Session types for access and information flow control ⋆

Sara Capecchi[1], Ilaria Castellani[2],
Mariangiola Dezani-Ciancaglini[1], and Tamara Rezk[2]

[1] Dipartimento di Informatica, Università di Torino, corso Svizzera 185, 10149 Torino, Italy
[2] INRIA, 2004 route des Lucioles, 06902 Sophia Antipolis, France

Abstract. We consider a calculus for multiparty sessions with delegation, enriched with security levels for session participants and data. We propose a type system that guarantees both session safety and a form of access control. Moreover, this type system ensures secure information flow, including controlled forms of declassification. In particular, the type system prevents leaks that could result from an unrestricted use of the control constructs of the calculus, such as session opening, selection, branching and delegation. We illustrate the use of our type system with a number of examples, which reveal an interesting interplay between the constraints used in security type systems and those used in session types to ensure properties like communication safety and session fidelity.

Keywords: concurrency, communication-centred computing, session types, access control, secure information flow.

1 Introduction

With the advent of web technologies and the proliferation of programmable and interconnectable devices, we are faced today with a powerful and heterogeneous computing environment. This environment is inherently parallel and distributed and, unlike previous computing environments, it heavily relies on communication. It therefore calls for a new programming paradigm which is sometimes called *communication-centred*. Moreover, since computations take place concurrently in all kinds of different devices, controlled by parties which possibly do not trust each other, security properties such as the confidentiality and integrity of data become of crucial importance. The issue is then to develop models, as well as programming abstractions and methodologies, to be able to exploit the rich potential of this new computing environment, while making sure that we can harness its complexity and get around its security vulnerabilities. To this end, calculi and languages for communication-centred programming have to be *security-minded* from their very conception, and make use of specifications not only for data structures, but also for communication interfaces and for security properties.

The aim of this paper is to investigate type systems for safe and secure sessions. A *session* is an abstraction for various forms of "structured communication" that may occur in a parallel and distributed computing environment. Examples of sessions are a client-service negotiation, a financial transaction, or a multiparty interaction among different services within a web application.

Language-based support for sessions has now become the subject of active research. Primitives for enabling programmers to code sessions in a flexible way, as well as type

⋆ Work partially funded by the INRIA Sophia Antipolis COLOR project MATYSS, by the ANR-SETI-06-010 and ANR-08-EMER-010 grants, and by the MIUR Projects DISCO and IPODS.

systems ensuring the compliance of programs to session specifications (session types), have been studied in a variety of calculi and languages in the last decade. *Session types* were originally introduced in a variant of the pi-calculus [20]. We refer to [7] for a survey on the session type literature. The key properties ensured by session types are *communication safety*, namely the consistency of the communication patterns exhibited by the partners (implying the absence of communication errors), and *session fidelity*, ensuring that channels which carry messages of different types do it in a specific order.

Enforcement of security properties via session types has been studied in [3, 15]. These papers propose a compiler which, given a multiparty session description, implements cryptographic protocols that guarantee *session execution integrity*. The question of ensuring *access control* in binary sessions has been recently addressed in [13] for the Calculus of Services with Pipelines and Sessions of [4], where delegation is absent. On the other hand, the property of *secure information flow* has not been investigated within session calculi so far. This property, first studied in the early eighties [9], has regained interest in the last decade, due to the evolution of the computing environment. It has now been thoroughly studied for both programming languages (cf [16] for a review) and process calculi [8, 10, 12].

In this paper, we address the question of incorporating mandatory access control and secure information flow within session types. We consider a calculus for multiparty sessions with delegation, enriched with security levels for both session participants and data, and providing a form of declassification for data [18], as required by most practical applications. We propose a type system that ensures access control, namely that each participant receives data of security level less than or equal to its own. For instance, in a well-typed session involving a Customer, a Seller and a Bank, the secret credit card number of the Customer will be communicated to the Bank, but not to the Seller. Moreover, our type system prevents insecure flows that could occur via the specific constructs of the language, such as session opening, selection, branching and delegation. Finally, we show that it allows controlled forms of declassification, namely those permitted by the access control policy. Our work reveals an interesting interplay between the constraints of security type systems and those used in session types to ensure properties like communication safety and session fidelity.

The rest of the paper is organised as follows. In Section 2 we motivate our access control and declassification policies with an example. Section 3 introduces the syntax and semantics of our calculus. In Section 4 we define the secure information flow property. In Section 5 we illustrate this property by means of examples. Section 6 presents our type system for safe and secure sessions and theorems establishing its soundness. Section 7 concludes with a discussion on future work. The reader is referred to the full paper (available at http://hal.inria.fr/INRIA) for complete definitions and proofs.

2 A First Example on Access Control and Declassification

In this section we illustrate by an example the basic features of our typed calculus, as well as our access control policy and its use for declassification. The question of secure information flow will only be marginal here. It will be discussed in Sections 4 and 5.

A client `C` sends the title of a book to a bookseller `S`. Then `S` *delegates* to a bank `B` both the reception of the credit card number of `C` and the control of its validity. This

delegation is crucial for assuring the secrecy of the credit card number, which should be read by B but not by S. Then B notifies S about the result of the control: for this a *declassification* is needed. Finally, if the credit card is valid, C receives a delivery date from S, otherwise the deal falls through. More precisely, the protocol is as follows:

1. C opens a connection with S and sends a title to S;
2. S opens a connection with B and *delegates* to B part of his conversation with C;
3. C sends his secret credit card number *apparently* to the untrusted party S but *really* - thanks to delegation - to the trusted party B;
4. B delegates back to S the conversation with C;
5. B selects the answer ok or ko for S depending on the validity of the credit card, thus performing a declassification;
6. S sends to C either ok and a date, or just ko, depending on the label ok or ko chosen by B.

In our calculus, which is an enrichment with security levels of the calculus in [2], this scenario may be described as the parallel composition of the following processes, where security levels appear as superscripts on both data and operators (here we omit unnecessary levels on operators and use $\bot$ to mean "public" and $\top$ to mean "secret"):

$$\mathtt{I} = \bar{a}[2] \mid \bar{b}[2]$$

$$\mathtt{C} = a[1](\alpha_1).\alpha_1!\langle 2, \mathsf{Title}^{\bot}\rangle.\alpha_1!^{\bot}\langle 2, \mathsf{CreditCard}^{\top}\rangle.\alpha_1\&(2, \{\mathsf{ok} : \alpha_1?(2, date^{\bot}).\mathbf{0}, \mathsf{ko} : \mathbf{0}\})$$

$$\begin{aligned}\mathtt{S} = a[2](\alpha_2).\alpha_2?(1, x^{\bot}).b[2](\beta_2).\beta_2!\langle\langle 1, \alpha_2\rangle\rangle.\beta_2?((1, \zeta)).\\ \beta_2\&(1, \{\mathsf{ok} : \zeta \oplus \langle 1, \mathsf{ok}\rangle.\zeta!\langle 1, \mathsf{Date}^{\bot}\rangle.\mathbf{0}, \mathsf{ko} : \zeta \oplus \langle 1, \mathsf{ko}\rangle.\mathbf{0}\})\end{aligned}$$

$$\begin{aligned}\mathtt{B} = b[1](\beta_1).\beta_1?((2, \zeta)).\zeta?^{\top}(2, cc^{\bot}).\beta_1!\langle\langle \zeta, 2\rangle\rangle.\\ \mathsf{if}\ \mathit{valid}(cc^{\bot})\ \mathsf{then}\ \beta_1 \oplus \langle 2, \mathsf{ok}\rangle.\mathbf{0}\ \mathsf{else}\ \beta_1 \oplus \langle 2, \mathsf{ko}\rangle.\mathbf{0}\end{aligned}$$

A session is a particular activation of a service, involving a number of parties with predefined roles. Here processes C and S communicate by opening a session on service a, while processes S and B communicate by opening a session on service b. The initiators $\bar{a}[2]$ and $\bar{b}[2]$ specify the number of participants of each service. We associate integers with participants in services: here C=1, S=2 in service a and B=1, S=2 in service b.

In process C, the prefix $a[1](\alpha_1)$ means that C wants to act as participant 1 in service a using channel α_1, matching channel α_2 of participant 2, who is S. When the session is established, C sends to S a title of level $\bot$ and a credit card number of level $\top$, indicating (by the superscript $\bot$ on the output operator) that the credit card number may be declassified to $\bot$. Then he waits for either ok, followed by a date, or ko.
indent Process S receives a value in service a and then enters service b as participant 2. Here the output $\beta_2!\langle\langle 1, \alpha_2\rangle\rangle$ sends channel α_2 to the participant 1 of b, who is B, thus delegating to B the use of α_2. Then S waits for a channel ζ from B. Henceforth, S communicates using both channels β_2 and ζ: on channel β_2 he waits for one of the labels ok or ko, which he then forwards to C on ζ, sending also a date if the label is ok. Forgetting session opening and abstracting from values to types, we may represent the whole communication protocol by the *global types* of Table 1 (where we use B, C, S instead of 1, 2), where the left-hand side and right-hand side describe services a and b, respectively. Line 1 says that C sends a String of level $\bot$ to S. In line 2, $\mathtt{S}\!\upharpoonright\!\delta$ means that the channel from S to C is delegated: this delegation is realised by the transmission of the channel (with type T) from S to B, as shown on the right-hand side. Line 3 says that

1. $\mathtt{C} \to \mathtt{S} : \left\langle \mathsf{String}^{\bot} \right\rangle$		
2. $\mathtt{S}\, \Rsh \delta$		$\mathtt{S} \to \mathtt{B} : \langle T \rangle$
3. $\mathtt{C} \to \mathtt{S} : \left\langle \mathsf{Number}^{\top\downarrow\bot} \right\rangle$		
4. $\mathtt{S}\, \Rsh \delta$		$\mathtt{B} \to \mathtt{S} : \langle T' \rangle$
5.		$\mathtt{B} \to \mathtt{S} : \{\mathsf{ok} : \mathsf{end}, \mathsf{ko} : \mathsf{end}\}$
6. $\mathtt{S} \to \mathtt{C} : \{\mathsf{ok} : \mathtt{S} \to \mathtt{C} : \left\langle \mathsf{String}^{\bot} \right\rangle ; \mathsf{end}, \mathsf{ko} : \mathsf{end}\}$		

Table 1. Global types of the B, C, S example.

C sends a Number of level $\top$ to S, allowing him to declassify it to $\bot$. Notice that due to the previous delegation the Number is received by B and not by S. Line 4 describes a delegation which is the inverse of that in Line 2: here the (behavioural) type of the channel has changed, since the channel has already been used to receive the Number. Line 5 says that B sends to S one of the labels ok or ko. Finally, line 6 says that S sends to C either the label ok followed by a String of level $\bot$, or the label ko. Since B's choice of the label ok or ko depends on a test on the Number, it is crucial that Number be previously declassified to $\bot$, otherwise the reception of a String of level $\bot$ by C would depend on a value of level $\top$ (this is where secure information flow comes into play).

Type T represents the conversation between C and S after the first communication, seen from the viewpoint of S. Convening that $?(-), !\langle - \rangle$ represent input and output in types, that ";" stands for sequencing and that $\oplus\langle -\{-\}\rangle$ represents the choice of sending one among different labels, it is easy to see that the *session type* T is:

$$?\left(\mathtt{C}, \mathsf{Number}^{\top\downarrow\bot}\right); \oplus \left\langle \mathtt{C}, \{\mathsf{ok} :! \left\langle \mathtt{C}, \mathsf{String}^{\bot} \right\rangle ; \mathsf{end}, \mathsf{ko} : \mathsf{end}\} \right\rangle$$

where the communication partner of S (namely C) is explicitly mentioned. The session type T' is the rest of type T after the first communication has been done:

$$\oplus \left\langle \mathtt{C}, \{\mathsf{ok} :! \left\langle \mathtt{C}, \mathsf{String}^{\bot} \right\rangle ; \mathsf{end}, \mathsf{ko} : \mathsf{end}\} \right\rangle$$

To formalise access control, we will give security levels to service participants, and require that a participant of a given level does not receive data of higher or incomparable level. Since the only secret data in our example is CreditCard, it is natural to associate $\bot$ with S in both services a and b, and $\top$ with B in service b. Notice that C may indifferently have level $\top$ or $\bot$, since it only sends, but does not receive, the high data CreditCard.

3 Syntax and Semantics

Our calculus for multiparty asynchronous sessions is essentially the same as that considered in [2], with the addition of runtime configurations and security levels.

Syntax. Let $(\mathscr{S}, \sqsubseteq)$ be a finite lattice of *security levels*, ranged over by ℓ, ℓ'. We denote by $\sqcup$ and $\sqcap$ the join and meet operations on the lattice, and by $\bot$ and $\top$ its minimal and maximal elements.

We assume the following sets: *service names*, ranged over by $a, b, \ldots$ each of which has an *arity* $n \geq 2$ (its number of participants) and a security level ℓ, *value variables*, ranged over by $x, y, \ldots$, all decorated with security levels, *identifiers*, i.e., service names and value variables, ranged over by $u, w, \ldots$, all decorated with security levels, *channel*

$P ::= \bar{u}[n]$	n-ary session initiator			
$\mid\ u[\mathtt{p}](\alpha).P$	p-th session participant	$c ::= \alpha \mid s[\mathtt{p}]$	Channel	
$\mid\ c!^{\ell}\langle \Pi, e\rangle.P$	Value sending	$e ::= v^{\ell} \mid x^{\ell}$		
$\mid\ c?^{\ell}(\mathtt{p}, x^{\ell'}).P$	Value receiving	$\mid\ e \text{ and } e' \mid \text{not } e \ldots$	Expression	
$\mid\ c!^{\ell}\langle\langle \mathtt{q}, c'\rangle\rangle.P$	Delegation sending			
$\mid\ c?^{\ell}((\mathtt{p}, \alpha)).P$	Delegation reception	$D ::= X(x^{\ell}, \alpha) = P$	Declaration	
$\mid\ c \oplus^{\ell} \langle \Pi, \lambda\rangle.P$	Selection	$\Pi ::= \{\mathtt{p}\} \mid \Pi \cup \{\mathtt{p}\}$	Set of participants	
$\mid\ c\&^{\ell}(\mathtt{p}, \{\lambda_i : P_i\}_{i\in I})$	Branching			
$\mid\ \text{if } e \text{ then } P \text{ else } Q$	Conditional	$\vartheta ::= v^{\ell\downarrow\ell'} \mid s[\mathtt{p}]^{\ell} \mid \lambda^{\ell}$	Message content	
$\mid\ P \mid Q$	Parallel			
$\mid\ \mathbf{0}$	Inaction	$m ::= (\mathtt{p}, \Pi, \vartheta)$	Message in transit	
$\mid\ (\nu a^{\ell})P$	Name hiding			
$\mid\ \text{def } D \text{ in } P$	Recursion	$h ::= m \cdot h \mid \varepsilon$	Queue	
$\mid\ X\langle e, c\rangle$	Process call	$H ::= H \cup \{s : h\} \mid \emptyset$	**Q**-set	
$u ::= x^{\ell} \mid a^{\ell}$	Identifier	$r ::= a^{\ell} \mid s$	Service/Session Name	
$v ::= a \mid \text{true} \mid \text{false} \mid \ldots$	Value			

Table 2. Syntax of expressions, processes, queues, and configurations

variables, ranged over by $\alpha, \beta, \ldots$, *labels*, ranged over by $\lambda, \lambda', \ldots$ (acting like labels in labelled records). *Values* v are either service names or basic values (boolean values, integers, etc.). When treated as an expression, a value is decorated with a security level ℓ; when used in a message, it is decorated with a declassified level of the form $\ell \downarrow \ell'$, where $\ell' \leq \ell$ (in case $\ell' = \ell$, we will write simply ℓ instead of $\ell \downarrow \ell$).

Sessions, the central abstraction of our calculus, are denoted with $s, s' \ldots$. A session represents a particular instance or activation of a service. Hence sessions only appear at runtime. We use $\mathtt{p}$, $\mathtt{q}$,... to denote the *participants of a session. In an n-ary session (a session corresponding to an n-ary service)* $\mathtt{p}$, $\mathtt{q}$ *are assumed to range over the natural numbers* $1, \ldots, n$. We denote by Π a non empty set of participants. Each session s has an associated set of *channels with role* $s[\mathtt{p}]$, one for each participant. Channel $s[\mathtt{p}]$ is the private channel through which participant $\mathtt{p}$ communicates with the other participants in the session s. A new session s on an n-ary service a^{ℓ} is opened when the *initiator* $\bar{a}^{\ell}[n]$ of the service synchronises with n processes of the form $a^{\ell}[\mathtt{p}](\alpha).P$. We use c to range over channel variables and channels with roles. Finally, we assume a set of *process variables* $X, Y, \ldots$, in order to define recursive behaviours.

The set of *expressions*, ranged over by $e, e', \ldots$, and the set of *processes*, ranged over by $P, Q \ldots$, are given by the grammar in Table 2, where syntax occurring only at runtime appears shaded. The primitives are decorated with security levels. When there is no risk of confusion we will omit the set delimiters $\{,\}$.

As in [11], in order to model TCP-like asynchronous communications (with non-blocking send but message order preservation between a given pair of participants), we use *queues of messages*, denoted by h; an element of h may be a value message $(\mathtt{p}, \Pi, v^{\ell\downarrow\ell'})$, indicating that the value v^{ℓ} is sent by participant $\mathtt{p}$ to all participants in Π, with the right of declassifying it from ℓ to ℓ'; a channel message $(\mathtt{p}, \mathtt{q}, s[\mathtt{p}']^{\ell})$, indicating that $\mathtt{p}$ delegates to $\mathtt{q}$ the role of $\mathtt{p}'$ with level ℓ in the session s; and a label message

$(\mathtt{p}, \Pi, \lambda^\ell)$, indicating that $\mathtt{p}$ selects the process with label λ among the processes offered by the set of participants Π. The empty queue is denoted by ε, and the concatenation of a new message m to a queue h by $h \cdot m$. Conversely, $m \cdot h$ means that m is the head of the queue. Since there may be nested and parallel sessions, we distinguish their queues by naming them. We denote by $s : h$ the *named queue* h associated with session s. We use H, K to range over sets of named queues, also called **Q**-sets.

Operational Semantics. The operational semantics is defined on configurations. A *configuration* is a pair $C = < P \ , \ H >$ of a process P and a **Q**-set H, possibly restricted with respect to service and session names, or a parallel composition of configurations, denoted by $C \| C$. In a configuration $(\nu s) < P \ , \ H >$, all occurrences of $s[\mathtt{p}]$ in P and H and of s in H are bound. By abuse of notation we will often write P instead of $< P \ , \ \emptyset >$.

We use a *structural equivalence* $\equiv$ [14] for processes, queues and configurations. Modulo $\equiv$, each configuration has the form $(\nu \tilde{r}) < P \ , \ H >$, where $(\nu \tilde{r})C$ stands for $(\nu r_1) \cdots (\nu r_k) C$, if $\tilde{r} = r_1 \cdots r_k$. In $(\nu a^\ell) C$, we assume that α-conversion on the name a^ℓ preserves the level ℓ. Among the rules for queues, we have one for commuting independent messages and another one for splitting a message for multiple recipients.

The transitions for configurations have the form $C \longrightarrow C'$. They are derived using the reduction rules in Table 3. Rule [Link] describes the initiation of a new session among n processes, corresponding to an activation of the service a^ℓ of arity n. After the connection, the participants share a private session name s and the corresponding queue, initialised to $s : \varepsilon$. The variable $\alpha_\mathtt{p}$ in each participant $P_\mathtt{p}$ is replaced by the corresponding channel with role $s[\mathtt{p}]$. The output rules [Send], [DelSend] and [Label] push values, channels and labels, respectively, into the queue $s : h$. In rule [Send], $e \downarrow v^\ell$ denotes the evaluation of the expression e to the value v^ℓ, where ℓ is the join of the security levels of the variables and values occurring in e. The superscript ℓ' on the output sign indicates that v^ℓ can be declassified to level ℓ', when received by an input process $s[\mathtt{q}]?^\ell(\mathtt{p}, x^{\ell'}).P$. This is why the value is recorded with both levels in the queue. The rules [Rec], [DelRec] and [Branch] perform the corresponding complementary operations. As usual, we will use $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

$$a^\ell[1](\alpha_1).P_1 \mid \ldots \mid a^\ell[n](\alpha_n).P_n \mid \bar{a}^\ell[n] \longrightarrow (\nu s) < P_1\{s[1]/\alpha_1\} \mid \ldots \mid P_n\{s[n]/\alpha_n\} \ , \ s : \varepsilon > \qquad \text{[Link]}$$

$$< s[\mathtt{p}]!^{\ell'}\langle \Pi, e \rangle.P \ , \ s : h > \longrightarrow < P \ , \ s : h \cdot (\mathtt{p}, \Pi, v^{\ell \downarrow \ell'}) > \qquad (e \downarrow v^\ell) \qquad \text{[Send]}$$

$$< s[\mathtt{q}]?^\ell(\mathtt{p}, x^{\ell'}).P \ , \ s : (\mathtt{p}, \mathtt{q}, v^{\ell \downarrow \ell'}) \cdot h > \longrightarrow < P\{v^{\ell'}/x^{\ell'}\} \ , \ s : h > \qquad \text{[Rec]}$$

$$< s[\mathtt{p}]!^\ell\langle\langle \mathtt{q}, s'[\mathtt{p}'] \rangle\rangle.P \ , \ s : h > \longrightarrow < P \ , \ s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']^\ell) > \qquad \text{[DelSend]}$$

$$< s[\mathtt{q}]?^\ell((\mathtt{p}, \alpha)).P \ , \ s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']^\ell) \cdot h > \longrightarrow < P\{s'[\mathtt{p}']/\alpha\} \ , \ s : h > \qquad \text{[DelRec]}$$

$$< s[\mathtt{p}] \oplus^\ell \langle \Pi, \lambda \rangle.P \ , \ s : h > \longrightarrow < P \ , \ s : h \cdot (\mathtt{p}, \Pi, \lambda^\ell) > \qquad \text{[Label]}$$

$$< s[\mathtt{q}] \&^\ell (\mathtt{p}, \{\lambda_i : P_i\}_{i \in I}) \ , \ s : (\mathtt{p}, \mathtt{q}, \lambda_{i_0}^\ell) \cdot h > \longrightarrow < P_{i_0} \ , \ s : h > \quad (i_0 \in I) \qquad \text{[Branch]}$$

$$C \longrightarrow (\nu \tilde{s}) C' \quad \Rightarrow \quad (\nu \tilde{r})(C \| C'') \longrightarrow (\nu \tilde{r})(\nu \tilde{s})(C' \| C'') \qquad \text{[ScopC]}$$

Table 3. Reduction rules (excerpt)

4 Information Flow Security in Sessions

We turn now to the question of ensuring *secure information flow* [6] within sessions. We shall be interested in the property of *noninterference* (NI) [9], combined with a limited form of *declassification* [1], which may only take place during a value communication. The property of NI requires that there is no flow of information from objects of a given level to objects of lower or incomparable level [21, 19, 16]. To set the stage for our information flow analysis, the first questions to ask are:

1. Which objects of the calculus should carry security levels?
2. Which information leaks can occur and how can they be detected?

As concerns objects, we shall see that besides values, also labels, delegated channels and services will need security levels. Since this question requires some discussion, which is best understood through examples, we defer it to the next section, just assuming here as a fact that queue messages have the form $(\mathtt{p}, \Pi, \vartheta)$, where ϑ may be $v^{\ell\downarrow\ell'}$, λ^{ℓ} or $s[\mathtt{p}]^{\ell}$. In the rest of this section, we will focus on the observation model, which will be based on bisimulation, as is now standard for concurrent processes [19, 17].

We assume that the observer can see the content of messages in session queues. To fix ideas, one may view the observer as a kind of buffer through which messages may transit while reaching or leaving a session queue. We do not want to go as far as allowing an observer to take part in a session, since that could affect the behaviour of other processes. In other words, we assume a passive observer rather than an active one.

What matters for security is observation relative to a given set of levels. Given a downward-closed subset $\mathscr{L}$ of $\mathscr{S}$, a $\mathscr{L}$-observer will only be able to see messages whose level belongs to $\mathscr{L}$. A notion of $\mathscr{L}$-equality $=_{\mathscr{L}}$ on **Q**-sets is then introduced, representing indistinguishability of **Q**-sets by a $\mathscr{L}$-observer. Based on $=_{\mathscr{L}}$, a notion of $\mathscr{L}$-bisimulation $\simeq_{\mathscr{L}}$ will formalise indistinguishability of processes by a $\mathscr{L}$-observer.

Formally, a queue $s : h$ is $\mathscr{L}$-observable if it contains some message with a level in $\mathscr{L}$. Then two **Q**-sets are *$\mathscr{L}$-equal* if their $\mathscr{L}$-observable queues have the same names and contain the same messages with a level in $\mathscr{L}$. This equality is based on a $\mathscr{L}$-projection operation on **Q**-sets, which discards all messages whose level is not in $\mathscr{L}$.

Definition 1. *Let the functions $lev_{\uparrow}$ and $lev_{\downarrow}$ be defined by:*
$lev_{\uparrow}(v^{\ell\downarrow\ell'}) = \ell$, $lev_{\downarrow}(v^{\ell\downarrow\ell'}) = \ell'$, *and* $lev_{\uparrow}(s[\mathtt{p}]^{\ell}) = lev_{\uparrow}(\lambda^{\ell}) = \ell = lev_{\downarrow}(s[\mathtt{p}]^{\ell}) = lev_{\downarrow}(\lambda^{\ell})$.

Definition 2. *The projection operation $\Downarrow \mathscr{L}$ is defined inductively on messages, queues and* **Q***-sets as follows:*

$$(\mathtt{p}, \Pi, \vartheta) \Downarrow \mathscr{L} = \begin{cases} (\mathtt{p}, \Pi, \vartheta) & \text{if } lev_{\downarrow}(\vartheta) \in \mathscr{L}, \\ \varepsilon & \text{otherwise.} \end{cases}$$

$$\varepsilon \Downarrow \mathscr{L} = \varepsilon$$

$$(m \cdot h) \Downarrow \mathscr{L} = m \Downarrow \mathscr{L} \cdot h \Downarrow \mathscr{L}$$

$$\emptyset \Downarrow \mathscr{L} = \emptyset$$

$$(H \cup \{s : h\}) \Downarrow \mathscr{L} = \begin{cases} H \Downarrow \mathscr{L} \cup \{s : h \Downarrow \mathscr{L}\} & \text{if } h \Downarrow \mathscr{L} \neq \varepsilon, \\ H \Downarrow \mathscr{L} & \text{otherwise.} \end{cases}$$

Definition 3 ($\mathscr{L}$-Equality of Q-sets).

Two **Q***-sets H and K are* $\mathscr{L}$-equal*, written $H =_{\mathscr{L}} K$, if $H \Downarrow \mathscr{L} = K \Downarrow \mathscr{L}$.*

When reducing a configuration $(\nu\tilde{r}) < P\ ,\ H >$, we have to make sure that input prefixes in P "agree" with messages in H. This is assured by our type system given in Section 6.

A relation on processes is a *$\mathscr{L}$-bisimulation* if it preserves $\mathscr{L}$-equality of **Q**-sets at each step, starting from typable configurations:

Definition 4 ($\mathscr{L}$-Bisimulation on processes).
A symmetric relation $\mathscr{R} \subseteq (\mathscr{P}r \times \mathscr{P}r)$ is a $\mathscr{L}$-bisimulation *if $P_1 \mathscr{R} P_2$ implies, for any pair of* **Q***-sets H_1 and H_2 such that $H_1 =_{\mathscr{L}} H_2$ and $< P_1\ ,\ H_1 >, < P_2\ ,\ H_2 >$ are typable:*

If $< P_1\ ,\ H_1 > \longrightarrow (\nu\tilde{r}) < P_1'\ ,\ H_1' >$, then either $H_1' =_{\mathscr{L}} H_2$ and $P_1' \mathscr{R} P_2$, or there exist P_2', H_2' such that $< P_2\ ,\ H_2 > \longrightarrow^* (\nu\tilde{r}) < P_2'\ ,\ H_2' >$, where $H_1' =_{\mathscr{L}} H_2'$ and $P_1' \mathscr{R} P_2'$.

Processes P_1, P_2 are $\mathscr{L}$-bisimilar*, $P_1 \simeq_{\mathscr{L}} P_2$, if $P_1 \mathscr{R} P_2$ for some $\mathscr{L}$-bisimulation $\mathscr{R}$.*

Note that $\tilde{r}$ may be either empty or be a service name or a fresh session name s, and in the last case s cannot occur in P_2 and H_2 by Barendregt convention.

Intuitively, a transition that adds or removes a message with level in $\mathscr{L}$ must be simulated in one or more steps, producing the same effect on the **Q**-set, whereas a transition that only affects messages with level not in $\mathscr{L}$ may be simulated by inaction.

Definition 5 ($\mathscr{L}$-Security). *A program P is* $\mathscr{L}$-secure *if $P \simeq_{\mathscr{L}} P$.*

5 Examples of Information Flow Security in Sessions

In this section we illustrate the various kinds of flow that can occur in our calculus, through simple examples. Since we aim at justifying the introduction of security levels in the syntax (other than on values and participants), we shall initially omit levels in all other objects. In queues, we will use v^{ℓ} as a shorthand for $v^{\ell\downarrow\ell}$. For the sake of simplicity, we assume here just two security levels $\bot$ and $\top$ (also called low and high). In all examples, we suppose $H_1 = \{s : (1, 2, \mathsf{true}^{\top})\}$ and $H_2 = \{s : (1, 2, \mathsf{false}^{\top})\}$.

5.1. High input should not be followed by low actions. A simple example of insecure flow, which is not specific to our calculus but arises in all process calculi with values and a conditional construct, is the following (assuming session s has four participants):

$$s[2]?(1, x^{\top}).\mathsf{if}\ x^{\top}\ \mathsf{then}\ s[2]!\langle 3, \mathsf{true}^{\top}\rangle.\mathbf{0}\ \mathsf{else}\ \mathbf{0} \\ \mid\ s[3]?(2, z^{\top}).s[3]!\langle 4, \mathsf{true}^{\bot}\rangle.\mathbf{0} \mid\ s[4]?(3, y^{\bot}).\mathbf{0}$$

This process is insecure because, depending on the high value received for $x^{\top}$ on channel $s[2]$, that is, on whether the **Q**-set is H_1 or H_2, the low value $\mathsf{true}^{\bot}$ will be emitted or not on channel $s[3]$, leading to $H_1' = \{s : (3, 4, \mathsf{true}^{\bot})\} \neq_{\mathscr{L}} H_2' = \{s : \varepsilon\}$ if $\mathscr{L} = \{\bot\}$. This shows that a high input should not be followed by a low output. Note that the reverse is not true, since output is not blocking: if we swapped the polarities of input and output in the third participant (and adjusted them accordingly in the other participants), then the resulting process would be secure.

Let us point out that this process is not typable in a classical session type system, since the session types of the conditional branches are not the same. However, it would become typable if the second branch of the conditional were replaced by the deadlocked process $(\nu b)b[1](\beta_1).s[2]!\langle 3, \mathsf{true}^{\top}\rangle.\mathbf{0}$. The expert reader will notice that by adding to

our type system the interaction typing of [2] (which enforces global progress) we would rule out also this second process. On the other hand, the interaction typing does not prevent deadlocks due to inverse session calls, as for instance:

$$\begin{array}{ll} \bar{b}[2] & \mid\ b[1](\beta_1).c[1](\gamma_1).s[2]!\langle 3,\mathsf{true}^\top\rangle.\mathbf{0} \\ \bar{c}[2] & \mid\ c[2](\gamma_2).b[2](\beta_2).\mathbf{0} \end{array}$$

Clearly, this deadlock could be used to implement the insecure flow in our example.

5.2. Need for levels on services. Consider the following process:

$$\begin{array}{l} s[2]?(1,x^\top).\mathsf{if}\ x^\top\ \mathsf{then}\ \bar{b}[2]\ \mathsf{else}\ \mathbf{0} \\ \mid\ b[1](\beta_1).\beta_1!\langle 2,\mathsf{true}^\perp\rangle.\mathbf{0} \mid\ b[2](\beta_2).\beta_2?(1,y^\perp).\mathbf{0} \end{array}$$

This process is insecure because, depending on the high value received for $x^\top$, it will initiate or not a session on service b, which performs a low value exchange. To rule out this kind of leak we annotate service names with security levels which are a lower bound for all the actions they execute. Then service b must be of level $\top$, since it appears in the branch of a $\top$-conditional, and hence it will not allow the output of the value $\mathsf{true}^\perp$.

5.3. Need for levels on selection and branching. Consider the following process:

$$\begin{array}{l} s[2]?(1,x^\top).\mathsf{if}\ x^\top\ \mathsf{then}\ s[2]\oplus\langle 3,\lambda\rangle.\mathbf{0}\ \mathsf{else}\ s[2]\oplus\langle 3,\lambda'\rangle.\mathbf{0} \\ \mid\ s[3]\&(2,\{\lambda : s[3]!\langle 4,\mathsf{true}^\perp\rangle.\mathbf{0},\lambda' : s[3]!\langle 4,\mathsf{false}^\perp\rangle.\mathbf{0}\}) \\ \mid\ s[4]?(3,y^\perp).\mathbf{0} \end{array}$$

This process is insecure because a selection in one participant, which depends on a high value, causes the corresponding branching participant to emit two different low values. To prevent this kind of leak, the selection and branching operators will be annotated with a security level which is a lower bound for all actions executed in the branches.

5.4. Need for levels on delegated channels. Consider the following process:

$$\begin{array}{l} s[2]?(1,x^\top).\mathsf{if}\ x^\top\ \mathsf{then}\ s[2]!\langle\langle 3,s'[1]\rangle\rangle.s[2]!\langle\langle 4,s''[1]\rangle\rangle.\mathbf{0}\ \mathsf{else}\ s[2]!\langle\langle 3,s''[1]\rangle\rangle.s[2]!\langle\langle 4,s'[1]\rangle\rangle.\mathbf{0} \\ \mid\ s[3]?((2,\eta)).\eta!\langle 2,\mathsf{true}^\perp\rangle.\mathbf{0} \mid\ s[4]?((2,\eta')).\eta'!\langle 2,\mathsf{false}^\perp\rangle.\mathbf{0} \\ \mid\ s'[2]?(1,x^\perp).\mathbf{0} \mid\ s''[2]?(1,y^\perp).\mathbf{0} \end{array}$$

This process is insecure because, depending on the high value received for $x^\top$, the participants 3 and 4 of s will be delegated to participate in sessions s' and s'', or vicevversa, feeding the queues of s' and s'' with different low values. This shows that delegation send and receive should also carry a level, which will be a lower bound for all actions executed in the receiving participant after the delegation.

5.5. Levels in queue messages. So far, we have identified which objects of the calculus need security levels, namely: values, service names, and the operators of selection, branching and delegation. We now discuss how levels are recorded into queue messages.

Values are recorded in the queues with both their level and their declassified level. The reason for recording also the declassified level is access control: the semantics does not allow a low input process to fetch a high value declassified to low. More formally, a value $v^{\top\downarrow\perp}$ in the queue can only be read by a process $s[\mathtt{q}]?^\top(\mathtt{p},x^\perp).P$. Concerning service names a^ℓ, the level ℓ guarantees that the session initiator and all the participants get started in a context of level $\ell'\leq\ell$ (see Example 5.2). Once the session is established, the name a^ℓ disappears and it is its global type (cf next section) that will ensure that all participants perform actions of levels greater than or equal to ℓ. As for the operators of

branching/selection and delegation, they disappear after the reduction and their level is recorded respectively into labels and delegated channels within queue messages. This is essential since in this case the communication is asynchronous and occurs in two steps. Hence queue messages have the form $(\mathtt{p}, \Pi, \vartheta)$, where ϑ is $v^{\ell\downarrow\ell'}$, λ^{ℓ} or $s[\mathtt{p}]^{\ell}$.

6 Type system

In this section we present our type system for secure sessions and state its properties. Just like process syntax, types will contain security levels.

Safety Global Types, Session Types, and Projections. A *safety global type* is a pair $\langle \mathsf{L}, G\rangle^{\ell}$, decorated with a security level ℓ, describing a service where:

- $\mathsf{L} : \{1,\ldots,n\} \to \mathscr{S}$ is a *safety mapping* from participants to security levels;
- G is a *global type*, describing the whole conversation scenario of an n-ary service;
- ℓ is the meet of all levels appearing in G, denoted by $\mathsf{M}(G)$.

The grammar of global types is:

$$\begin{array}{lll} \text{Global} & G ::= \mathtt{p} \to \Pi : \langle U\rangle.G & \text{Exchange } U ::= S^{\ell\downarrow\ell'} \mid T \mid \langle \mathsf{L}, G\rangle^{\ell} \\ & \mid\ \mathtt{p} \to \Pi : \{\lambda_i : G_i\}_{i\in I}^{\ell} & \text{Sorts } S ::= \mathsf{bool} \mid \ldots \\ & \mid\ \mathtt{p}\,\Gamma\!\!\!\uparrow\delta.G & \\ & \mid\ \mu\mathbf{t}.G \mid \mathbf{t} \mid \mathsf{end} & \end{array}$$

The type $\mathtt{p} \to \Pi : \langle U\rangle.G$ says that participant $\mathtt{p}$ multicasts a message of type U to all participants in Π and then the interactions described in G take place. *Exchange types* U may be *sort* types $S^{\ell\downarrow\ell'}$ for values (base types decorated with a declassification $\ell \downarrow \ell'$), *session types* T for channels (defined below), or safety global types for services. If $U = T$, then Π is a singleton $\{\mathtt{q}\}$. We use S^{ℓ} as short for $S^{\ell\downarrow\ell}$, called a *trivial declassification*. Type $\mathtt{p} \to \Pi : \{\lambda_i : G_i\}_{i\in I}^{\ell}$, where $\ell = \bigsqcap_{i\in I} \mathsf{M}(G_i)$, says that participant $\mathtt{p}$ multicasts one of the labels λ_i to the participants in Π. If λ_j is sent, interactions described in G_j take place. Type $\mathtt{p}\,\Gamma\!\!\!\uparrow\delta.G$ says that the role of $\mathtt{p}$ is delegated to another participant; this construct does not appear in the original global types of [11]. It is needed here to "mark" the delegated part of the type, which is discharged when calculating its join (see below).

Type $\mu\mathbf{t}.G$ is a recursive type, where the type variable $\mathbf{t}$ is guarded in the standard way. In the grammar of exchange types, we suppose that G does not contain free type variables. Type end represents the termination of a session. While global types represent the whole session protocol, *session types* correspond to the communication actions, representing each participant's contribution to the session.

As for $\mathsf{M}(G)$, we denote by $\mathsf{M}(T)$ the meet of all security levels appearing in T.

$$\begin{array}{lrlrl} \text{Session} & T ::= & !\langle \Pi, S^{\ell\downarrow\ell'}\rangle;T & \textit{send} & \mid\ ?(\mathtt{p}, S^{\ell\downarrow\ell'});T \quad \textit{receive} \\ & \mid & !^{\ell}\langle \mathtt{q}, T\rangle;T' & \textit{delsend} & \mid\ ?^{\ell}(\mathtt{p}, T);T' \quad \textit{delreceive} \\ & \mid & \oplus^{\ell}\langle \Pi, \{\lambda_i : T_i\}_{i\in I}\rangle & \textit{selection} & \mid\ \&^{\ell}(\mathtt{p}, \{\lambda_i : T_i\}_{i\in I}) \quad \textit{branching} \\ & \mid & \mu\mathbf{t}.T & \textit{recursive} & \mid\ \mathbf{t} \quad \textit{variable} \\ & \mid & \Gamma\!\!\!\uparrow\delta;T & \textit{delegation} & \mid\ \mathsf{end} \quad \textit{end} \end{array}$$

The *send* type $!\langle \Pi, S^{\ell\downarrow\ell'}\rangle;T$ expresses the sending to all participants in Π of a value of type S, of level ℓ declassified to ℓ', followed by the communications described in T. The *delsend* type $!^{\ell}\langle \mathtt{q}, T\rangle;T'$, where $\ell = \mathsf{M}(T)$, says that a channel of type T is sent

to participant $\mathtt{q}$, and then the protocol specified by T' takes place. The *selection* type $\oplus^{\ell}\langle \Pi, \{\lambda_i : T_i\}_{i\in I}\rangle$, where $\ell = \bigsqcap_{i\in I} \mathsf{M}(T_i)$, represents the transmission to all participants in Π of a label λ_j in $\{\lambda_i \mid i \in I\}$, followed by the communications described in T_j. The *delegation* type $\triangledown\delta; T$, says that the communications described in T will be delegated to another agent. The *receive*, *delreceive* and *branching* types are dual to the *send*, *delsend*, and *selection* ones. The type system will assure that $\ell' \leq \mathsf{M}(T)$ in type $?(\mathtt{p}, S^{\ell\downarrow\ell'}); T$, that $\ell \leq \mathsf{M}(T')$ in type $?^{\ell}(\mathtt{p}, T); T'$ and that $\ell = \bigsqcap_{i\in I} \mathsf{M}(T_i)$ in type $\&^{\ell}(\mathtt{p}, \{\lambda_i : T_i\}_{i\in I})$. In all cases, the need for the security level ℓ is motivated by one of the examples in Section 5.

The relation between global types and session types is formalised by the notion of projection [11]. The *projection of G onto* $\mathtt{q}$, denoted $(G \upharpoonright \mathtt{q})$, gives participant $\mathtt{q}$'s view of the protocol described by G. For example the projection of $G = \mathtt{p} \to \mathtt{p}' : \langle T\rangle.G'$ on $\mathtt{q}$ is the following, assuming $\ell = \mathsf{M}(T)$:

$$(\mathtt{p} \to \mathtt{p}' : \langle T\rangle.G') \upharpoonright \mathtt{q} = \begin{cases} !^{\ell}\langle \mathtt{p}', T\rangle; (G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}, \\ ?^{\ell}(\mathtt{p}, T); (G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}', \\ G' \upharpoonright \mathtt{q} & \text{otherwise} \end{cases}$$

Well-formedness of safety global types. To formulate the well-formedness condition for safety global types, we define the join $\mathsf{J}(T)$ of a session type T. Intuitively, while $\mathsf{M}(T)$ is needed for secure information flow, $\mathsf{J}(T)$ will be used for access control. Recall from Section 2 our access control policy, requiring that participants in a session only read data of level less than or equal to their own level. This motivates our (slightly non standard) definition of join: in short, $\mathsf{J}(T)$ is the join of all the security levels decorating the input constructs in T (*receive, delreceive, branching*). Moreover, unlike $\mathsf{M}(T)$, $\mathsf{J}(T)$ forgets the delegated part of T.

This leads to the following condition of well-formedness for safety global types, where $dom(\mathsf{L})$ denotes the domain of L:

A safety global type $\langle \mathsf{L}, G\rangle^{\ell}$ is well formed if for all $\mathtt{p} \in dom(\mathsf{L})$: $\mathsf{L}(\mathtt{p}) \geq \mathsf{J}(G \upharpoonright \mathtt{p})$.

Henceforth we shall only consider well-formed safety global types.

Typing expressions. The typing judgments for expressions are of the form:

$$\Gamma \vdash e : S^{\ell}$$

where Γ is the *standard environment* which maps variables to sort types with trivial declassification, services to safety global types, and process variables to pairs of sort types with trivial declassification and session types. Formally, we define:

$$\Gamma ::= \emptyset \mid \Gamma, x^{\ell} : S^{\ell'} \mid \Gamma, a^{\ell} : \langle \mathsf{L}, G\rangle^{\ell'} \mid \Gamma, X : S^{\ell}\ T$$

assuming that we can write $\Gamma, x^{\ell} : S^{\ell'}$ (respectively $\Gamma, a^{\ell} : \langle \mathsf{L}, G\rangle^{\ell'}$ and $\Gamma, X : S^{\ell}\ T$) only if x^{ℓ} (respectively a^{ℓ} and X) does not belong to the domain of Γ. An environment Γ is well formed if $x^{\ell} : S^{\ell'} \in \Gamma$ implies $\ell' = \ell$ and $a^{\ell} : \langle \mathsf{L}, G\rangle^{\ell'} \in \Gamma$ implies that $\ell' = \ell$ and G is well formed. Hence, if Γ is well formed, $a^{\ell} : \langle \mathsf{L}, G^{\ell}\rangle \in \Gamma$ implies $\ell = \mathsf{M}(G)$. In the following we will only consider well-formed environments.

We type values by decorating their type with their security level, and names according to Γ:

$$\Gamma \vdash \mathsf{true}^{\ell}, \mathsf{false}^{\ell} : \mathsf{bool}^{\ell} \qquad \Gamma, u : S^{\ell} \vdash u : S^{\ell} \qquad \lfloor \textsc{Name} \rfloor$$

We type expressions by decorating their type with the join of the security levels of the variables and values they are built from.

Typing processes The typing judgments for processes are of the form:

$$\Gamma \vdash_{\ell} P \triangleright \Delta$$

where Δ is the *process environment* which associates session types with channels:

$$\Delta ::= \emptyset \mid \Delta, c : T$$

We decorate the derivation symbol $\vdash$ with the security level ℓ inferred for the process: this level is a lower bound for the actions and communications performed in the process.

Let us now present some selected typing rules for processes.

– Rule $\lfloor$SUBS$\rfloor$ allows the security level inferred for a process to be decreased.

$$\frac{\Gamma \vdash_\ell P \triangleright \Delta \quad \ell' \leq \ell}{\Gamma \vdash_{\ell'} P \triangleright \Delta} \lfloor \textsc{Subs} \rfloor$$

– In rule $\lfloor$MINIT$\rfloor$, the standard environment must associate with the identifier u a safety global type. The premise matches the number of participants in the domain of L with the number declared by the initiator. The emptiness of the process environment in the conclusion specifies that there is no further communication behaviour after the initiator.

$$\frac{dom(\mathsf{L}) = \{1, \ldots, n\}}{\Gamma, u : \langle \mathsf{L}, G \rangle^\ell \vdash_\ell \bar{u}[n] \triangleright \emptyset} \lfloor \textsc{MInit} \rfloor$$

– In rule $\lfloor$MACC$\rfloor$, the standard environment must also associate with u a safety global type. The premise guarantees that the type of the continuation P in the p-th participant is the p-th projection of the global type G of u.

$$\frac{\Gamma, u : \langle \mathsf{L}, G \rangle^\ell \vdash_\ell P \triangleright \Delta, \alpha : G \upharpoonright \mathtt{p}}{\Gamma, u : \langle \mathsf{L}, G \rangle^\ell \vdash_\ell u[\mathtt{p}](\alpha).P \triangleright \Delta} \lfloor \textsc{MAcc} \rfloor$$

Concerning security levels, in rule $\lfloor$MACC$\rfloor$ we check that the continuation process P conforms to the security level ℓ associated with the service name u. Note that this condition does not follow from well-formedness of environments, since the process P may participate in other sessions, but it is necessary to avoid information leaks. For example, without this condition we could type

$$\begin{array}{ll} \bar{a}^\top[2] & \mid\ a^\top[1](\alpha_1).\alpha_1!\langle 2, \mathsf{true}^\top\rangle.\mathbf{0} \mid\ a^\top[2](\alpha_2).\alpha_2?(1, x^\top).\mathsf{if}\ x^\top\ \mathsf{then}\ \bar{b}^\top[2]\ \mathsf{else}\ \mathbf{0} \\ & \mid\ b^\top[1](\beta_1).c^\perp[1](\gamma_1).\gamma_1!\langle 2, \mathsf{true}^\perp\rangle.\mathbf{0} \mid\ b^\top[2](\beta_2).\mathbf{0} \\ \bar{c}^\perp[2] & \mid\ c[2](\gamma_2).\gamma_2?(1, y^\perp).\mathbf{0} \end{array}$$

– In rule $\lfloor$SEND$\rfloor$, the first hypothesis binds expression e with type S^ℓ, where ℓ is the join of all variables and values in e. The second hypothesis imposes typability of the continuation of the output with security level ℓ''. The third hypothesis relates levels ℓ, ℓ'' and ℓ' (the level to which e will be declassified), preserving the invariant that ℓ'' is a lower bound for all security levels of the actions in the process.

$$\frac{\Gamma \vdash e : S^\ell \quad \Gamma \vdash_{\ell''} P \triangleright \Delta, c : T \quad \ell'' \leq \ell' \leq \ell}{\Gamma \vdash_{\ell''} c!^{\ell'}\langle \Pi, e \rangle.P \triangleright \Delta, c : !\langle \Pi, S^{\ell \downarrow \ell'} \rangle; T} \lfloor \textsc{Send} \rfloor$$

Note that the hypothesis $\ell'' \leq \ell' \leq \ell$ is not really constraining, since P can always be downgraded to ℓ'' using rule $\lfloor$SUBS$\rfloor$ and $\ell' \leq \ell$ follows from well-formedness of $S^{\ell \downarrow \ell'}$.

– Rule $\lfloor$RCV$\rfloor$ is the dual of rule $\lfloor$SEND$\rfloor$, but it is more restrictive in that it requires the continuation P to be typable with *exactly* the level ℓ':

$$\frac{\Gamma, x^{\ell'} : S^{\ell'} \vdash_{\ell'} P \triangleright \Delta, c : T \qquad \ell' \leq \ell}{\Gamma \vdash_{\ell'} c?^\ell(\mathtt{p}, x^{\ell'}).P \triangleright \Delta, c : ?(\mathtt{p}, S^{\ell \downarrow \ell'}); T} \lfloor \textsc{Rcv} \rfloor$$

Notice for instance that we cannot type the reception of a $\top$ value followed by a $\perp$ action. On the other hand we can type the reception of a $\top \downarrow \perp$ value followed by a $\perp$ action. For instance, in our introductory example of Section 2, rule [Rcv] allows the

delegation send in process B to be decorated by $\bot$: this is essential for the typability of both the process B and the session b between S and B.
– Rule $\lfloor$IF$\rfloor$ requires that the two branches of a conditional be typed with the same process environment, and with the same security level as the tested expression.

$$\frac{\Gamma \vdash e : \mathsf{bool}^{\ell} \quad \Gamma \vdash_{\ell} P \triangleright \Delta \quad \Gamma \vdash_{\ell} Q \triangleright \Delta}{\Gamma \vdash_{\ell} \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta}\ \lfloor \text{IF} \rfloor$$

We say that a process P is typable in Γ if $\Gamma \vdash_{\ell} P \triangleright \Delta$ holds for some ℓ, Δ.
Typing queues and Q-sets. *Message types* represent the messages contained in queues.

$$\begin{array}{rll} \text{Message } \mathbf{T} ::= & !\langle \Pi, S^{\ell \downarrow \ell'} \rangle & \textit{message value send} \\ | & !^{\ell}\langle \mathsf{q}, T \rangle & \textit{message delegation} \\ | & \oplus^{\ell}\langle \Pi, \lambda \rangle & \textit{message selection} \\ | & \mathbf{T}; \mathbf{T}' & \textit{message sequence} \end{array}$$

Message types are very close to the *send, delsend, selection* session types, hence we shall not dwell on them. Let us just mention the associativity of the construct $\mathbf{T}; \mathbf{T}'$.

Typing judgments for queues have the shape

$$\Gamma \vdash s : h \triangleright \Theta$$

where Θ is a *queue environment* associating message types with channels.
Example: we can derive $\vdash s : (2, \{1,3\}, \mathsf{ok}^{\top}) \triangleright \{s[2] : \oplus^{\top}\langle \{1,3\}, \mathsf{ok} \rangle\}$.

Typing judgments for **Q**-sets have the shape:

$$\Gamma \vdash_{\Sigma} H \triangleright \Theta$$

where Σ is the set of session names which occur free in H.
Typing configurations. Typing judgments for runtime configurations C have the form:

$$\Gamma \vdash_{\Sigma} C \triangleright < \Delta \diamond \Theta >$$

They associate with a configuration the environments Δ and Θ mapping channels to session and message types respectively. We call $< \Delta \diamond \Theta >$ a *configuration environment*.

A *configuration type* is a session type, or a message type, or a message type followed by a session type:

$$\begin{array}{rll} \text{Configuration } \mathscr{T} ::= & T & \textit{session} \\ | & \mathbf{T} & \textit{message} \\ | & \mathbf{T}; T & \textit{continuation} \end{array}$$

An example of configuration type is:

$$\oplus^{\bot}\langle \{1,3\}, \mathsf{ok} \rangle; !\langle \{3\}, \mathsf{String}^{\top} \rangle; ?(3, \mathsf{Number}^{\bot}); \mathsf{end}$$

A configuration is *initial* if the process is closed, it does not use runtime syntax and the **Q**-set is empty. It is easy to check that for typable initial configurations the set of session names and the process and queue environments are all empty.

Since channels with roles occur both in processes and in queues, a configuration environment associates configuration types with channels, in notation $< \Delta \diamond \Theta >(c)$. Configuration types can be projected on a participant p. We write $\mathscr{T} \upharpoonright \mathtt{p}$ to denote the projection of the type $\mathscr{T}$ on the participant p. We also define a duality relation $\bowtie$ between projections of configuration types, which holds when opposite communications are offered (input/output, selection/branching). The above definitions are needed to state coherence of configuration environments. Informally, this holds when the inputs and the branchings offered by the process agree both with the outputs and the selections offered by the process and with the messages in the queues. More formally:

Definition 6. *A configuration environment* $< \Delta \diamond \Theta >$ *is coherent if* $s[\mathtt{p}] \in dom(\Delta) \cup dom(\Theta)$ *and* $s[\mathtt{q}] \in dom(\Delta) \cup dom(\Theta)$ *imply*

$$< \Delta \diamond \Theta > (s[\mathtt{p}]) \upharpoonright \mathtt{q} \bowtie < \Delta \diamond \Theta > (s[\mathtt{q}]) \upharpoonright \mathtt{p}.$$

Typing rules assure that configurations are always typed with coherent environments.

Since process and queue environments represent future communications, by reducing processes we get different configuration environments. This is formalised by the notion of reduction of configuration environments, denoted by $< \Delta \diamond \Theta > \Rightarrow < \Delta' \diamond \Theta' >$.

We say that a queue is *generated by service a*, or *a-generated*, if it is created by applying rule [Link] to the parallel composition of *a*'s participants and initiator.

We are now able to state our main results, namely type preservation under reduction and the soundness of our type system for both access control and noninterference. In Theorem 2, we will use the function $lev_{\uparrow}(\vartheta)$ defined in Section 4 (Definition 1).

Theorem 1 (Subject Reduction). *Suppose* $\Gamma \vdash_{\Sigma} C \triangleright < \Delta \diamond \Theta >$ *and* $C \longrightarrow^* C'$. *Then* $\Gamma \vdash_{\Sigma} C' \triangleright < \Delta' \diamond \Theta' >$ *with* $< \Delta \diamond \Theta > \Rightarrow < \Delta' \diamond \Theta' >$.

Theorem 2 (Access Control).
Let C be an initial configuration, and suppose $\Gamma \vdash_{\emptyset} C \triangleright < \emptyset \diamond \emptyset >$ *for some standard environment* Γ *such that* $a^{\ell} : \langle \mathsf{L}, G \rangle^{\ell} \in \Gamma$. *If* $C \longrightarrow^* (\nu s)C'$, *where the queue of name s in* C' *is a-generated and contains the message* $(\mathtt{p}, \mathtt{q}, \vartheta)$, *then* $lev_{\uparrow}(\vartheta) \leq \mathsf{L}(\mathtt{q})$.

Theorem 3 (Noninterference). *If P is typable, then* $P \simeq_{\mathscr{L}} P$ *for all down-closed* $\mathscr{L}$.

7 Conclusion and future work

In this work, we have investigated the integration of security requirements into session types. Interestingly, there appears to be an influence of session types on security.

For instance, it is well known that one of the causes of insecure information flow in a concurrent scenario is the possibility of different termination behaviours in the branches of a high conditional. In our calculus, we may distinguish three termination behaviours: (proper) termination, deadlock and divergence. Now, the classical session types of [20] already exclude some combinations of these behaviours in conditional branches. For instance, a non-trivial divergence (whose body contains some communication actions) in one branch cannot coexist with a non-trivial termination in the other branch. Moreover, session types prevent *local deadlocks* due to a bad matching of the communication behaviours of participants in the same session. By adding to classical session types the interaction typing of [2], we would also exclude most of the *global deadlocks* due to a bad matching of the protocols of two interleaved sessions. However, this typing does not prevent deadlocks due to inverse session calls. We plan to study a strengthening of interaction typing that would rule out also this kind of deadlock. This would allow us to simplify our type system by removing our constraint in the typing rule for input.

The form of declassification considered in this work is admittedly quite simple. However, it already illustrates the connection between declassification and access control, since a declassified value may only be received by a participant whose level is greater than or equal to the original level of the value. This means that declassification is constrained by the access control policy, as in [5]. We plan to extend declassification

also to data which are not received from another participant, by allowing declassification of a tested expression, as in this variant of the B process of our example in Section 2:

$$\mathtt{B}' \;=\; \ldots \text{ if } \{cc^{\perp} = secret^{\top}\}^{\top\downarrow\perp} \text{ then } \beta_1 \oplus^{\perp} \langle 2, \mathsf{ok}\rangle.\mathbf{0} \text{ else } \beta_1 \oplus^{\perp} \langle 2, \mathsf{ko}\rangle.\mathbf{0}$$

Again, this declassification would be controlled by requiring $\mathtt{B}'$ to have level $\top$.

Acknowledgments We would like to thank Nobuko Yoshida for her encouragement to engage in this work, and the anonymous referees for helpful feedback.

References

1. A. Almeida Matos and G. Boudol. On Declassification and the Non-Disclosure Policy. In *Journal of Computer Security*, volume 17, pages 549–597, 2009.
2. L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *Proc. CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
3. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. In *Proc. CSF'09*, pages 124–140. IEEE Computer Society, 2009.
4. M. Boreale, R. Bruni, R. Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In *Proc. FMOODS '08*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.
5. G. Boudol and M. Kolundzija. Access Control and Declassification. In *Proc. Computer Network Security*, volume 1 of *CCIS*, pages 85–98. Springer, 2007.
6. D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
7. M. Dezani-Ciancaglini and U. de' Liguoro. Sessions and Session Types: an Overview. In *Proc. WSFM'09*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010.
8. R. Focardi and R. Gorrieri. Classification of Security Properties (Part I: Information Flow). In *Proc. FOSAD'00*, volume 2171 of *LNCS*, pages 331–396. Springer, 2001.
9. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.
10. K. Honda and N. Yoshida. A Uniform Type Structure for Secure Information Flow. In *Proc. POPL'02*, pages 81–92. ACM Press, 2002.
11. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *Proc. POPL'08*, pages 273–284. ACM Press, 2008.
12. N. Kobayashi. Type-Based Information Flow Analysis for the Pi-Calculus. *Acta Informatica*, 42(4–5):291–347, 2005.
13. M. Kolundzija. Security Types for Sessions and Pipelines. In *Proc. WSFM'08*, volume 5387 of *LNCS*, pages 175–190. Springer, 2009.
14. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. CUP, 1999.
15. J. Planul, R. Corin, and C. Fournet. Secure Enforcement for Global Process Specifications. In *Proc. CONCUR'09*, volume 5710 of *LNCS*, pages 511–526. Springer, 2009.
16. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
17. A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proc. CSFW'00*, pages 200–214. IEEE Computer Society, 2000.
18. A. Sabelfeld and D. Sands. Dimensions and Principles of Declassification. In *Proc. CSFW'05*. IEEE Computer Society, 2005.
19. G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Proc. POPL'98*, pages 355–364. ACM Press, 1998.
20. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *Proc. PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.
21. D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.