ \Omega = \Omega_Q, \text{queue}(a' : 1, \text{end, }.) \]

\[ \Omega_P, \text{proc}(c, \text{wait } a' : Q), \Omega_H \]

\[ H; \Theta \upharpoonright \Omega \]

By inversion of (1), and inversion Lemma 4, we have the following.

Each box is a typing derivation, read from top to bottom.

(2) \[ \Psi_Q^i \upharpoonright \Psi_Q \]

(3) \[ \Psi_Q^i \upharpoonright \Omega_Q \]

(4) \[ \text{split into four contexts in gray} \]

(5) \[ \Psi_p, \text{c : B} \]

(6) \[ \Psi_H^i \]

By inversion of (6)

\[ \Psi_c^i \vdash Q : c : B \]  

and

\[ \Psi_p, \text{c : B} = \psi_2 \]

\[ \psi_3 \]

the union of the two gray contexts equals the following (10)

\[ \text{provider}(\Gamma_Q) \]

\[ \text{case one_s.} \]

\[ \text{proc}(c, \text{close } c) \odot \text{queue}(c' : A_2, q : c : 1) \odot \text{!monitor}(\text{end } > 1) \]

By assumption

\[ \text{proc}(c, \text{close } c) \]

\[ \text{queue}(c' : A_2, q : c : 1) \odot \text{!monitor}(\text{end } > 1) \]
\( \Omega = \Omega_Q, \text{queue}(c' : A_2, q, c : 1), \Omega_P, \text{proc}(c, \text{close } c), \Omega_H \)
\( H; \Theta \models \Omega \)  
(1)

By inversion of (1), and inversion Lemma 4, we have two cases.
One is \( c \in H \) and the other is \( c \notin H \).

**Subcase** \( c \notin H \)

<table>
<thead>
<tr>
<th>(2)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Psi_Q )</td>
<td>( \Psi_\Omega )</td>
</tr>
<tr>
<td>( \vdash_q )</td>
<td>( \vdash_q )</td>
</tr>
<tr>
<td>( \Omega_Q )</td>
<td>( \Omega_Q )</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>( \Gamma_Q^o )</td>
<td>( \vdash c' : A_2, q, c : 1 )</td>
</tr>
</tbody>
</table>

and

<table>
<thead>
<tr>
<th>(11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdash c' : A_2 \leq c )</td>
</tr>
</tbody>
</table>

split into four contexts in gray (4)

<table>
<thead>
<tr>
<th>( \Psi_Q )</th>
<th>( \Psi_\Omega )</th>
<th>( \vdash )</th>
<th>( (\Psi_H)^* )</th>
<th>( \vdash \Theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \models u ), ( \Omega_P ), ( \vdash_h ), ( \Omega_H )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \models c, \text{close } c )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Psi_H^o )</td>
<td>( \models c : 1 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Psi_H^o )</td>
<td>( \models )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Psi_2 )</td>
<td>( \models )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \vdash )</td>
<td>( \vdash )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Psi_2 )</td>
<td>( \models )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \vdash )</td>
<td>( \vdash )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Psi_1 )</td>
<td>( \models )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \vdash )</td>
<td>( \vdash )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

By Lemma 9 and (3)

\( \Psi_2 \models c, \text{close } c \)  
(11)

By (8)

\( \Psi_2^o = \Psi_2 \setminus c : 1 \)  
(12)

By (10)

\( \text{provider}(\Gamma_Q^o) = \Psi_2 \setminus c : 1, \Psi_1 \)  
(13)

By \text{1.s rule}

\( \Omega' = \Omega_Q, \text{queue}(c' : A_2, q, \text{end }, \_), \Omega_P, \Omega_H \)

We can construct the typing derivation \( H; \Theta \models \Omega' \) as follows

<table>
<thead>
<tr>
<th>(1)</th>
<th>(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Psi_2^o )</td>
<td>( \Psi_2^o )</td>
</tr>
<tr>
<td>( \models )</td>
<td>( \models )</td>
</tr>
<tr>
<td>( \Omega_Q )</td>
<td>( \Omega_Q )</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>( \Gamma_Q^o )</td>
<td>( \vdash c' : A_2, q, \text{end }, _ )</td>
</tr>
</tbody>
</table>

and

By Lemma 9 and (3)

\( \Psi_2^o \models \text{queue}(c' : A_2, q, \text{end }, \_), c' : A_2 \leq_\_ \)

<table>
<thead>
<tr>
<th>(13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdash c' : A_2 \leq__ )</td>
</tr>
</tbody>
</table>

split into three contexts in gray (4)

We can construct the typing derivation \( H; \Theta \models \Omega' \) as follows

<table>
<thead>
<tr>
<th>( \models )</th>
<th>( \models )</th>
<th>( \models )</th>
<th>( \models )</th>
<th>( \models )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \models u ), ( \Omega_P ), ( \vdash_h ), ( \Omega_H )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \models c, \text{close } c )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Psi_H^o )</td>
<td>( \models c : 1 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Psi_H^o )</td>
<td>( \models )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Psi_2 )</td>
<td>( \models )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \vdash )</td>
<td>( \vdash )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Psi_1 )</td>
<td>( \models )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \vdash )</td>
<td>( \vdash )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

By inversion 

\( \text{havoc} \)

\( \text{proc}(c, \text{close } c), \Omega_H \)

We can construct the typing derivation \( H; \Theta \models \Omega' \) as follows
\[
\begin{array}{c}
\Psi_Q \vdash \Psi_p \quad \text{(13)}
\end{array}
\]

and

\[
\begin{array}{c}
\Psi_Q \vdash \Psi_p \\
\Omega_Q \\
\Gamma_Q \\
\end{array}
\]

\[
\begin{array}{c}
\Psi_p \vdash \Psi_H^* \\
\end{array}
\]

By inversion of (1), and inversion Lemma 4.

\[
\begin{array}{c}
\Psi_p \vdash \Psi_{\Psi_H^*} \\
\end{array}
\]

the union of the two gray contexts equals the following (10)

\[
\begin{array}{c}
\text{provider}(\Gamma_Q) \\
\end{array}
\]

\[
\begin{array}{c}
\text{client}(\Gamma_Q), c': A_2
\end{array}
\]

\[
\begin{array}{c}
\Psi_Q \vdash \Psi_p \\
\Omega_Q \\
\Gamma_Q \\
\end{array}
\]

\[
\begin{array}{c}
\Psi_p \vdash \Psi_{\Psi_H^*} \\
\end{array}
\]

By equality constraints from (8) and (10)

\[
\begin{array}{c}
\Psi_p, a : D \\
\Psi_2 \\
\Psi_1 \\
\end{array}
\]

the union of the two gray contexts equals the following (10)

\[
\begin{array}{c}
\text{provider}(\Gamma_Q), a : A \otimes B^+
\end{array}
\]

\[
\begin{array}{c}
\Psi_Q \vdash \Psi_p \\
\Omega_Q \\
\Gamma_Q \\
\end{array}
\]

\[
\begin{array}{c}
\Psi_p, c : 1 \\
\Psi_2 \\
\end{array}
\]

By inversion of (3) and Lemma 9.

\[
\begin{array}{c}
\text{client}(\Gamma_Q), c : A_1
\end{array}
\]

A₁ = 1, which contradicts with \(\neg(\text{end} \triangleright A₁)\).

\textbf{case one_sₐ₁.}

The proof is similar to \textbf{one_sₐₙ}. If we assume \(c \notin H\), the inversion of the process typing shows \(c = a\), which contradicts with the monitor’s check.

\textbf{case tensor_s₁ \(\otimes\) tensor_s₂}

\[
\begin{array}{c}
\text{queue}(a' : C^+, p, a : A^+ \otimes B^+) \otimes \text{proc}(a, \text{send} a (y \leftarrow P_y) ; Q) \\
\otimes (a \not\in H)
\end{array}
\]

\[
\begin{array}{c}
\text{\(\Rightarrow\)} \{ \exists b. \exists \beta. \text{queue}(a' : C^+, p \cdot b, a : B^+) \otimes \text{proc}(a, Q) \\
\otimes \text{queue}(b' : A^+, \cdot, b : A^+) \otimes \text{proc}(b, P) \otimes G(a \rightarrow_{s_p} b)
\end{array}
\]

By assumption

\[
\begin{array}{c}
\Omega = \Omega_Q, \text{queue}(a' : C^+, p, a : A^+ \otimes B^+), \\
\Omega_P, \text{proc}(a, \text{send} a (y \leftarrow P_y) ; Q), \Omega_H \\
\end{array}
\]

\[
\begin{array}{c}
H, \Theta \models \Omega
\end{array}
\]

By inversion of (1), and inversion Lemma 4.

\[
\begin{array}{c}
\Psi_Q \vdash \Psi_p \\
\Omega_Q \\
\Gamma_Q \\
\end{array}
\]

\[
\begin{array}{c}
\Psi_p, a : D \\
\Psi_2 \\
\Psi_1 \\
\end{array}
\]

By (8), (9), (10)

\[
\begin{array}{c}
\text{provider}(\Gamma_Q), a : A \otimes B^+
\end{array}
\]

\[
\begin{array}{c}
\Psi_Q \vdash \Psi_p \\
\Omega_Q \\
\Gamma_Q \\
\end{array}
\]

\[
\begin{array}{c}
\Psi_p, c : 1 \\
\Psi_2 \\
\end{array}
\]

By (8), (9), (10)

\[
\begin{array}{c}
\text{provider}(\Gamma_Q), c : A_1
\end{array}
\]

By (8) and \(b \) is fresh

\[
\begin{array}{c}
\Psi_p, a : B^+, b : A^+ = \Psi_2 \setminus (a : A \otimes B^+, a : B^+, b : A^+)
\end{array}
\]

By assumption \(c \notin H\), the inversion of the process typing shows \(c = a\), which contradicts with the monitor’s check.

\textbf{case one_sₐₙ.}

The proof is similar to \textbf{one_sₐₙ}. If we assume \(c \notin H\), the inversion of the process typing shows \(c = a\), which contradicts with the monitor’s check.
By (10), (17)  
\[ \text{provider}(\Gamma_Q^o), a;B^+; b;A^+ = \Psi_2 \setminus a;A^+ \otimes B^+; a;B^+; b;A^+; \Psi_1 \]  
(18)

By tensor $s_1$ + tensor $s_2$  
\[ \Omega' = \Omega_Q, \text{queue}(a';C^+; p; \cdot; b;A^+), \text{queue}(b';A^+; \cdot; \cdot; b;A^+), \Omega_P, \text{proc}(a, Q), \text{proc}(b, P_b), \Omega_H \]

We can construct the typing derivation $H; \Theta \vdash \Omega'$ as follows.

and

\begin{align*}
\text{client}(\Gamma_Q^o), a';C^+, b';A^+ \quad & \\
\text{split into three contexts in gray (4)}
\end{align*}

at (10) and $b$ is fresh  
\[ \text{fn}(send (a \rightarrow P_b); Q) \subset (\Psi_H)^*, \Theta, a, \text{ancestors}(H, a) \]

$H; (\Psi_H)^*; \Theta, a, \text{ancestors}(H, a)$  
\[ \text{fn}(Q) \subset (\Psi_H^o)^*, \Theta, a, \text{ancestors}(H, a) \]

By weakening lemma Lemma 6  
\[ H[E := E, a \rightarrow sp \ b]; (\Psi_H^o)^*, \Theta \vdash_h \Omega_H : \Psi_H^o \]

By queue typing rules,  
\[ \vdash_q \text{queue}(b';A^+; \cdot; b;A^+) : b;A^+ \rightleftharpoons b;A^+ \]

By inversion of (3) and Lemma 9  
\[ \Psi_{aq}, b';A^+ \vdash_q \text{queue}(a';C^+; p; \cdot; b'; a;B^+) \]

By tensor $sh_1 + tensor s_2$  
\[ \Omega' = \Omega_Q, \text{queue}(a';C^+; p; \cdot; b;A^+), \text{queue}(b';A^+; \cdot; \cdot; b;A^+), \Omega_P, \text{proc}(a, Q), \text{proc}(b, P_b), \Omega_H \]

$H' = H[E := E, a \rightarrow sp \ b]$  

We can construct the typing derivation $H'; \Theta \vdash \Omega'$ as follows.
case tensor \_sh1 + alarm\_p

\[ \text{queue}(a':C, p, a:B) \otimes \text{proc}(a, \text{send} \ c \ (y \leftarrow P_y); \ Q) \]

\[ = \ \exists b, b', \cdots, \otimes ((\text{monitor} \ (\text{ch}(b') \rightarrow B)) \rightarrow \{\text{alarm}(a)\}) \]

By assumption

\[ \Omega = \Omega_Q, \text{queue}(a':A^+ \otimes B^+, b, q, a:C^+); \ q, a:C^+ \]

\[ \Omega_P, \text{proc}(c, y \leftarrow \text{recv} \ a'; R_y), \Omega_H \]

\[ H; \Theta \models \Omega \]

(1)

By inversion of (1), and inversion Lemma 4, we have two cases. One is \( c \in H \) and the other is \( c \notin H \).

Subcase \( c \notin H \)

\[ \text{split into four contexts in gray (4)} \]

By (2) and (3) and (12)

\[ \exists q, \ \Omega_Q, \text{queue}(a':A^+ \otimes B^+, b, q, a:C^+) \]

\[ \Gamma_0^\omega \models \ a':A^+ \otimes B^+ \leftarrow a':C^+ \]

and

\[ \text{split into four contexts in gray (4)} \]

By equality constraints from (8) and (10)

\[ D = B \]

By inversion of (6)

\[ B = D = D_1^+ \otimes D_2^+, \text{ which contradicts with } \neg ((\text{ch}(b') \rightarrow B) \]

By assumption

\[ \Omega' = \Omega_Q, \text{queue}(a':B^+, b, q, a:C^+), \Omega_P, \text{proc}(c, R_y), \Omega_H \]

We can construct the typing derivation \( H; \Theta \models \Omega' \) as follows

\[ \text{split into four contexts in gray (4)} \]

By (2) and (11) and (13)

\[ \Gamma_0^\omega \models \ a':B^+ \leftarrow a':C^+ \]
We can construct the typing derivation $H; \Theta \models \Omega'$ as follows.

### Subcase $c \in H$

#### Case lolli_s1 + lolli_s2

The typing derivation for this case is shown in the diagram below.

- **Case lolli_s1**
  - Type $(a': A^+ \otimes B^+, q, a:C^+)$
  - Typing rule $\Gamma \vdash \psi_a$ for $a': A^+$
  - Typing rule $\psi_q$ for $q$
  - Typing rule $\psi_a$ for $a:C^+$

- **Case lolli_s2**
  - Type $(b': A^+, \psi_b', a': B^+)$
  - Typing rule $\Gamma \vdash \psi_b'$ for $b': A^+$
  - Typing rule $\psi_a$ for $a': B^+$

By inversion lemma on (3) and weakening lemma Lemma 7, we have

\[ \Gamma \vdash \psi_a \]

By applying inversion lemma Lemma 4 on (6), we get

\[ \text{fn}(y \leftarrow \text{recv } a'; R_y) \subseteq (\psi_{\Omega_H})^* \]

By (11) and (4), we have

\[ \psi_{\Omega_H} = \psi_{\Omega}^* \]

By (11), (10), and (4), we have

\[ \text{fn}(R_y) \subseteq (\psi_{\Omega_H})^* \]

By (13) and havoc process typing rules, we get

\[ H; (\psi_{\Omega_H}^*; a': B^+) \vdash b' \leftarrow \text{proc}(c, R_y) \]

By weakening lemma Lemma 7 and (11), we have

\[ H; (\psi_{\Omega_H}^*; b') \vdash b' \leftarrow \text{proc}(c, R_y) \]

By tensor rule, we get

\[ \Omega' = \Omega_0, \text{queue}(a': B^+, q, a:C^+), \Omega_0, \text{proc}(c, R_y), \Omega_H \]

By inversion of (6) and (4)
\[ \Psi_1 \vdash P_y : y : A^+ \quad \text{and} \quad \Psi_2 \vdash \text{proc}(c, \text{send} \ a' (y \leftarrow P_y) ; Q) \]
\[ : c : D \]
\[ \Omega_P \]
and

By assumption
\[ \ominus \exists b. \exists \bar{b}'. \text{queue}(b' : A^+, b' : p, a : C^-) \odot \text{proc}(c, Q) \]
\[ \odot \text{proc}(b, P_y) \]
\[ \ominus \exists \bar{b}''. \text{proc}(b, P_y) \]
\[ : c : D, b : A^+ \]
\[ \Omega_H \]
\[ \ominus \exists \bar{b}''. \text{proc}(b, P_y) \]
\[ : c : D, b : A^+ \]
\[ \Omega_H \]

By inversion of (1), and inversion Lemma 4.

We can construct the typing derivation \( H ; \Theta \vdash \Omega' \) as follows.

By (4) and \( b \) is fresh
\[ \text{client}(\Gamma_Q), a' : A^+ \rightarrow B^- \]

split into three contexts in gray (16)

By inversion lemma Lemma 4 on (6).

By applying inversion lemma Lemma 4 on (6).

By (10) and \( b \) is fresh

By (10) and \( b \) is fresh
We can construct the typing derivation

By assumption

By inversion of (6) and (11)

By inversion of (6) and (11)

By assumption

We assume $c \notin H$ and derive a contradiction.

By inversion of (1), and inversion Lemma 4,
We can construct the typing derivation

By inversion of (6)

By inversion lemma on (3)

By substitution and (13) and

By (4) and (12)

By (11)

By weakening lemma Lemma 7 and (11)

By applying inversion lemma Lemma 4 on (6)

By (13) and havoced process typing rules

By weakening lemma 7 and (11)

By lolli_r rule

We can construct the typing derivation $H; \Theta \models \Omega'$ as follows

and

We can construct the typing derivation $H; \Theta \models \Omega'$ as follows

Subcase $c \in H$
By assumption
\[ \Omega = \Omega_Q, \text{queue}(a'\triangleright B^+, \rightarrow, p, a:C^-) \otimes \text{proc}(c, \text{send } a' \text{ shift } ; Q) \]

By inversion of (1), and inversion Lemma 4, we have two cases.
One is \( c \in H \) and the other is \( c \notin H \).

**Subcase** \( c \notin H \)

\[
\begin{align*}
\Psi_Q^{\uparrow} & \vdash \Psi_H^{\uparrow} & (1) \\
\vdash_u & \text{queue}(a'\triangleright B^+, \rightarrow, p, a:C^-) \otimes \text{proc}(c, \text{send } a' \text{ shift } ; Q) \\
\Omega_Q & : \\
\Gamma_Q^{\uparrow} & \vdash \text{queue}(a'\triangleright B^+, \rightarrow, p, a:C^-) \\
\end{align*}
\]

**Subcase** \( c \in H \)

By case up_s
\[ \text{queue}(a'\triangleright B^+, \rightarrow, p, a:C^-) \otimes \text{proc}(c, \text{send } a' \text{ shift } ; Q) \]

By inversion of (1), and inversion Lemma 4, we have two cases.
One is \( c \in H \) and the other is \( c \notin H \).

**Subcase** \( c \notin H \)

\[
\begin{align*}
\Psi_Q^{\uparrow} & \vdash \Psi_H^{\uparrow} & (1) \\
\vdash_u & \text{queue}(a'\triangleright B^+, \rightarrow, p, a:C^-) \otimes \text{proc}(c, \text{send } a' \text{ shift } ; Q) \\
\Omega_Q & : \\
\Gamma_Q^{\uparrow} & \vdash \text{queue}(a'\triangleright B^+, \rightarrow, p, a:C^-) \\
\end{align*}
\]

"split into three contexts in gray (by (4) and (10))"

\[
\begin{align*}
\Psi_Q^{\uparrow} & \vdash \Psi_H^{\uparrow} & (1) \\
\vdash_u & \text{queue}(a'\triangleright B^+, \rightarrow, p, a:C^-) \otimes \text{proc}(c, \text{send } a' \text{ shift } ; Q) \\
\Omega_Q & : \\
\Gamma_Q^{\uparrow} & \vdash \text{queue}(a'\triangleright B^+, \rightarrow, p, a:C^-) \\
\end{align*}
\]

By Lemma 9 and (3)

\[
\begin{align*}
\Psi_{c,H}^{\uparrow} & \vdash \Psi_{c,H}^{\uparrow} & (1) \\
\vdash_u & \text{queue}(a'\triangleright B^+, \rightarrow, p, a:C^-) \otimes \text{proc}(c, \text{send } a' \text{ shift } ; Q) \\
\Omega_Q & : \\
\Gamma_Q^{\uparrow} & \vdash \text{queue}(a'\triangleright B^+, \rightarrow, p, a:C^-) \\
\end{align*}
\]

By c is in a provider channel of \( \Gamma_Q^{\uparrow} \) and \( a' \) is a client channel of \( \Gamma_Q^{\uparrow} \).

By inversion of (6)
\[ \Psi_{c,H}^{\uparrow}, a'\triangleright B^+ \vdash Q::c:D \]
and \[ \Psi_{c,H}^{\uparrow} = \Psi_{c,H}^{\uparrow}, a'\triangleright B^+ \]

By Lemma 9 and (3)
By applying inversion lemma Lemma 4 on (6)  
\[
\text{fn(send } a \text{ shift } ; Q : c) \subseteq (\Psi _{1})^{+}, \Theta, c, \text{ ancestors}(H, c)
\]
\[
H; (\Psi _{1})^{+}, \Theta \models_{h} \Omega _{H} : \Psi _{H}^{a}\]
and \(\Psi _{H}^{a} = \Psi _{H}^{1}, c : \text{ havoc}
\]  
By (10) and havocd process typing rules  
\[
H; (\Psi _{H}^{a}, a' : B^{+})^{+}, \Theta \models_{h} \text{proc}(c, Q) : c : \text{ havoc}
\]
By Lemma 9 and (3)  
\[
\text{c / } \Psi _{H}^{a} = \Psi _{H}^{1}, a' : B^{+} \subseteq a : C^{-}
\]
By Lemma 9 and (3)  
\[
\text{c / } \Psi _{H}^{a} = \Psi _{H}^{1}, a' : B^{+} \subseteq a : C^{-}
\]
We can construct the typing derivation \(H; \Theta \models \Omega '\) as follows.

\[
\begin{align*}
\Psi _{Q} = & \quad \begin{cases}
\Psi _{1}' \quad & \text{client}(\Gamma _{Q}^{a}), a' : B \\
\Psi _{1} \quad & \text{split into four contexts in gray (4)}
\end{cases} \\
\Omega _{Q} = & \quad \begin{cases}
\Omega _{1} \quad & \text{provider}(\Gamma _{Q}^{a}), a : C \\
\Omega _{1} \quad & \text{the union of the two gray contexts}
\end{cases} \\
\Gamma _{Q}^{a} = & \quad \begin{cases}
\Gamma _{1} \quad & \text{split into four contexts in gray (4)}
\end{cases}
\end{align*}
\]

By c is in a provider channel of \(\Gamma _{Q}^{a}\) and \(a'\) is a client channel of \(\Gamma _{Q}^{a}\)  
\[
c \neq a'
\]
By inversion of (6)  
\[
\Psi _{1}' = \Psi _{1}^{+}, a' : B^{+} \subseteq a : C^{-}
\]
\text{By assumption}
\[
\Omega = \Omega _{Q}, \text{queue}(a' : B^{+}, \text{ shift }) : p, a : C^{-}
\]
\text{By assumption}
\[
\text{H}, \Theta \models \Omega
\]
We assume that \(c \notin H\) and derive a contradiction.

\[
\begin{align*}
\Psi _{Q} = & \quad \begin{cases}
\Psi _{1}' \quad & \text{case alarm } c \text{ (when the message is shift)}
\end{cases} \\
\Omega _{Q} = & \quad \begin{cases}
\Omega _{1} \quad & \text{queue}(a' : B^{+}, p, a : C) \\
\Omega _{1} \quad & \text{assert}(\text{alarm } c)
\end{cases}
\end{align*}
\]

By assumption
\[
\text{H}, \Theta \models \Omega
\]

\[
\begin{align*}
\Psi _{Q} = & \quad \begin{cases}
\Psi _{1}' \quad & \text{case alarm } c \text{ (when the message is shift)}
\end{cases} \\
\Omega _{Q} = & \quad \begin{cases}
\Omega _{1} \quad & \text{queue}(a' : B^{+}, p, a : C) \\
\Omega _{1} \quad & \text{assert}(\text{alarm } c)
\end{cases}
\end{align*}
\]

By assumption
\[
\text{H}, \Theta \models \Omega
\]

By assumption
\[
\text{H}, \Theta \models \Omega
\]

By assumption
\[
\text{H}, \Theta \models \Omega
\]
By (8) and (10)
\[ D = \uparrow B^+ \]  
(11)

By queue typing rules
\[ \vdash \text{queue}(c':B^+, \cdot, c:B^+) \]  
(12)

By inversion of (6)
\[ \Psi_c, \uparrow \text{proc}(c, R) :: c:B^+ \]  
(13)

By \text{up}_r \text{ rule}
\[ \Omega' = \Omega_Q, \text{queue}(c':B^+, \cdot, c:B^+), \Omega_P, \text{proc}(c, R), \Omega_H \]

We can construct the typing derivation \( H; \Theta \models \Omega' \) as follows

\[ \begin{array}{c|c|c|c|c|c}
\hline
\text{step} & \Psi_Q & \Psi_I & \Psi_c & (\Psi_H)^* & \Theta \\
\hline
(2) & \vdash & \vdash & \vdash & \vdash & \Theta \\
\hline
(3) & \Omega _Q & \vdash & \vdash & \vdash & \Theta \\
\hline
(4) & \text{queue}(c':B^+, \cdot, c:B^+) & \vdash & \vdash & \vdash & \Theta \\
\hline
(5) & \vdash & \vdash & \vdash & \vdash & \Theta \\
\hline
(6) & \vdash & \vdash & \vdash & \vdash & \Theta \\
\hline
\end{array} \]

\[ \Omega' = \Omega_Q, \text{queue}(c':B^+, \cdot, c:B^+), \Omega_P, \text{proc}(c, R), \Omega_H \]

The union of the two gray contexts equals the following (9)

\[ \text{provider}(\Gamma_Q^{\circ}), c: B^+ \]

**Subcase** \( c \in H \)

\[ \begin{array}{c|c|c|c|c|c}
\hline
\text{step} & \Psi_Q & \vdash & \vdash & \vdash & \Theta \\
\hline
(2) & \vdash & \vdash & \vdash & \vdash & \Theta \\
\hline
(3) & \Omega _Q & \vdash & \vdash & \vdash & \Theta \\
\hline
(4) & \text{queue}(c':B^+, \cdot, c:B^+) & \vdash & \vdash & \vdash & \Theta \\
\hline
(5) & \vdash & \vdash & \vdash & \vdash & \Theta \\
\hline
\end{array} \]

\[ \text{the union of the two gray contexts equals the following (10)} \]

\[ \text{provider}(\Gamma_Q^{\circ}), c: B^+ \]

By queue typing rules
\[ \vdash \text{queue}(c':B^+, \cdot, c:B^+) \]  
(10)

By applying inversion lemma \( \text{Lemma 4} \) on (6)
\[ \text{fn(shift}_{\rightarrow} \text{recv } c ; R) \subseteq (\Psi_H)^*, \Theta, c, \text{ancestors}(H, c) \]

By (11)
\[ \text{H; (\Psi_H)^*, \Theta \models h; \Omega_H ; \Psi_H^{\circ} and \Psi_H^{\circ} = \Psi_H^{\circ}, c : \text{havoc} \]  
(11)

By (12) and havoc process typing rules
\[ \text{H; (\Psi_H)^*, \Theta \models h; \text{proc}(c, R) ; c : \text{havoc} \]  
(12)

By \text{up}_r \text{ rule}
\[ \Omega' = \Omega_Q, \text{queue}(c':B^+, \cdot, c:B^+), \Omega_P, \text{proc}(c, R), \Omega_H \]

We can construct the typing derivation \( H; \Theta \models \Omega' \) as follows

\[ \begin{array}{c|c|c|c|c|c}
\hline
\text{step} & \Psi_Q & \vdash & \vdash & \vdash & \Theta \\
\hline
(2) & \vdash & \vdash & \vdash & \Theta \\
\hline
(3) & \Omega _Q & \vdash & \vdash & \Theta \\
\hline
(4) & \text{queue}(c':B^+, \cdot, c:B^+) & \vdash & \vdash & \Theta \\
\hline
(5) & \vdash & \vdash & \vdash & \Theta \\
\hline
\end{array} \]

\[ \text{the union of the two gray contexts equals the following (9)} \]

\[ \text{provider}(\Gamma_Q^{\circ}), c: B^+ \]

**Case down_s**

\[ \text{queue}(a':C^+, p \cdot a: B^-) \otimes \text{proc}(a, \text{send a shift } ; Q) \]

\[ \otimes \text{(monitor(shift }_{\rightarrow} \downarrow B^-) \]

\[ \rightarrow \{ \text{queue}(a':C^+, p \cdot \text{shift, a: B^-) \otimes \text{proc}(a, Q)} \}


By assumption
\[ \Omega = \Omega_Q, \text{queue}(a':C^+, \ p, a\downarrow B^-), \]
\[ \Omega_P, \text{proc}(a, \text{send } a \text{ shift}; Q), \Omega_H \]
\[ H; \Theta \models \Omega \]
\[ \text{By inversion of (1), and inversion Lemma 4, we have two cases.} \]
\[ \text{One is } c \in H \text{ and the other is } c \notin H. \]

**Subcase** \( a \notin H \)

By (2), \( \vdash_a \)
\[ \Omega_Q \]
\[ \text{queue}(a':C^+, \ p, a\downarrow B^-) \]
\[ \Gamma_Q^\text{o} \]
\[ a':C^+ \ll a\downarrow B^- \]

split into four contexts in gray (4)

\[ \vdash \Psi_Q \]
\[ \text{queue}(a':C^+, \ p, a\downarrow B^-) \]
\[ \Gamma_Q^\text{o} \]
\[ a':C^+ \ll a\downarrow B^- \]

By equality constraints from (8) and (10)
\[ D = \downarrow B^- \]
\[ \text{By inversion of (6)} \]
\[ \vdash_a \]
\[ \Omega_Q \]
\[ \text{queue}(a':C^+, \ p, a\downarrow B^-) \]
\[ \Gamma_Q^\text{o} \]
\[ a':C^+ \ll a\downarrow B^- \]

The union of the two gray contexts

**Subcase** \( a \in H \)

By (2), \( \vdash_a \)
\[ \Omega_Q \]
\[ \text{queue}(a':C^+, \ p, a\downarrow B^-) \]
\[ \Gamma_Q^\text{o} \]
\[ a':C^+ \ll a\downarrow B^- \]

split into three contexts in gray (4)

\[ \vdash \Psi_Q \]
\[ \text{queue}(a':C^+, \ p, a\downarrow B^-) \]
\[ \Gamma_Q^\text{o} \]
\[ a':C^+ \ll a\downarrow B^- \]

By applying inversion Lemma 4 on (6)
\[ \text{fn(send a shift; Q)} \subseteq (\Psi_H^\text{h})^\ast, \Theta, a, \text{ancestors}(H, a) \]
\[ H; (\Psi_H^\text{h})^\ast, \Theta \models \_ h, \Omega_H : \Psi_H^\text{h} \]
\[ \text{and } \Psi_H^\text{h} = \Psi_H^\text{h}, a : \text{havoc} \]
\[ \text{By (11) and havoced process typing rules} \]
\[ H; (\Psi_H^\text{h})^\ast, \Theta \models \_ h, \text{proc}(a, Q) : a : \text{havoc} \]

By (8) and Lemma 9
\[ \vdash \Psi_Q \]
\[ \Omega_Q \]
\[ \text{queue}(a':C^+, \ p, a\downarrow B^-) \]
\[ \Gamma_Q^\text{o} \]
\[ a':C^+ \ll a\downarrow B^- \]

We can construct the typing derivation \( H; \Theta \models \Omega' \) as follows.
By equality constraints from (8) and (10)

\[ D = B \]

By inversion of (6)

\[ B = D = \downarrow D' \], which contradicts with \( \neg (\text{shift} \triangleright \downarrow D' \)
We can construct the typing derivation \( H; \Theta \models \Omega' \) as follows

\[
\begin{array}{c}
\text{(2)} \\
\Psi_Q^1 \vdash \\
\int_{q} \\
\Omega_Q \\
\Gamma_Q^0 \\
\vdash a':B^- \leftarrow a:B^-
\end{array}
\]

and

\[
\begin{array}{c}
\text{(11)} \\
\Psi_Q^1 \vdash \\
\int_{q} \\
\Omega_Q \\
\Gamma_Q^0 \\
\vdash a':B^- \leftarrow a:B^-
\end{array}
\]

Subcase \( c \in H \)

By inversion, \( B^- = C^- \)

\[
\begin{array}{c}
\text{(2)} \\
\Psi_Q^1 \vdash \\
\int_{q} \\
\Omega_Q \\
\Gamma_Q^0 \\
\vdash \text{queue}(a':B^-, \text{shift}, a:B^-)
\end{array}
\]

and

\[
\begin{array}{c}
\text{provider}(\Gamma_Q^0), a:B^-
\end{array}
\]

Case with \( s \)

\[
\begin{array}{c}
\text{proc}(c, a'.lab_j ; Q) \otimes \text{queue}(a': \& (\text{lab}_j : A^+_j), q, a:B^-) \otimes !\text{monitor}(a', \text{lab}_j \& (\text{lab}_i : A^+_i)) \\
\rightarrow \{ \text{proc}(c, Q) \otimes \text{queue}(a':A^+_j, \text{lab}_j \cdot q, a:B^-) \}
\end{array}
\]

By assumption

\[
\Omega = \Omega_Q, \text{queue}(a': \& (\text{lab}_j : A^+_j), q, a:B^-), \\
\Omega_P, \text{proc}(c, a'.lab_j ; Q), \Omega_H \\
H; \Theta \models \Omega
\]

By inversion of (1), and inversion Lemma 4, we have two cases.

One is \( c \in H \) and the other is \( c \notin H \).

Subcase \( c \notin H \)

\[
\begin{array}{c}
\text{(2)} \\
\Psi_Q^1 \vdash \\
\int_{q} \\
\Omega_Q \\
\Gamma_Q^0 \\
\vdash a':\& (\text{lab}_i : A^+_i), \leftarrow a:B^-
\end{array}
\]

By queue typing

\[
\cdot \vdash \text{queue}(a':B^- \rightarrow q, a:B^-)
\]

By applying inversion lemma Lemma 4 on (6)

\[
\text{fn}(\text{shift} \rightarrow \text{recv } a'; R) \subseteq (\Psi_H)^+, \Theta, c, \text{ ancestors}(H, c)
\]

\[
H; (\Psi_H)^+, \Theta \models_h \Omega_H : \Psi_H^0
\]

and \( \Psi_H^0 = \Psi_H^0, c : \text{havoc} \)

By (11) and (4)

\[
\Psi_H^0 = \Psi_H^0, a':\& B^-
\]

By (13) and havoced process typing rules

\[
H; (\Psi_H^0, a':B^-)^+, \Theta \models_h \text{proc}(c, R) : c:\text{havoc}
\]

By (4) and (12)

\[
\text{client}(\Gamma_Q^0), a':B^- = \Psi_Q^0, \Psi_Q^1, \Psi_H^0, a':B^-
\]

By down rule

\[
\Omega' = \Omega_Q, \text{queue}(a':B^- \rightarrow q, a:B^-), \Omega_P, \text{proc}(c, R), \Omega_H
\]
By inversion of (6) and (4)
\(\Psi_i^t, a'^i:A_j^i \vdash c.D\)
and \(\Psi_i^t = \Psi_i^t, a' : \{lab_b:A_i^t\}_i\)
(11)

By (11)
\(\Psi_i^t, a'^i:A_j^i \vdash c.D\)
(12)

By Lemma 9
\(\Psi_a^i \sqsubseteq \text{queue}(a'^i:A_j^i, lab_b \cdot p, a:B^-) : a'^i: A_j^i \leftarrow a:B^-\)
(13)

By (4)
client\((\Gamma_Q^\alpha), a'^i:A_j^i\) = \(\Psi_Q^i, \Psi_a^i, \Psi_c^i, a'^i:A_j^i, \Psi_H^i\)
(14)

By with_s rule
\(\Omega' = \Omega_Q, \text{queue}(a'^i:A_j^i, lab_b \cdot q, a:B^-),\)
\(\Omega_P, \text{proc}(c, Q), \Omega_H\)

We can construct the typing derivation \(H; \Theta = \Omega'\) as follows.

By applying inversion lemma Lemma 4 on (6)
\(\text{fn}(a'^i, lab_c \cdot Q) \subseteq (\Psi_i^t)^*, \Theta, c, \text{ancestors}(H, c)\)
(10)

\(H; (\Psi_i^t)^*, \Theta \vdash h, \Omega_H : \Psi_H^i\)
and \(\Psi_H^i = \Psi_H^i, c: \text{havoc}\)
(11)

By (10) and (4)
\(\text{fn}(Q) \subseteq (\Psi_i^t)^*, \Theta, c, \text{ancestors}(H, c)\)
and \(\Psi_H^i = \Psi_H^i, a' : \{lab_b:A_i^t\}_i\)
(12)

By (12) and havoced process typing rules
\(H; (\Psi_H^i)^*, \Theta \vdash h, \text{proc}(c, Q) : c: \text{havoc}\)
(13)

By Lemma 9
\(\Psi_a^i \sqsubseteq \text{queue}(a'^i:A_j^i, lab_b \cdot p, a:B^-) : a'^i: A_j^i \leftarrow a:B^-\)
(14)

By with_s rule
\(\Omega' = \Omega_Q, \text{queue}(a'^i:A_j^i, lab_b \cdot q, a:B^-),\)
\(\Omega_P, \text{proc}(c, Q), \Omega_H\)

We can construct the typing derivation \(H; \Theta = \Omega'\) as follows.
\( 
\)
By applying inversion lemma Lemma 4 on (6)

\[ \text{fn} \{ \text{case } c \{ \text{lab}_i \to P_i \}_1 \subseteq (\Psi_i^1)^*, \Theta, c, \text{ancestors}(H, c) \} \]

\[ H; (\Psi_i^1)^*, \Theta \models_a \Omega_H : \Psi_i^0, \] and

\[ \Psi_i^0 = \Psi_i \] (11)

By (11)

\[ \text{fn}(P_i) \subseteq (\Psi_i^1)^*, \Theta, c, \text{ancestors}(H, c) \] (12)

By (12) and havoced process typing rules

\[ H; (\Psi_i^1)^*, \Theta \models_a \text{proc}(c, P_i) :: \text{chavoc} \] (13)

By (8)

\[ \Psi_i^0 = \text{dom } \Psi_i \backslash c: A_i^+ \] (14)

By (9)

\[ \text{provider}(\Gamma_Q^0), c: A_i^- = \Psi_2, \Psi_1 \backslash c: \{ \text{lab}_i : A_i^- \}_1, c: A_i^- \] (15)

By with rule

\[ \Omega' = \Omega_Q, \text{queue}(c':B^-, q, c: A_j^-), \Omega_P, \text{proc}(c, P_i), \Omega_H \]
We can construct the typing derivation $H; \Theta \vdash \Omega'$. By applying inversion lemma Lemma 4 on (6) in H, we have

By (10) and (4)

and

Subcase $c \in H$
case plus s_a

\[\{\text{queue}(c': B, q, c: A) \otimes \text{queue}(c': B, q, c: A)\}\]

\[\text{split into four contexts in gray (4)}\]

\[\begin{align*}
\text{split into three contexts in gray (4)} & \\
\Psi_Q^i \supseteq \Psi_P^j \supseteq \Psi_c \\
\text{By assumption} & \\
\Omega & = \Omega_Q, \text{queue}(c': B, q, c: A), \\
\Omega_{P}, \text{proc}(c, c, \text{labj} : Q), \Omega_H & \\
H; \Theta & \models \Omega
\end{align*}\]

We assume \(c \notin H\) and derive a contradiction.

By inversion of (1), and inversion Lemma 4.

\[\begin{align*}
\text{(2)} & \\
\Psi_Q^i \supseteq \Psi_c \\
\models \varphi_q & \\
\varphi_q & \models \varphi_q \\
\text{queue}(c': B, q, c: A) & \\
\Gamma_Q^i & \models c': B \leftarrow c: A
\end{align*}\]

\[\begin{align*}
\text{client}(\Gamma_Q^i), c': B & \\
\text{split into four contexts in gray (4)} & \\
\begin{array}{c}
\Psi_Q^i \supseteq \\
\models \varphi_q \\
\text{By assumption} & \\
\Omega & = \Omega_Q, \text{queue}(c': B, q, c: A), \\
\Omega_{P}, \text{proc}(c, c, \text{labj} : Q), \Omega_H & \\
H; \Theta & \models \Omega
\end{array}
\end{align*}\]

\[\begin{align*}
\text{case plus r} & \\
\text{queue}(a': \oplus \{\text{lab}_i : A_i^+\}, q, a: B^+) & \\
\otimes \text{proc}(c, \text{case } a' \{\text{lab}_i \rightarrow P_i\}) & \\
\text{split into the two gray contexts} & \\
\text{equals the following (16)} & \\
\text{provider}(\Gamma_Q^i), c: A
\end{align*}\]

\[\begin{align*}
\text{client}(\Gamma_Q^i), c': B & \\
\text{split into four contexts in gray (4)} & \\
\begin{array}{c}
\Psi_Q^i \supseteq \\
\models \varphi_q \\
\text{By assumption} & \\
\Omega & = \Omega_Q, \text{queue}(c': B, q, c: A), \\
\Omega_{P}, \text{proc}(c, c, \text{labj} : Q), \Omega_H & \\
H; \Theta & \models \Omega
\end{array}
\end{align*}\]

\[\begin{align*}
\text{case plus s_op} & \\
\text{The proof is similar to the previous cases.}
\end{align*}\]
split into four contexts in gray (13)

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \Psi_{\omega} & \Psi_{\omega} & (\Psi_H)^* & \Theta \\
\hline
\Psi_{\omega} & \vdash u & \Omega_P & : & \vdash_h & \Omega_H \\
\Psi_{\omega} & : & \vdash & : & \Psi_{\omega} & (7) \\
\hline
\end{array}
\]

and

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \Psi_{\omega} & \Psi'_{\omega} & c : D & \omega \\
\hline
\Psi'_{\omega} & \vdash & \vdash_d & \vdash & : & \Psi_1 \\
\Psi'_{\omega} & : & \vdash & : & \Psi_2 & (8) \\
\hline
\end{array}
\]

the union of the two gray contexts equals the following (9)

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \omega & \omega & \omega & \omega \\
\hline
\text{provider}(\Gamma_Q^\omega), a' : A^+_j \\
\hline
\end{array}
\]

Subcase \( c \in H \)

(2)

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \Psi_{\omega} & \Psi_{\omega} & \Psi'_{\omega} & \omega \\
\hline
\omega & \vdash & \vdash_d & \vdash & \vdash_d & \vdash \\
\Omega_Q & : & \vdash & : & \Psi_1 & (9) \\
\hline
\end{array}
\]

and

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \omega & \omega & \omega & \omega \\
\hline
\text{client}(\Gamma_Q^\omega), a' : \{lab_i : A^+_j\}_i \\
\hline
\end{array}
\]

split into three contexts in gray (4)

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \Psi_{\omega} & \Psi_{\omega} & (\Psi_H)^* & \Theta \\
\hline
\Psi_{\omega} & \vdash u & \Omega_P & : & \vdash_h & \Omega_H \\
\Psi_{\omega} & : & \vdash & : & \Psi_{\omega} & (5) \\
\hline
\end{array}
\]

and

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \Psi_{\omega} & \Psi_{\omega} & \omega \\
\hline
\Psi_{\omega} & \vdash & \vdash_d & \vdash_d & \vdash \\
\Psi'_{\omega} & : & \vdash & : & \Psi_1 & (7) \\
\hline
\end{array}
\]

the union of the two gray contexts equals the following (9)

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \omega & \omega & \omega & \omega \\
\hline
\text{provider}(\Gamma_Q^\omega), a' : B^+ \\
\hline
\end{array}
\]

By inversion on (3)

\[
\Psi_{\omega} \vdash \text{queue}(a' : A^+_j, q, a : B^+) 
\]

By applying inversion lemma Lemma 4 on (6)

\[
\text{fn}(\text{case } a' \{\text{lab}_i \rightarrow P_i\}, i) \subseteq (\Psi_H)^*, \Theta, c, \text{ ancestors}(H, c) \\
H : (\Psi_H)^*, \Theta \vdash_h \Omega_H : \Psi_H^c \\
\text{and } \Psi_{\omega}^c = \Psi_{\omega}^c, c : \text{ havoc} 
\]

By (11)

\[
\text{fn}(P_i) \subseteq (\Psi_H)^*, \Theta, c, \text{ ancestors}(H, c) \\
\text{By (12) and havoced process typing rules} \\
H : (\Psi_H)^* \oplus \{\text{lab}_i : A^+_j\}_i, a : A^+_j)^*, \Theta \\
\vdash_h \text{proc}(c, P_i) : c : \text{ havoc} 
\]

By (4)

\[
\text{client}(\Gamma_Q^\omega), a' : A^+_j = \Psi_H^c \setminus a' : \{\text{lab}_i : A^+_j\}_i, a : A^+_j 
\]

By plus, \( \epsilon \) rule

\[
\Omega' = \Omega_Q, \text{queue}(a' : A^+_j, q, a : B^+), \Omega_P, \text{proc}(c, P_i), \Omega_H \\
\text{We can construct the typing derivation } H ; \Theta = \Omega' \text{ as follows} 
\]

and

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \omega & \omega & \omega & \omega \\
\hline
\text{client}(\Gamma_Q^\omega), a' : A^+_j \\
\hline
\end{array}
\]

split into three contexts in gray (14)

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \Psi_{\omega} & \Psi_{\omega} & (\Psi_H)^* & \Theta \\
\hline
\Psi_{\omega} & \vdash u & \Omega_P & : & \vdash_h & \Omega_H \\
\Psi_{\omega} & : & \vdash & : & \Psi_{\omega} & (5) \\
\hline
\end{array}
\]

and

\[
\begin{array}{c|c|c|c|c|c}
\hline
& \omega & \omega & \omega & \omega & \omega \\
\hline
\text{provider}(\Gamma_Q^\omega), a' : B^+ \\
\hline
\end{array}
\]
id : queue(c′:A₁, p, c:A₂) ⊗ proc(c, c ← d′) ⊗ queue(d′:B₁, q, d:B₂) ⊗ ! (monitor(A₂ = B₁)) ⊗ ![c /∈ H]
→ { queue(c′:A₁, p, q, d:B₂) }

idₜ : queue(c′:A₁, p, c:A₂) ⊗ proc(c, c ← d′) ⊗ queue(d′:B₁, q, d:B₂) ⊗ ! (monitor(A₂ = B₁)) ⊗ ![c /∈ H]
→ { queue(c′:A₁, p, q, d:B₂) | ![dangling(d′)] }

idₜₜ : queue(c′:A₁, p, c:A₂) ⊗ proc(c, c ← d′) ⊗ queue(d′:B₁, q, d:B₂) ⊗ !(monitor(A₂ ≠ B₁))
→ { alarm(c) }

idₜₜₜ : queue(c′:A₁, p, c:A₂) ⊗ proc(c, c ← d) ⊗ queue(d′:B₁, q, d:B₂) → { alarm(c) }

idₜₜₜₜ : queue(c′:A₁, p, c:A₂) ⊗ proc(c, a ← d′) ⊗ !(monitor(a ≠ c)) → { alarm(c) }

cut⁺ : proc(c, x:A⁺ ← Pₛ ; Qₛ) ⊗ ![c /∈ H]
→ { (∃ a′. proc(c, Qₛ a′) ⊗ queue(a′:A⁺, b, a:A⁺) ⊗ proc(a, Pₛ) ⊗ ![G(c → sp a)] ) }

cut⁻ : proc(c, x:A⁻ ← Pₛ ; Qₛ) ⊗ ![c /∈ H]
→ { (∃ a′. proc(a, Qₛ a′) ⊗ queue(a′:A⁻, b, a:A⁻) ⊗ proc(a, Pₛ) ⊗ ![G(c → sp a)] ) }

one_r : proc(c, wait a′ ; Q) ⊗ queue(a′:1, end, _) ⊗ ![c /∈ H] → { proc(c, Q) }

one_rₜ : proc(c, wait a′ ; Q) ⊗ queue(a′:1, end, _) ⊗ ![c /∈ H] → { proc(c, Q) ⊗ ![dangling(a′)] }

one_s : queue(a′:B, q, a:1) ⊗ proc(a, close a) ⊗ ![monitor(end ▷ 1)] → { queue(a′:B, q, end, R) }

one_sₜ : queue(a′:B, q, a:A) ⊗ proc(a, close a) ⊗ ![monitor(¬(end ▷ A))] → { alarm(a) }

one_sₜₜ : queue(a′:B, q, a:A) ⊗ proc(a, close b) ⊗ ![monitor(a ≠ b)] → { alarm(a) }

up_s : proc(a, send c′ shift : Qₛ) ⊗ queue(c′:B⁺, q, c:C⁻) ⊗ ![monitor(shift ▷ B⁺)]
→ { proc(a, Qₛ c′) ⊗ queue(c′:B⁺, shift ▷ q, c:C⁻) }

down_s : queue(c′:C⁺, p, c, B⁻) ⊗ proc(c, send c shift ; Q) ⊗ ![monitor(shift ▷ B⁻)]
→ { queue(c′:C⁺, p, shift ▷ c, B⁻) ⊗ proc(c, Q) }

with_s : proc(c, a′.labj ; Q) ⊗ queue(a′: & { labi : A⁺ }] , q, a: B) ⊗ ![monitor(a′.labj ▷ & { labi : A⁺ }] )
→ { proc(c, Q) ⊗ queue(a′:A⁺, labj, q, a:B) }

with_sₜ : proc(c, a′.labj ; Q) ⊗ queue(a′:A, q, a: B) ⊗ ![monitor(¬(a′.labj ▷ A))] → { alarm(a) }

plus_s : queue(a′:B, p, a, a′ ⊕ { labi : A⁺ }, q, a: B) ⊗ ![monitor(a′.labj ▷ & { labi : A⁺ }] )
→ { queue(a′:B, p, labj, a:A⁺) ⊗ proc(a, Q) }

plus_sₜ : queue(a′:B, p, a:A) ⊗ proc(a, labj ; Q) ⊗ ![monitor(¬(a′.labj ▷ A))] → { alarm(a) }

plus_sₜₜ : queue(a′:B, p, a:A) ⊗ proc(a, labj ; Q) ⊗ ![monitor(a ≠ b)] → { alarm(a) }

Figure 11: Monitoring rules.
tensor_s1 : \text{proc}(c, \text{send} \ a \ (y \leftarrow P_y) ; Q) \otimes ![a \notin H] \\
\rightarrow^3 \exists b'. \text{proc}(c, \text{send} \ a \ \text{ch}(b') ; Q) \\
\otimes \text{queue}(b' ; [], \cdot, b[]) \otimes \text{proc}(b, P_b) \otimes \{H(a \rightarrow_{sp} b)\}

tensor_sh1 : \text{proc}(a, \text{send} \ a \ (y \leftarrow P_y) ; Q) \otimes ![a \in H] \\
\rightarrow^3 \exists b'. \text{proc}(c, \text{send} \ a \ \text{ch}(b') ; Q) \\
\otimes \text{queue}(b' ; [], \cdot, b[]) \otimes \text{proc}(b, P_b) \otimes \{H(a \rightarrow_{sp} b)\}

tensor_s2 : \text{queue}(a':C^+, p, a:A^+ \otimes B^+) \otimes \text{proc}(c, \text{send} \ a \ \text{ch}(b') ; Q) \otimes \text{queue}(b' ; [], \cdot, b[]) \\
\otimes ![\text{monitor}(\text{ch}(b') \triangleright A^+ \otimes B^+)] \\
\rightarrow^2 \text{queue}(a':C^+, p \cdot b', a:B^+) \otimes \text{proc}(a, Q) \otimes \text{queue}(b':A^+, p, b:A^+) \otimes \text{proc}(b, P_b)

lollis1 : \text{proc}(a, \text{send} \ c' \ (y \leftarrow P_y) ; Q) \otimes ![a \notin H] \\
\rightarrow^3 \exists b'. \text{proc}(a, \text{send} \ c' \ \text{ch}(b') ; Q) \\
\otimes \text{queue}(b' ; [], \cdot, b[]) \otimes \text{proc}(b, P_b) \otimes \{H(a \rightarrow_{sp} b)\}

lollis1h : \text{proc}(a, \text{send} \ c' \ (y \leftarrow P_y) ; Q) \otimes ![a \in H] \\
\rightarrow^3 \exists b'. \text{proc}(a, \text{send} \ c' \ \text{ch}(b') ; Q) \\
\otimes \text{queue}(b' ; [], \cdot, b[]) \otimes \text{proc}(b, P_b) \otimes \{H(a \rightarrow_{sp} b)\}

lollis2 : \text{queue}(c':A^+ \rightarrow B^-, p, c:C^-) \otimes \text{proc}(a, \text{send} \ c' \ \text{ch}(b') ; Q) \otimes \text{queue}(b' ; [], \cdot, b[]) \\
\otimes ![\text{monitor}(\text{ch}(b') \triangleright A^+ \rightarrow B^-)] \\
\rightarrow^2 \text{queue}(c':B^+, p \cdot b, c:C^+) \otimes \text{proc}(a, Q) \otimes \text{queue}(b':A^+, p, b:A^+) \otimes \text{proc}(b, P_b)

alarm_{m} : \text{proc}(c, \text{send} \ a \ m ; R) \otimes \text{queue}(a':A, q, a:B) \otimes ![\text{monitor}(\neg (m \triangleright A))] \rightarrow \{\text{alarm}(c)\}

alarm'_{m} : \text{queue}(c', A, q, c:B) \otimes \text{proc}(c, \text{send} \ c \ m ; R) \otimes ![\text{monitor}(\neg (m \triangleright B))] \rightarrow \{\text{alarm}(c)\}

alarm_{p} : \text{queue}(c':A, q, b; B) \otimes \text{proc}(a, \text{send} \ b \ m ; R) \otimes ![\text{monitor}(a \neq b)] \rightarrow \{\text{alarm}(a)\}

havoc : \text{proc}(c, P) \otimes ![\text{fn}(P) \supseteq \text{fn}(Q)] \rightarrow \{\text{proc}(c, Q) \otimes ![\text{havoc}(c)\}

Figure 12: Monitoring rules (cont.)