Secure Information Flow as Typed Process Behaviour

Kohei Honda[1], Vasco Vasconcelos[2], and Nobuko Yoshida[3]

[1] Queen Mary and Westfield College, London, U.K.
[2] University of Lisbon, Lisbon, Portugal.
[3] University of Leicester, Leicester, U.K.

Abstract. We propose a new type discipline for the π-calculus in which secure information flow is guaranteed by static type checking. Secrecy levels are assigned to channels and are controlled by subtyping. A behavioural notion of types capturing causality of actions plays an essential role for ensuring safe information flow in diverse interactive behaviours, making the calculus powerful enough to embed known calculi for type-based security. The paper introduces the core part of the calculus, presents its basic syntactic properties, and illustrates its use as a tool for programming language analysis by a sound embedding of a secure multi-threaded imperative calculus of Volpano and Smith. The embedding leads to a practically meaningful extension of their original type discipline.

1 Introduction

In present-day computing environments, a user often employs programs which are sent or fetched from different sites to achieve her/his goals, either privately or in an organisation. Such programs may be run as a code to do a simple calculation task or as interactive parallel programs doing IO operations or communications, and sometimes deal with secret information, such as private data of the user or classified data of the organisation. Similar situations may occur in any computing environments where multiple users share common computing resources. One of the basic concerns in such a context is to ensure programs do not leak sensitive data to the third party, either maliciously or inadvertently. This is one of the key aspects of the security concerns, which is often called *secrecy.* Since it is difficult to dynamically check secrecy at run-time, it may as well be verified statically, i.e. from a program text alone [7]. The *information flow analysis* [7,11,25] addresses this concern by clarifying conditions when flow of information in a program is safe (i.e. high-level information never flows into low-level channels). Recent studies [2,35,33] have shown how we can integrate the techniques of type inference in programming languages with the ideas of information flow analysis, accumulating the basic principles of compositional static verification for secure information flow.

The study of type-based secrecy so far has been done in the context of functional or imperative calculi that incorporate secrecy. Considering that concurrency and communication are a norm in modern programming environments, one may wonder whether a similar study is possible in the framework of process calculi. There are two technical reasons why such an endeavour can be interesting. First, process calculi have been accumulating mathematically rigorous techniques to reason about computation based on communicating processes. In particular, given that an equivalence on program phrases plays a basic role for semantic justification of a type discipline for secrecy [35], the theories of behavioural equivalences [17,20,26,28], which are a cornerstone in the study of process calculi, would offer a semantic basis for safe information flow in communicating processes. Second, type disciplines for communicating processes are widely studied recently, especially in the context of name passing process calculi such as the π-calculus, e.g. [6,15,20,28,32,36]. Further, recent studies have shown that name passing calculi enjoy great descriptive power, uniformly representing diverse language constructs as name passing processes, including those of sequential, concurrent, imperative, functional and object-oriented languages. Since many real-life programming languages are equipped with diverse constructs from different programming paradigms, it would be interesting to see whether we can obtain a typed calculus based on name passing in which information flow involving various language constructs are analysable on a uniform syntactic basis.

Against these backgrounds, the present work introduces a typed π-calculus in which secure information flow is guaranteed by static typing. Secrecy levels are attached to channels, and a simple subtyping ensures that interaction is always secrecy-safe. Information flow in this context arises as transformation of interactive behaviour to another interactive behaviour. Thus the essence of secure information flow becomes that a low-level interaction never depends on a high-level (or incompatible-level) interaction. Interestingly, this interaction-based principle of secure information flow strongly depends on the given type structures as *prerequisites*: that is, even semantically, certain behaviours can become either secure or insecure according to the given types. This is because types restrict a possible set of behaviours (which act as information in the present context), thus affecting the notion of safe information flow itself. For this reason, a strong type discipline for name passing processes for linear and deadlock-free interaction [6,20,36] plays a fundamental role in the present typed calculus, by which we can capture safety of information flow in a wide range of computational behaviours, including those of diverse language constructs. This expressiveness can be used to embed and analyse typed programming languages for secure information flow. In this paper we explore the use of the calculus in this direction through a sound embedding of a secure multi-threaded imperative calculus of Volpano and Smith [33]. The embedding offers an analysis of the original system in which the underlying observable scenario is made explicit and is elucidated by typed process representation. As a result, we obtain a practically meaningful extension of [33] with enlarged typability. We believe this example suggests a general use of the proposed framework, given the fundamental importance of the notion of observables in the analysis of secure computing systems [25,33,34].

Technically speaking, our work follows, on the one hand, Abadi's work on type-based secrecy in the π-calculus [1] and the studies on secure information flow in CCS and CSP [8,24,29,31], and, on the other, the preceding works on type disciplines for name passing processes. In comparison with [1], the main novelty of the present typing system is that it ensures safety of information flow for general process behaviours rather than that for ground values, which is often essential for the embedding of securely typed programming languages. Compared to [8,24,31], a key difference lies in the fundamental role type information plays in the present system for defining and guaranteeing secrecy. Further, these works are not aimed at ensuring secrecy via static typing. Other notable works on the study of security using name passing processes include [3,5]. These works are not about information flow analysis, though they do address other aspects of secrecy.

In the context of type disciplines for name passing processes, the full use of dualised and directed types (cf. §3), as well as their combination with causality-based dynamic types, is new, though the ideas are implicit in [4,10,14,20,36]. Our construction is based on graph-based types in [36], incorporating the partial algebra of types from [15] (the basic idea of modalities used here and in [15] originally comes from linear logic [10]). The syntax of the present calculus is based on [32], among others branching and recursion. We use the synchronous version since it gives a much simpler typing system. The branching and recursion play an essential role in type discipline, as we shall discuss in § 3. The calculus is soundly embeddable into the asynchronous π-calculus (also called the ν-calculus [17]) by concise encoding [32]. The operational feasibility of branching and recursion is further studied in [9,23]. For non-deterministic secrecy in general, security literature offers many studies based on probabilistic non-interference, cf. [13]. The present calculus and its theory are introduced as a basic stratum for the study of secure information flow in typed name passing processes, focussing on a simpler realm of possibilistic settings. Incorporation of the probability distribution in behavioural equivalences [22] is an important subject of future study. Further discussions on related works, including comparisons with functional and imperative secure calculi, are given in the full version [16].

This paper offers a summary of key technical ideas and results, leaving the detailed theoretical development to the full version [16]. In the remainder, Section 2 informally illustrates the basic ideas using examples. Section 3 introduces types, subtyping and the typing rules. Section 4 discusses key syntactic properties of typed terms. Finally Section 5 presents the embedding result and discusses how it suggests an extension of the original type discipline by Volpano and Smith.

Acknowledgement. We deeply thank anonymous referees for their significant comments on an early version. Our thanks also go to Martin Berger, Gavin Lowe, Peter O'Hearn, Edmund Robinson and Pasquale Malacaria for their comments and discussions.

2 Basic Ideas

2.1 A Simple Principle

Let us consider how the notion of information flow arises in interacting processes, taking a simplest example. A CCS term $a.\overline{b}.\mathbf{0}$ represents a behaviour which synchronises at a as input, then synchronises at b as output, and does nothing. Suppose we attach a secrecy level to each port, for example "High" to a and "Low" to b. Intuitively this means that we wish interaction at a to be secret, while interaction at b may be known by a wider public: any high-level security process may interact at a and b, while a low-level security process can interact only at b. Then this process represents insecure interactions: any process observing b, which can be done by a low-level process, has the possibility to know an interaction at a, so information is indeed transmitted to a lower level from a higher level. Note that this does not depend on a being used for input and b used for output: $\overline{a}.b.\mathbf{0}$ with the same assignment of secrecy levels is similarly unsafe. In both cases, we are saying that if there is a causal dependency from an action at a high-level channel to the one at a low-level channel, the behaviour is not safe from the viewpoint of information flow. Further, if we have value passing in addition, we would naturally take dependency in terms of communicated values into consideration.

The above informal principle based on causal dependency[1] is simple, but may look basic as a way of stipulating information flow for processes. Since many language constructs are known to be representable as interacting processes [18,19], one may wonder whether the above idea can be used for understanding safety in information flow in various programming languages. In the following, we consider this question by taking basic examples of information flow in imperative programs.

2.2 Syntax

Let $a, b, c, \ldots x, y, z, \ldots$ range over *names* (which are both points of interaction and values to be communicated), and $X, Y, \ldots$ over *agent variables.* We write $\vec{y}$ for a vector of names $y_0 \cdots y_{n-1}$ with $n \geq 0$. Then the syntax for *processes*, written $P, Q, R, \ldots$, is given by the following grammar. We note that this syntax extends the standard polyadic π-calculus with branching and recursion. These extensions play a fundamental role in the type discipline, in that intended types are hard to deduce if we use their encoding into, say, the polyadic π-calculus (see [16] for further discussions).

$P ::= x(\vec{y}).P$	input	$\mid P \mid Q$	parallel
$\mid\ \overline{x}\langle(\boldsymbol{\nu}\,\vec{z})\vec{y}\rangle.P$	output	$\mid (\boldsymbol{\nu}\,x)P$	hiding
$\mid\ x[(\vec{y}).P \,\&\, (\vec{z}).Q]$	branching input	$\mid \mathbf{0}$	inaction
$\mid\ \overline{x}\,\mathtt{inl}\langle(\boldsymbol{\nu}\,\vec{z})\vec{y}\rangle.P$	left selection	$\mid X\langle\vec{x}\rangle$	recursive variable
$\mid\ \overline{x}\,\mathtt{inr}\langle(\boldsymbol{\nu}\,\vec{z})\vec{y}\rangle.P$	right selection	$\mid (\mu X(\vec{x}).P)\langle\vec{y}\rangle$	recursion

[1] Related ideas are studied in the context of CCS [8] and CSP [31].

There are two kinds of inputs, one unary and another binary: the former is the standard input in the π-calculus, while the latter, the *branching input*, has two branches, waiting for one of them to be selected with associated communication [32]. Accordingly there are outputs with left and right selections, as well as the standard one. We require all vectors of names in round parenthesis are pairwise distinct, which act as binders. In the value part of an output (including selections), say $\langle(\boldsymbol{\nu}\,\vec{z})\vec{y}\rangle$, names in $\vec{z}$ should be such that $\{\vec{z}\} \subset \{\vec{y}\}$ ($\{\vec{x}\}$ is the set of names in $\vec{x}$), and the order of occurrences of names in $\vec{z}$ should be the same as the corresponding names in $\vec{y}$. Here $(\boldsymbol{\nu}\,\vec{z})$ indicate names $\vec{z}$ are new names and are exported by output. $\langle(\boldsymbol{\nu}\,\vec{z})\vec{y}\rangle$ is written $\langle\vec{y}\rangle$ if $\vec{z} = \emptyset$, and $(\boldsymbol{\nu}\,\vec{z})$ if $\vec{y} = \vec{z}$. We often omit vectors of the length zero (for example, we write $\mathtt{inr}$ for $\mathtt{inr}\langle\ \rangle$) as well as the trailing $\mathbf{0}$. The binding and α-convertibility $\equiv_\alpha$ are defined in the standard way. In a recursion $(\mu X(\vec{x}).P)\langle\vec{y}\rangle$, we require that P is *input guarded*, that is P is either a unary input or a branching input, and free names in P are a subset of $\{\vec{x}\}$. The reduction relation $\longrightarrow$ is defined in the standard manner, which we illustrate below (the formal definition is given in [16]).

We illustrate the syntax by examples. First, the following agents represent boolean constants denoting the truth and the conditional selection (let c and y be fresh).

$$\mathbf{T}\langle b\rangle \;=\; b(c).(\overline{c}\,\mathtt{inl}\mid \mathbf{T}\langle b\rangle) \qquad \text{and} \qquad \mathbf{If}\langle x,\ P,\ Q\rangle \stackrel{\text{def}}{=} \overline{x}(\boldsymbol{\nu}\,y).y[().P \,\&\, ().Q]$$

The recursive definition of $\mathbf{T}\langle b\rangle$ is a notational convention and actually stands for $\mathbf{T}\langle b\rangle \stackrel{\text{def}}{=} (\mu X(b).b(c).(\overline{c}\mathtt{inl} \mid X\langle b\rangle))\langle b\rangle$. The truth agent first inputs a name c via b, then, via c, does the left selection with no value passing as well as recreating the original agent. By replacing $\mathtt{inl}$ by $\mathtt{inr}$, we can define the falsity. The conditional process invokes a boolean agent, then waits with two branches. If the other party is truth it generates P: if else it generates Q. We can now show how these two processes interact:

$$\mathbf{If}\langle x,\ P,\ Q\rangle \mid \mathbf{T}\langle x\rangle \;\longrightarrow\; (\boldsymbol{\nu}\,y)(y\,[().P \,\&\, ().Q] \mid \overline{y}\,\mathtt{inl} \mid \mathbf{T}\langle x\rangle) \;\longrightarrow\; P \mid \mathbf{T}\langle x\rangle$$

Next we consider a representation of imperative variable as a process.

$$\mathbf{Var}\langle xv\rangle = x[(z).(\overline{z}\langle v\rangle \mid \mathbf{Var}\langle xv\rangle) \;\&\; (v').\mathbf{Var}\langle xv'\rangle]$$

In this representation, we label the main interaction point of the process (called *principal port* in Interaction Net [21]) by the name of the variable x. It has two branches, of which the left one corresponds to the "read" option, while the right one corresponds to the "write" option. If the "read" is selected and z is received, the process sends the current value v to z, while regenerating the original self. On the other hand, if the "write" branch is selected and v' is received, then the process regenerates itself with a new value v'. We can then consider the representation of the assignment "$x := y$," which first "reads" the value from the variable y, then "writes" that value to the variable x.

$$\mathbf{Assign}\langle xy\rangle \;\stackrel{\text{def}}{=}\; \overline{y}\,\mathtt{inl}(\boldsymbol{\nu}\,z).z(v).\overline{x}\,\mathtt{inr}\langle v\rangle$$

2.3 Imperative Information Flow in Process Representation

(1) Causal Dependency. We can now turn to the information flow. We first consider the process representation of the following obviously insecure code [25].

$$x^{\mathrm{L}} := y^{\mathrm{H}}$$

Here the superscripts "L" and "H" indicate the secrecy levels of variables: thus y is a high (or secret) variable and x is a low (or public) variable. This command is insecure intuitively because the content of a secret variable becomes visible to the public through x. Following the previous discussion, its process representation becomes:

$$\mathbf{Assign}\langle x^{\mathrm{L}} y^{\mathrm{H}}\rangle \overset{\text{def}}{=} \overline{y}^{\mathrm{H}}\,\mathtt{inl}(\boldsymbol{\nu}\, c).c^{\mathrm{H}}(v).\,\overline{x}^{\mathrm{L}}\,\mathtt{inr}\langle v\rangle.\text{ nteractional}$$

Note we are labeling *channels* by secrecy levels. We can easily see that this process violates the informal principle stipulated in §2.1, because its low-level behaviour (at x) depends on its preceding high-level behaviour (at y, c). Thus this example does seem explainable from our general principle. Similarly, we can check the well-known example of implicit insecure flow "$\mathtt{if}\ z^{\mathrm{H}}\ \mathtt{then}\ x^{\mathrm{L}} := y^{\mathrm{L}}\ \mathtt{end}$" (where the information stored in z can be indirectly revealed by reading x), is translated into insecure process interaction "$\overline{z}^{\mathrm{H}}(\boldsymbol{\nu}\, c).c^{\mathrm{H}}[().\mathbf{Assign}\langle x^{\mathrm{L}} y^{\mathrm{L}}\rangle \,\&\, ().\mathbf{0}]$". Here again the low-level interactions (in $\mathbf{Assign}\langle x^{\mathrm{L}} y^{\mathrm{L}}\rangle$) depend on the high-level interactions at z and c.

(2) Deadlock-Freedom. So far there has been no difficulty in applying our general principle to process presentation of imperative information flow. However there are subtleties to be understood, one of which arises in the following sequential composition.

$$x^{\mathrm{H}} := y^{\mathrm{H}}\ ;\ z^{\mathrm{L}} := w^{\mathrm{L}}$$

The whole command is considered to be safe since whatever the content of x and y would be, they do not influence the content of z and w. However the following process representation of this command seems *not* safe in the light of our principle:

$$\overline{y}^{\mathrm{H}}\,\mathtt{inl}(\boldsymbol{\nu}\, c_1).c_1^{\mathrm{H}}(v_1).\,\overline{x}^{\mathrm{H}}\,\mathtt{inr}\langle v_1\rangle.\,\overline{w}^{\mathrm{L}}\,\mathtt{inl}(\boldsymbol{\nu}\, c_2).c_2^{\mathrm{L}}(v_2).\,\overline{z}^{\mathrm{L}}\,\mathtt{inr}\langle v_2\rangle \qquad (\star)$$

Here the behaviours at low-level ports (w and z) depend on, via prefixing, those at high-level ports (x and y). Does this mean our principle and the standard idea in information flow are incompatible with each other? However, a closer look at the above representation reveals that this problematic dependency does not exist in effect, *provided* that the above process interacts with the processes for imperative variables given in §2.2. If we assume so, the actions at y and x (together with those at z and w) by the above process are always enabled: whenever a program wishes to access a variable, it always succeeds (in the i parlance, we are saying that interactions at these names are guaranteed to be *deadlock-free*). Thus we can guarantee that, under the assumption, the action at say w above will surely take place, which means the dependency as expressed in syntax does not exist. Observing there is no dependency at the level of communicated values between the two halves of ($\star$), we can now conclude that the actions at w and z do *not* causally depend on the preceding actions at y and x.

(3) Innocuous Interaction. We now move to another subtle example, using the following command.

$$\texttt{if } z^{\mathrm{H}} \texttt{ then } x^{\mathrm{H}} := y^{\mathrm{L}} \texttt{ end}$$

While this phrase is considered to be secrecy-wise safe [25], its representation in the π-calculus becomes:

$$\overline{z}^{\mathrm{H}}(\boldsymbol{\nu}\, c^{\mathrm{H}}).c[().\overline{y}^{\mathrm{L}}\texttt{inl}(\boldsymbol{\nu}\, e).e^{\mathrm{L}}(v).\overline{x}^{\mathrm{H}}\,\texttt{inr}\langle v\rangle \;\&\; ().\mathbf{0}] \qquad (\star\star)$$

which again shows apparently unsafe dependency between the second action at c and the third action at y. In this example, the process does get information at c in the form of binary selection, even though c is deadlock-free. Moreover the output at y does not occur in the right branch, so the output depends on the action at c even observationally. But the preceding study [33,35] shows the original imperative behaviour is indeed safe. How can it be so? Simple, because this command only *reads* from y, without writing anything: so it is as if it did nothing to y. Returning to ($\star\star$), we find the idea we made resort to in (2), is again effective: we consider this output action as not affecting the environment (hence not transmitting any information) *provided* that the behaviour of the environment is such that invoking its left branch has no real effect – in other words, if it behaves just as the imperative variable given in §2.2 does. We call such an output *innocuous*: thus, if we decide to ignore the effect of innocuous actions, there is no unsafe dependency from the high-level to the low-level (note the left branch as a whole now becomes high-level). We further observe that the insecure examples in (1) are still insecure even after incorporating deadlock-freedom and innocuousness.

The preceding discussions suggest two things: first, we may be able to formally stipulate the interactional framework of safe information flow which may have wide applicability along the line of the informal notion given in §2.1. Secondly, however, just for that purpose, we need a non-trivial notion of types for behaviours which in particular concerns not only the behaviour of the process but also that of the assumed environment. The formal development in the following sections shows how these ideas can be materialised as a typed process calculus for safe information flow.

3 A Typed π-Calculus for Secure Information Flow

3.1 Overview

In addition to *names* and *agent variables* (cf. §2.1), the typed calculus we introduce below uses a set of multiple *secrecy levels*, which are assumed to form a lattice. $s, s', \ldots$ range over secrecy levels, and $s \leq s'$ etc. denotes the partial order (where the lesser means the lower, i.e. more public). Using these data as base sets, our objective in this section is to introduce a typing system whose provable sequent has the following form:

$\Gamma \vdash_s P \triangleright A$ a process P has an action type A under a base Γ with a secrecy level s

We offer an overview of the four elements in the above sequent.

(1) The *base* Γ is a finite function from names and agent variables to types and vectors of types, respectively. Intuitively a type assigned to a channel denotes the basic structure of *possible interaction* at that channel, for example input/output and branching/selection. We also include refined modalities for recursive inputs and their dual outputs, which indicate whether they involve state change or not.

(2) The *process* P is an untyped term in §2.2 which is annotated with types in its bound names, e.g. a unary input becomes $x(\vec{y}:\vec{\alpha}).P$ (here and elsewhere we assume $len(\vec{\alpha}) = len(\vec{y})$ where $len(\vec{y})$ denotes the length of a vector, so that each y_i is assigned a type α_i). As one notable aspect, we only use those processes whose outputs (in any of three forms) are *bound*, e.g. each unary output has a form $\overline{x}(\boldsymbol{\nu}\,\vec{y}:\vec{\alpha}).P$ (this restricted output is an important mode of communication which arises in the context of both π-calculus [30] and games semantics [19,18]). Accordingly we set names in each vector instantiating agent variables to be pairwise distinct. These restrictions make typing rules simpler, while giving enough descriptive power to serve our present purpose.

(3) The *secrecy index* s guarantees that P under Γ only affects the environment at levels at s or higher: that is, it is only transmitting information (or *tampering* the environment) at levels no less than s.

(4) The *action type* A gives abstraction of the causal dependency among (actions on) free channels in P, ensuring, among others, certain deadlock-free properties on its linear and recursive channels. The activation ordering is represented by a partial order on nodes whose typical form is $\mathsf{p}x$ where p denotes a type of action to be done at x. There is a partial algebra over action types [15], by which we can control the composability of two action types (hence of typed processes which own them), thus enabling us to stipulate assumptions on the possible forms of the environments, cf. §2.

3.2 Types and Subtyping

We start with the set of *action modes*, denoted $m, m', \ldots$, whose underlying operational ideas are illustrated by the following table.

$\Downarrow$ non-linear (non-deterministic) input	$\Uparrow$ non-linear (non-deterministic) output
$\downarrow$ truly linear input (truly once)	$\uparrow$ truly linear output (truly once)
$\mathbf{!}$ recursive input (always available)	$\mathbf{?}$ zero or more output (always enabled)

The notations $\mathbf{!}$ and $\mathbf{?}$ come from Linear Logic [10], which first introduced these modalities. We also let $\kappa, \kappa', \ldots$, called *mutability indices*, range over $\{\boldsymbol{\iota}, \boldsymbol{\mu}\}$.

(Well-formedness and Compatibility)

$$\frac{-}{\vdash \tau} \quad \frac{\vdash \tau \asymp \tau'}{\vdash \langle \tau,\ \tau' \rangle} \quad \frac{\vdash \tau \asymp \tau'}{\vdash \tau' \asymp \tau} \quad \frac{\vdash \tau_i \asymp \tau'_i}{\vdash (\vec{\tau})^{\Downarrow}_s \asymp (\vec{\tau}')^{\Uparrow}_s} \quad \frac{\vdash \tau_i \asymp \tau'_i \quad s \geq s'}{\vdash (\vec{\tau})^{\downarrow}_s \asymp (\vec{\tau}')^{\uparrow}_{s'}} \quad \frac{\vdash \tau_i \asymp \tau'_i \quad s \geq s'}{\vdash (\vec{\tau})^{!}_{s,\kappa} \asymp (\vec{\tau}')^{?}_{s',\kappa}}$$

$$\frac{\vdash \tau_{ij} \asymp \tau'_{ij}}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\Downarrow}_s \asymp [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{\Uparrow}_s} \quad \frac{\vdash \tau_{ij} \asymp \tau'_{ij} \quad s \geq s'}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\downarrow}_s \asymp [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{\uparrow}_{s'}} \quad \frac{\vdash \tau_{ij} \asymp \tau'_{ij} \quad s \geq s'}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{!}_{s,\kappa_1 \& \kappa_2} \asymp [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{?}_{s',\kappa_1 \oplus \kappa_2}}$$

(Subtyping)

$$\frac{\vdash \tau_i \leq \tau'_i}{\vdash (\vec{\tau})^{\Downarrow}_s \leq (\vec{\tau}')^{\Downarrow}_s} \quad \frac{\vdash \tau_i \leq \tau'_i \quad s \geq s'}{\vdash (\vec{\tau})^{\downarrow}_s \leq (\vec{\tau}')^{\downarrow}_{s'}} \quad \frac{\vdash \tau_i \leq \tau'_i \quad s \geq s'}{\vdash (\vec{\tau})^{!}_{s,\kappa} \leq (\vec{\tau}')^{!}_{s',\kappa}}$$

$$\frac{\vdash \tau_i \leq \tau'_i}{\vdash (\vec{\tau})^{\Uparrow}_s \leq (\vec{\tau}')^{\Uparrow}_s} \quad \frac{\vdash \tau_i \leq \tau'_i \quad s \leq s'}{\vdash (\vec{\tau})^{\uparrow}_s \leq (\vec{\tau}')^{\uparrow}_{s'}} \quad \frac{\vdash \tau_i \leq \tau'_i \quad s \leq s'}{\vdash (\vec{\tau})^{?}_{s,\kappa} \leq (\vec{\tau}')^{?}_{s',\kappa}}$$

$$\frac{\vdash \tau_{ij} \leq \tau'_{ij}}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\Downarrow}_s \leq [\vec{\tau}'_1 \& \vec{\tau}'_2]^{\Downarrow}_s} \quad \frac{\vdash \tau_{ij} \leq \tau'_{ij} \quad s \geq s'}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\downarrow}_s \leq [\vec{\tau}'_1 \& \vec{\tau}'_2]^{\downarrow}_{s'}} \quad \frac{\vdash \tau_{ij} \leq \tau'_{ij} \quad s \geq s'}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{!}_{s,\kappa_1 \& \kappa_2} \leq [\vec{\tau}'_1 \& \vec{\tau}'_2]^{!}_{s',\kappa_1 \& \kappa_2}}$$

$$\frac{\vdash \tau_{ij} \leq \tau'_{ij}}{\vdash [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\Uparrow}_s \leq [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{\Uparrow}_s} \quad \frac{\vdash \tau_{ij} \leq \tau'_{ij} \quad s \leq s'}{\vdash [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\uparrow}_s \leq [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{\uparrow}_{s'}} \quad \frac{\vdash \tau_{ij} \leq \tau'_{ij} \quad s \leq s'}{\vdash [\vec{\tau}_1 \oplus \vec{\tau}_2]^{?}_{s,\kappa_1 \oplus \kappa_2} \leq [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{?}_{s',\kappa_1 \oplus \kappa_2}}$$

$$\frac{\vdash \langle \tau_1,\ \tau_2 \rangle \quad \vdash \tau \leq \tau_1 \text{ or } \vdash \tau \leq \tau_2}{\vdash \tau \leq \langle \tau_1,\ \tau_2 \rangle} \quad \frac{\vdash \langle \tau'_1,\ \tau'_2 \rangle \quad \vdash \tau_i \leq \tau'_i}{\vdash \langle \tau_1,\ \tau_2 \rangle \leq \langle \tau'_1,\ \tau'_2 \rangle}$$

Fig. 1. Subtyping

Mutability indices indicate whether a recursive behaviour is stateful or not: for input, $\boldsymbol{\iota}$ denotes the lack of state, which we call *innocence*, cf. [19], while $\boldsymbol{\mu}$ means it may be stateful, that is it may change behaviour after invocation; for output, $\boldsymbol{\iota}$ denotes innocuousness, that is the inputting party is innocent, while $\boldsymbol{\mu}$ denotes possible lack of innocuousness. Given these base sets, the grammar of *types*, denoted $\alpha, \beta, \ldots$, are given by:

$$\begin{array}{lcl} \alpha ::= \tau \ \mid \ \langle \tau,\ \tau' \rangle & & \tau \ ::= \ \alpha_{\mathtt{I}} \ \mid \ \alpha_{\mathtt{O}} \\ \alpha_{\mathtt{I}} ::= (\vec{\tau})^{\Downarrow}_s \ \mid \ (\vec{\tau})^{\downarrow}_s \ \mid \ (\vec{\tau})^{!}_{s,\kappa} \ \mid \ [\vec{\tau}_1 \& \vec{\tau}_2]^{\Downarrow}_s \ \mid \ [\vec{\tau}_1 \& \vec{\tau}_2]^{\downarrow}_s \ \mid \ [\vec{\tau}_1 \& \vec{\tau}_2]^{!}_{s,\kappa_1 \& \kappa_2} \\ \alpha_{\mathtt{O}} ::= (\vec{\tau})^{\Uparrow}_s \ \mid \ (\vec{\tau})^{\uparrow}_s \ \mid \ (\vec{\tau})^{?}_{s,\kappa} \ \mid \ [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\Uparrow}_s \ \mid \ [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\uparrow}_s \ \mid \ [\vec{\tau}_1 \oplus \vec{\tau}_2]^{?}_{s,\kappa_1 \oplus \kappa_2} \end{array}$$

Types of form $\langle \tau,\ \tau' \rangle$ are *pair types*, indicating structures of interaction for both input and output, while others are *single types*, which are only for either input or output. We write $\mathsf{md}(\alpha)$ for the set of action modes of the outermost type(s) in α, e.g. $\mathsf{md}((\vec{\tau})^m_s) = \{m\}$ and $\mathsf{md}(\langle (\vec{\tau}_1)^{m_1}_{s_1}, (\vec{\tau}_2)^{m_2}_{s_2} \rangle) = \{m_1, m_2\}$. We often write $\mathsf{md}(\alpha) = m$ for $\mathsf{md}(\alpha) = \{m\}$. Similarly, we write $\mathsf{sec}(\tau)$ for the security level of the outermost type in τ, e.g. $\mathsf{sec}((\vec{\tau})^m_s) = s$. We define the *dual* of m, written

$\overline{m}$, as: $\overline{\Downarrow} = \Uparrow$, $\overline{\Uparrow} = \Downarrow$, $\overline{\uparrow} = \downarrow$, $\overline{\downarrow} = \uparrow$, $\overline{!} = ?$ and $\overline{?} = !$. Then the dual of a type α, denoted by $\overline{\alpha}$, is given by inductively dualising each action mode in α, as well as exchanging & and $\otimes$. Among types, those with body $(\vec{\tau})$ correspond to unary input/output, those with body $[\vec{\tau}_1 \& \vec{\tau}_2]$ correspond to branching input, and those with body $[\vec{\tau}_1 \oplus \vec{\tau}_2]$ correspond to output with selections.

We say α is *well-formed*, written $\vdash \alpha$, if it is derivable from the rules in Figure 1, where we also define the *compatibility relation* $\asymp$ over single types. A pair type is well-formed iff its constituting single types are compatible. We also say α is a *subtype of* β, denoted $\vdash \alpha \leq \beta$, if this sequent is derivable by the rules in Figure 1. Some comments on types, subtyping and compatibility follow.

Remark 1. **(nested types)** Nested types denote what the process would do after exporting or importing new channels (hence covariance of subtyping on nested types): as an example, neglecting the secrecy and mutability, $x : (()^{\downarrow})^{\uparrow}$ denotes the behaviour of doing a truly linear output at x exporting one single new name, and at that name doing a truly linear input without importing any name.

(secrecy levels, compatibility and subtyping) Since safe information flow should never go from a higher level to a lower level, a rule of thumb is that two types are compatible if such a flow is impossible. Thus, because a flow can occur in both ways at non-deterministic channels (cf. §2.1), two non-linear types can be related only when they have the same secrecy level. On the other hand, for compatibility of linear types, we require that the inputting side is higher than the outputting side in secrecy levels, since the flow never comes from the inputting party (further, in truly linear unary types, even the outputting party does not induce flow). Accordingly, the subtyping is covariant for output and contravariant for input with respect to secrecy levels.

(mutability index) As we explained already, the index ι represents the recursive input behaviour without state change (innocence) or, dually, the output which does not tamper the corresponding recursive processes (innocuousness). Note an index is only meaningful for recursive behaviours and their dual output. Naturally we stipulate that an innocent input can only be compatible with an innocuous output; and an innocent input can only be a subtype of an innocent input, and an innocuous output can only be a subtype of an innocuous output.

3.3 Action Types

An *action type A* is a finite poset whose elements, called *action nodes*, are given by the following grammar.

$$\mathbf{n} \ ::= \ \downarrow x \ \mid \ \uparrow x \ \mid \ \updownarrow x \ \mid \ !x \ \mid \ ?x \ \mid \ ?^{\iota}x \ \mid \ \Updownarrow x \ \mid \ X\langle \vec{x} \rangle.$$

$\updownarrow x$ indicates x is already used exactly once for both input and output. $?^{\iota}x$ indicates that all actions occurring at x so far are innocuous. $X\langle \vec{x} \rangle$ (with $len(\vec{x}) \geq 1$ always) indicates the point to which the behaviour recurs. $\Updownarrow$ indicates possibility of nonlinear (nondeterministic) input and output. Other symbols are already explained in the table in §3.2. As an illustration of causality, write $\mathbf{n} \to \mathbf{n}'$ when $\mathbf{n}'$ is strictly bigger than $\mathbf{n}$ without any intermediate element. Then $\downarrow x \to \uparrow y$ says that a truly linear output at y becomes active just after a truly linear input at x.

We only use those action types which conform to a well-formedness condition that in particular includes linearity (for details see [16]). In the typing rules, we use the following abbreviations for action types (let $\{x_i\}$ be free names in A).

$\downarrow\uparrow A$	A only contains $\downarrow x_i$ or $\uparrow x_i$	A^{-x}	x does not occur in A
$\mathbf{?}A$	A only contains $\mathbf{?}x_i$, $\mathbf{?}^{\iota}x_i$ or $\Updownarrow x_i$	$A \otimes B$	disjoint union, with $A \cap B = \emptyset$
$\mathbf{?}^{\iota}A$	A only contains $\mathbf{?}^{\iota}x_i$	$\vec{\mathsf{p}x}$	$\mathsf{p}_0x_0 \otimes \mathsf{p}_1x_1 \cdots \mathsf{p}_{n-1}x_{n-1}$ $(n \geq 0)$

We also say x is *active in* A if $\mathsf{p}x$ (for some p) is minimal in A.

3.4 Typing System

We now introduce the main typing rules with illustration. We use the following notation: given a base Γ, (1) $x : \alpha$ (resp. $X : \vec{\alpha}$) denotes $\Gamma(x) = \alpha$ (resp. $\Gamma(X) = \vec{\alpha}$); and (2) $\Gamma \cdot \Delta$ denotes the disjoint union of two bases, assuming their domains do not intersect. Henceforth we assume all types and bases are well-formed. We start from the typing rules for basic process operators: the inaction, parallel composition and hiding.

$$\text{(Zero)}\quad \frac{}{\Gamma \vdash_s \mathbf{0} \triangleright \emptyset} \qquad \text{(Par)}\quad \frac{A_1 \asymp A_2 \quad \Gamma \vdash_s P_i \triangleright A_i \quad (i = 1,2)}{\Gamma \vdash_s P_1 \mid P_2 \triangleright A_1 \odot A_2} \qquad \text{(Res)}\quad \frac{\Gamma \cdot x : \alpha \vdash_s P \triangleright A \otimes \mathsf{p}x \quad \mathsf{p} \in \{\updownarrow, \mathbf{!}, \Updownarrow\}}{\Gamma \vdash_s (\boldsymbol{\nu}\, x : \alpha)P \triangleright A}$$

In (Par), we use *coherence* $A_1 \asymp A_2$ and *composition* $A_1 \odot A_2$, both following [36]. Essentially speaking, $A_1 \asymp A_2$ says A_1 and A_2 are composable without violating linearity or causing vicious circles; then $A_1 \odot A_2$ is the result of the composition. See [16] for details. In (Res), we do not allow a name with a mode in $\{\downarrow, \uparrow, \mathbf{?}, \mathbf{?}^{\iota}\}$ to be restricted since these actions expect their complementary actions to get composed — in other words, actions with these types assume the existence of actions with their dual types in the environment. With the complementary actions left uncomposed, the hiding leads to an insecure system. In addition, we have the weakening rules for $\mathbf{?}x$, $\mathbf{?}^{\iota}x$, $\updownarrow x$ and $\Updownarrow x$, and the *degradation rule* in which $\Gamma \vdash_s P \triangleright A$ is degraded into $\Gamma \vdash_{s'} P \triangleright A$ when $s' \leq s$ (cf. § 3.1 (3)).

We next turn to non-liner prefix rules. The rules for prefix actually control the secrecy levels of each action.

$$\text{(In)}\quad \frac{\vdash (\vec{\tau})^{\Downarrow}_{s} \leq \Gamma(x) \quad \Gamma \cdot \vec{y}{:}\vec{\tau} \vdash_s P \triangleright \overrightarrow{\mathsf{p}y} \otimes \mathbf{?}A \otimes \Updownarrow x}{\Gamma \vdash_s x(\vec{y}{:}\vec{\tau}).P \triangleright A \otimes \Updownarrow x} \qquad \text{(Out)}\quad \frac{\vdash (\vec{\tau})^{\Uparrow}_{s} \leq \Gamma(x) \quad \Gamma \cdot \vec{y}{:}\vec{\tau} \vdash_s P \triangleright \overrightarrow{\mathsf{p}y} \otimes \mathbf{?}A \otimes \Updownarrow x}{\Gamma \vdash_s \overline{x}(\boldsymbol{\nu}\, \vec{y}{:}\vec{\alpha}).P \triangleright A \otimes \Updownarrow x}$$

Since the subtyping on non-linear types is trivial with respect to their secrecy levels, $\vdash (\vec{\tau})^{\Uparrow,s} \leq \Gamma(x)$ means $\Gamma(x)$ has precisely the level s. Thus, in both rules, the initial action at level s is followed by actions affecting the same or higher levels (because P is typed with s). Note also all abstracted actions ($\overrightarrow{\mathtt{p}y}$ above) should be active, which is essential for the subject reduction. Non-linear prefix rules for branching and selections are essentially the same.

Among linear prefix rules, the following shows a stark contrast with the non-linear (In) and (Out) rules.

$$\dfrac{\begin{array}{l}(\mathsf{In}^{\downarrow})\quad (\text{where } C/\vec{y} = \downarrow\uparrow B)\\ \vdash (\vec{\tau})^{\downarrow}_{s'} \leq \Gamma(x)\\ \Gamma\cdot\vec{y}{:}\vec{\tau} \vdash_s P \triangleright \mathbf{?}A \otimes C^{\text{-}x}\end{array}}{\Gamma \vdash_s x(\vec{y}{:}\vec{\tau}).P \triangleright A \otimes \downarrow x \to B} \qquad \dfrac{\begin{array}{l}(\mathsf{Out}^{\uparrow})\quad (\text{where } C/\vec{y} = \downarrow\uparrow B)\\ \vdash (\vec{\tau})^{\uparrow}_{s'} \leq \Gamma(x)\\ \Gamma\cdot\vec{y}{:}\vec{\tau} \vdash_s P \triangleright \mathbf{?}A \otimes C^{\text{-}x}\end{array}}{\Gamma \vdash_s \overline{x}(\boldsymbol{\nu}\,\vec{y}{:}\vec{\tau}).P \triangleright A \otimes \uparrow x \to B}$$

The notation $C/\vec{y}$ denotes the result of taking off nodes with names among $\vec{y}$, as well as stipulating the condition that each y_i should be active in C. We observe that the "true linearity" in these and later rules is stronger than those studied in [15,20], which only requires "no more than once". In the rule, since s' is not given any condition in the antecedent, both rules completely neglect the secrecy level of x in Γ, saying we may not regard these actions as either receiving or giving information from/to the environment. The operation $\mathbf{n}\to B$, which is given in [16] following [36], records the causality.

The next rules show that branching/selection need a different treatment from the unary cases when types are truly linear. Intuitively, the act of selection gives rise to a non-trivial flow of information.

$$\dfrac{\begin{array}{l}(\mathsf{Bra}^{\downarrow})\quad (\text{where } C_i/\vec{y}_i = \downarrow\uparrow B)\\ \vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\downarrow}_{s} \leq \Gamma(x)\\ \Gamma\cdot\vec{y}_i{:}\vec{\tau}_i \vdash_s P_i \triangleright \mathbf{?}A \otimes C_i^{\text{-}x} \quad (i=1,2)\end{array}}{\Gamma \vdash_s x[(\vec{y}_1{:}\vec{\tau}_1).P_1 \,\&\, (\vec{y}_2{:}\vec{\tau}_2).P_2] \triangleright A \otimes \downarrow x \to B} \qquad \dfrac{\begin{array}{l}(\mathsf{Sel}_l^{\uparrow})\quad (\text{where } C/\vec{y}_1 = \downarrow\uparrow B)\\ \vdash [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\uparrow}_{s} \leq \Gamma(x)\\ \Gamma\cdot\vec{y}_1{:}\vec{\tau}_1 \vdash_s P \triangleright \mathbf{?}A \otimes C^{\text{-}x}\end{array}}{\Gamma \vdash_s \overline{x}\mathtt{inl}(\boldsymbol{\nu}\,\vec{y}_1{:}\vec{\tau}_1).P \triangleright A \otimes \uparrow x \to B}$$

Here the subtyping is used non-trivially: in (Bra$^{\downarrow}$), the real level of x in Γ is the same or lower than s, so the level elevates. In (Sel$^{\uparrow}$, the real level of x is the same or higher, so the level may go down, but it is recorded in the conclusion. It is notable that this inference crucially depends on the employment of branching as a syntactic construct: without it, these rules should have the same strict conditions as non-linear prefixes.

The final class of rules show the treatment of **!**-**?** modalities and mutability indices, dealing with recursive inputs and their dual outputs, and are most involved. We first have the variable introduction rule (Var$^{!}$), in which we derive $\Gamma \cdot X : \vec{\alpha} \vdash_s X\langle\vec{x}\rangle \triangleright X\langle\vec{x}\rangle$ when we have both $\vdash \alpha_i \leq \Gamma(x_i)$ and $\mathsf{md}(\alpha_0) = \mathbf{!}$, as well as (for consistency with repetitive invocation) $\mathsf{md}(\alpha_i) \in \{\mathbf{?}, \Downarrow, \Uparrow\}$ $(i \neq 0)$. Here we give no restriction on s since when the introduced variable is later bound, all potential tampering at free names would have been recorded except the subject of this recursion, the latter not being tampering. Below we introduce linear recursion rules, for which there are two pairs, one for unary prefix and another for binary prefix. We show the rules for unary input/output.

$$\textsf{(In}^{\mathbf{!}}\textsf{)}\quad \frac{\begin{array}{l}\vdash (\vec{\tau})^{\mathbf{!}}_{s,\kappa} \leq \Gamma(z_0) \quad \vdash \alpha_i \leq \Gamma(z_i) \\ \begin{cases}\Gamma\{\vec{x}/\vec{z}\}\cdot\vec{y}:\vec{\tau}\cdot X:\vec{\alpha} \vdash_s P \triangleright \overrightarrow{\mathtt{p}y}\otimes\mathbf{?}^{\boldsymbol{\iota}}A\{\vec{x}/\vec{z}\}\otimes X\langle\vec{x}\rangle\ (\kappa=\boldsymbol{\iota}) \\ \Gamma\{\vec{x}/\vec{z}\}\cdot\vec{y}:\vec{\tau}\cdot X:\vec{\alpha} \vdash_s P \triangleright \overrightarrow{\mathtt{p}y}\otimes\mathbf{?}A\{\vec{x}/\vec{z}\}\otimes X\langle x\vec{w}\rangle(\kappa=\boldsymbol{\mu})\end{cases}\end{array}}{\Gamma \vdash_s (\mu X(\vec{x}:\vec{\alpha}).x_0(\vec{y}:\vec{\tau}).P)\langle\vec{z}\rangle \triangleright \mathbf{!}z_0 \otimes A}$$

$$\textsf{(Out}^{\mathbf{?}}\textsf{)}\ \ (\text{where } C/\vec{y} = \downarrow\!\uparrow B)\quad \frac{\begin{array}{l}\vdash (\vec{\tau})^{\mathbf{?}}_{s',\kappa} \leq \Gamma(x) \quad \mathtt{p} \in \{\mathbf{?}, \mathbf{?}^{\boldsymbol{\iota}}\} \\ \Gamma\cdot\vec{y}{:}\vec{\tau} \vdash_s P \triangleright \mathbf{?}A\otimes C\otimes\mathtt{p}x \\ \kappa=\boldsymbol{\mu} \Rightarrow (s=s' \wedge \mathtt{p}=\mathbf{?})\end{array}}{\Gamma \vdash s \triangleright \overline{x}(\boldsymbol{\nu}\,\vec{y}:\vec{\tau}).PA\otimes B\otimes\mathtt{p}x}$$

In (In$^{\mathbf{!}}$), we check that the process is immediately recurring to precisely the same behaviour ($X\langle\vec{x}\rangle$) if it is innocent, or, if it is not innocent, it recurs to the same subject ($X\langle x_0\vec{w}_j\rangle$). The process can only do free actions with $\mathbf{?}^{\boldsymbol{\iota}}$-modes in the innocent case in addition to the recurrence (except at $\vec{y}$, which are immediately abstracted), so that the process is stateless in its entire visible actions. In the conclusion, the new subject z_0 is introduced with the mode $\mathbf{!}$. In the dual (Out$^{\mathbf{?}}$), if the prefix is an innocuous output ($\kappa = \boldsymbol{\iota}$), there is no condition on the level of x (s'), so that the level is *not* counted either in the antecedent or in the conclusion (e.g. even if $s' = \bot$ we can have $s \neq \bot$): we are regarding the action as not affecting, and not being affected by, the environment. However if the action is *not* innocuous ($\kappa = \boldsymbol{\mu}$), it is considered as affecting the environment, so that we record its secrecy level by requiring $s' = s$. Note that, even if it is unary, a $\mathbf{?}$-mode output action may indeed affect the environment simply because such an action may or may not exist: just as a unary non-deterministic input/output induces information flow. The corresponding rules for the branching and selection are defined in the same way, see [16].

3.5 Examples of Typing

(Non-linear) Let $\mathtt{sync}^{\Downarrow}_s \stackrel{\text{def}}{=} ()^{\Downarrow}_s$. Then $a:\overline{\mathtt{sync}^{\Downarrow}_{s'}} \cdot b:\mathtt{sync}^{\Downarrow}_s \vdash_{s'} \overline{a}.b \triangleright \Updownarrow a \otimes \Updownarrow b$, for $s' \leq s$.

(Truly linear) Let $\mathtt{sync}^{\downarrow}_s \stackrel{\text{def}}{=} ()^{\downarrow}_s$, and its dual $\mathtt{sync}^{\uparrow}_s \stackrel{\text{def}}{=} ()^{\uparrow}_s$. Then, for arbitrary s and s', we have $a:\mathtt{sync}^{\uparrow}_s \cdot b:\mathtt{sync}^{\downarrow}_{s'} \vdash_{\top} \overline{a}.b \triangleright \uparrow a \rightarrow \downarrow b$.

(Branching) Let $\mathtt{bool}^{\mathbf{!}}_s \stackrel{\text{def}}{=} ([\oplus]^{\uparrow}_s)^{\mathbf{!}}_s$ be the type of a boolean constant. Then we have $b:\mathtt{bool}^{\mathbf{!}}_s \vdash_s \mathbf{T}\langle b\rangle \triangleright \mathbf{!}b$. For the conditional $\mathbf{If}\langle b, P_1, P_2\rangle$ introduced in §2, suppose that the two branches P_1 and P_2 can be typed at a security level above that of the boolean constant b; that is, P_i is such that $\Gamma\cdot b:\mathtt{bool}^{\mathbf{?}}_{s'} \vdash_s P_i \triangleright \mathbf{?}A\otimes\mathbf{?}^{\boldsymbol{\iota}}b$, for $s' \leq s$. Then $\Gamma\cdot b:\mathtt{bool}^{\mathbf{?}}_{s'} \vdash_s \mathbf{If}\langle b, P_1, P_2\rangle \triangleright A\otimes\mathbf{?}^{\boldsymbol{\iota}}b$. The innocuousness at b is crucial to show that $(\mathtt{bool}^{\mathbf{!}}_{s'})^{\mathbf{?}}_{\top} \leq (\mathtt{bool}^{\mathbf{!}}_{s'})^{\mathbf{?}}_{s'}$ in rule $\textsf{Out}^{\mathbf{?}}$.

(Copy-cat) The following agent concisely represents the idea of safe information flow in the present calculus. It also serves as a substitute for free name passing for various purposes, including the imperative variable below.

$$[b^s \leftarrow b'^{s'}] \ = \ b(c:\mathtt{bool}^{\uparrow}_s).(\mathbf{If}\langle b', \overline{c}\,\mathtt{inl}, \overline{c}\,\mathtt{inr}\rangle \mid [b \leftarrow b'])$$

This agent transforms a boolean behaviour from b' to b. If $s' \leq s$, then we have: $b:\mathtt{bool}^{\mathbf{!}}_s, b':\mathtt{bool}^{\mathbf{?}}_{s'} \vdash_s [b \leftarrow b'] \triangleright \mathbf{!}b\otimes\mathbf{?}^{\boldsymbol{\iota}}b'$.

(Imperative variable) We give a representation of an imperative variable, alternative to that presented in §2.

$$\mathbf{Var}\langle x^s b^{s'}\rangle^s = x[(z\,{:}\,(\mathtt{bool}^{!}_{s})^{\uparrow}_{s}).(\overline{z}(\boldsymbol{\nu}\, b'\,{:}\,\mathtt{bool}^{?}_{s}).[b' \leftarrow b]|\mathbf{Var}\langle xb\rangle) \;\&\; (b'\,{:}\,\mathtt{bool}^{?}_{s}).\mathbf{Var}\langle xb'\rangle]$$

By the copy-cat, sending a new b' has the same effect as sending b. To type this process, let $\mathtt{var}^{!}_{s} \stackrel{\text{def}}{=} [(\mathtt{bool}^{!}_{s})^{\uparrow}_{s}\&\mathtt{bool}^{?}_{s}]^{!}_{s,\boldsymbol{\iota}\&\boldsymbol{\mu}}$. Then $x : \mathtt{var}^{!}_{s}, b : \mathtt{bool}^{!}_{s'} \vdash_s \mathbf{Var}\langle x^s b\rangle \triangleright \mathbf{!}x \otimes \mathbf{?}^{\boldsymbol{\iota}} b$ for $s' \leq s$. Note b has the level s' but the secrecy index is still s, since at b the output is innocuous.

(Assignment) The following offers the typing of the behaviour representing $x^{\mathrm{H}} := y^{\mathrm{L}}$. Let $\mathtt{var}^{?}_{s} \stackrel{\text{def}}{=} \overline{\mathtt{var}^{!}_{s}}$ and $\Gamma = x\,{:}\,\mathtt{var}^{?}_{\mathrm{H}} \cdot y\,{:}\,\mathtt{bool}^{!}_{\mathrm{L}}$. Then

$$\Gamma \vdash_{\mathrm{H}} \overline{y}\mathtt{inl}(z\,{:}\,(\mathtt{bool}^{?}_{\mathrm{L}})^{\downarrow}_{\mathrm{L}}).z(b\,{:}\,\mathtt{bool}^{?}_{\mathrm{H}}).\overline{x}\mathtt{inr}(b'\,{:}\,\mathtt{bool}^{!}_{\mathrm{H}}).[b' \leftarrow b] \triangleright \mathbf{?}x \otimes \mathbf{?}^{\boldsymbol{\iota}} y.$$

4 Elementary Properties of Typed Processes

This section presents the most basic syntactic properties of typed terms. We also briefly discuss one key behavioural property typed terms enjoy. First, the typing system satisfies the standard properties as weakening, strengthening and substitution closure. We only list two important properties. Below (1) says that every typable term has a *canonical typing*, i.e. whenever P is typable, P has the minimum action type and the highest secrecy index, and (2) means that channel types in Γ represent the constraints on the behaviour of P, rather than that of the outside environment (below $A \leq_{\mathtt{G}} A'$ iff $A = A'_0 \otimes \overrightarrow{\mathbf{?}^{\boldsymbol{\iota}} x}$ and $A' = A'_0 \otimes \overrightarrow{\mathbf{?}x} \otimes \overrightarrow{\updownarrow y} \otimes \overrightarrow{\Updownarrow w}$ for some A'_0).

Proposition 1. (1) (canonical typing) *If $\Gamma \vdash_s P \triangleright A$, then there exists s_0 and A_0 such that $\Gamma \vdash_{s_0} P \triangleright A_0$, and whenever $\Gamma \vdash_{s_1} P \triangleright A_1$ we have $s_1 \leq s_0$ and $A_0 \leq_{\mathtt{G}} A_1$.*

(2) (subsumption-narrowing) *If $\Gamma \cdot x\,{:}\,\alpha \vdash_s P \triangleright A$ and $\alpha \leq \alpha'$, then $\Gamma \cdot x\,{:}\,\alpha' \vdash_s P \triangleright A$.*
Also if $\Gamma \cdot X\,{:}\,\vec{\alpha} \vdash_s P \triangleright A$ and $\alpha_i \geq \beta_i$ for each i, then $\Gamma \cdot X\,{:}\,\vec{\beta} \vdash_s P \triangleright A$.

A fundamental property of the typing system follows. Below $\twoheadrightarrow$ is the multi-step reduction over preterms, defined just as that over untyped terms.

Theorem 1. (subject reduction) *If $\Gamma \vdash_s P \triangleright A$ and $P \twoheadrightarrow Q$ with $\mathsf{bn}(Q) \cap \mathsf{fn}(\Gamma) = \emptyset$, then $\Gamma \vdash_s Q \triangleright A$.*

The theorem says that whatever internal reduction takes place, its composability with the outside, which is controlled by both Γ and A, does not change; and that, moreover, the process is still secure with a no less secrecy index. For the proof, see [16].

The subject reduction is the basis of various significant behavioural properties for typed processes. Here we discuss only one of them, a non-interference property in typed terms (cf. [1,11,25]). A *$\langle\Gamma{\cdot}s{\cdot}A\rangle$-context* is a typed context whose hole is typed under the triple $\langle \Gamma, s, A\rangle$. Then, with respect to security level s, we can define the *s-sensitive maximum sound typed congruence* (cf. [17,28,36]), denoted $\cong_s$, following the standard construction (see [16] for the full definition). We then obtain:

> (**behavioural non-interference**) Let $C[\cdot]$ be a $\langle \Gamma_0 \cdot s_0 \cdot A_0\rangle$-context. If $s \lneq s_0$ and $\Gamma_0 \vdash_{s_0} P_i \triangleright A_0$ $(i = 1, 2)$, then $C[P_1] \cong_s C[P_2]$.

The statement says that the behaviour of the whole at lower levels are never affected by its constituting behaviours which only act at higher levels. The proof uses a secrecy-sensitive version of typed bisimilarity, which is a fundamental element of the present theory and which turns out to be a subcongruence of the above maximum sound equality at each secrecy level. By noting ground constants are representable as constant behaviours, one may say the result extends Abadi's non-interference result for ground values [1] to typed process behaviours.

5 Imperative Information Flow as Typed Process Behaviour

5.1 A Multi-Threaded Imperative Calculus

Smith and Volpano [33] presented a type discipline for a basic multi-threaded imperative calculus in which well-typedness ensures secure information flow. In this section we show how the original system can be embedded in the typed calculus introduced in this paper, with a suggestion for a practically interesting extension of the original type discipline through the analysis of the notion of observables. We start with the syntax of untyped phrases of the original calculus, using $x, y, z, \ldots$ for imperative variables.

$$e \ ::= \ x \ \mid \ \mathtt{b} \ \mid \ e_1 \ \mathtt{and} \ e_2 \qquad\qquad \mathtt{b} \ ::= \ \mathtt{tt} \ \mid \ \mathtt{ff}$$
$$c \ ::= \ x := e \mid c_1 ; c_2 \mid c_1 \mid c_2 \mid \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ e\ \mathtt{do}\ c \mid \mathtt{skip}$$

For simplicity we restrict data types to booleans. We also added the `skip` command, and use the parallel composition rather than a system of threads.

The typing system is given in Figure 2. It uses *command types* of form

$$\rho \ ::= \ s\,\mathtt{cmd}\Downarrow \ \mid \ s\,\mathtt{cmd}\Uparrow.$$

Here $s\,\mathtt{cmd}\Downarrow$ (resp. $s\,\mathtt{cmd}\Uparrow$) indicates convergent (resp. divergent) phrases and $s, s', \ldots$ are secrecy levels as before. Note we take secrecy levels from an arbitrary lattice rather than from the two point one. We also use a base E, which is a finite map from variables to secrecy levels. Subsumption in expressions is merged into their typing rules for simplicity. Notice the contravariance in the first two subtyping rules [33,35] and the invariance in the last rule. The types in the original system are embedded into the command types above by setting:

$$(\textsc{h})^\circ \stackrel{\text{def}}{=} \top \qquad (\textsc{l})^\circ \stackrel{\text{def}}{=} \bot \qquad (\textsc{h}\ \mathtt{cmd})^\circ \stackrel{\text{def}}{=} \top\,\mathtt{cmd}\Downarrow \qquad (\textsc{l}\ \mathtt{cmd})^\circ \stackrel{\text{def}}{=} \bot\,\mathtt{cmd}\Uparrow,$$

(Subtyping) $\dfrac{s' \leq s}{s\ \mathtt{cmd}\Downarrow\ \leq s'\ \mathtt{cmd}\Downarrow}$ $\quad\dfrac{s' \leq s}{s\ \mathtt{cmd}\Downarrow\ \leq s'\ \mathtt{cmd}\Uparrow}$ $\quad s\ \mathtt{cmd}\Uparrow\ \leq s\ \mathtt{cmd}\Uparrow$

(Typing)

(var) $\dfrac{E(x) \leq s}{E \vdash x : s}$ (bool) $E \vdash b : s$ (and) $\dfrac{E \vdash e_i : s \quad (\mathrm{i} = 1{,}2)}{\vdash e_1\ \mathtt{and}\ e_2 : s}$

(skip) $E \vdash \mathtt{skip} : s\ \mathtt{cmd}\Downarrow$ (subs) $\dfrac{E \vdash c : \rho \quad \rho \leq \rho'}{E \vdash c : \rho'}$ (compose) $\dfrac{E \vdash c_i : \rho}{E \vdash c_1; c_2 : \rho}$ (parallel) $\dfrac{E \vdash c_i : \rho}{E \vdash c_1 \mid c_2 : \rho}$

(assign) $\dfrac{E \vdash e : s \quad E(x) = s}{E \vdash x := e : s\ \mathtt{cmd}\Downarrow}$ (if) $\dfrac{E \vdash e : \mathsf{sec}(\rho) \qquad E \vdash c_i : \rho}{E \vdash \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \rho}$ (while) $\dfrac{E \vdash e : \bot \quad E \vdash c : \bot\ \mathtt{cmd}\Uparrow}{E \vdash \mathtt{while}\ e\ \mathtt{do}\ c : \bot\ \mathtt{cmd}\Uparrow}$

Fig. 2. Typing System of Smith-Volpano calculus

which makes explicit the notion of termination in the original types. With this mapping, the present system is a conservative extension of the original one in both subtyping judgement and typability.

5.2 Embedding

We start with the embedding of types and bases, given in Figure 3. Both command types and bases are translated into two forms, one using channel types and the other using action types. In $[\![\rho]\!]$, a terminating type becomes a truly linear synchronisation type, and a non-terminating type becomes a non-linear synchronisation type, both described in §3.5. $\langle\!\langle \rho \rangle\!\rangle_f$ gives an action type accordingly. We note: given command types ρ, ρ', $\rho \leq \rho'$ iff either (1) $\mathsf{sec}([\![\rho]\!]) \geq \mathsf{sec}([\![\rho']\!])$ and both are truly linear unary, (2) $\mathsf{sec}([\![\rho]\!]) \geq \mathsf{sec}([\![\rho']\!])$, $[\![\rho]\!]$ is truly linear unary and $[\![\rho']\!]$ is nonlinear, or (3) $[\![\rho]\!] = [\![\rho']\!]$ and both are nonlinear. This dissects command types into (a) the secrecy level of the whole behaviour (which guarantees the lowest tampering level and which can be degraded by the degradation rule) and (b) the nature of the termination behaviour (noting "non-linear" means a termination action is not guaranteed).

We next turn to the embedding of terms into processes in Figure 3. The framework assumes two boolean constant agents whose behaviours are given in §2.2 and which are shared by all processes, with principal channels *tt* and *ff*. These free channels are given the $\bot$-level, which is in accordance with Smith and Volpano's idea that constants have no secrecy. Following the translation of types, each command becomes a process that upon termination emits an output signal at a channel given as a parameter, typically f (cf. [26]). We are using copy-cat in §3.5 to represent the functionality of value passing. The encoding of terms should be easily understandable, following the known treatment as in [26]: the

(Type and Base)

$$[\![s\ \mathtt{cmd}\Downarrow]\!] \overset{\text{def}}{=} \mathtt{sync}^{\uparrow}_{s} \quad \langle\!\langle s\ \mathtt{cmd}\Downarrow\rangle\!\rangle_f \overset{\text{def}}{=} \uparrow f \qquad [\![\emptyset]\!] \overset{\text{def}}{=} tt, ff : \mathtt{var}_{\bot} \qquad \langle\!\langle\emptyset\rangle\!\rangle \overset{\text{def}}{=} \mathbf{?}^{\iota} tt \otimes \mathbf{?}^{\iota} ff$$

$$[\![s\ \mathtt{cmd}\Uparrow]\!] \overset{\text{def}}{=} \mathtt{sync}^{\Uparrow}_{s} \quad \langle\!\langle s\ \mathtt{cmd}\Uparrow\rangle\!\rangle_f \overset{\text{def}}{=} \Updownarrow f \quad [\![E\cdot x : s]\!] \overset{\text{def}}{=} [\![E]\!]\cdot x : \mathtt{var}_s \quad \langle\!\langle E\cdot x : s\rangle\!\rangle \overset{\text{def}}{=} [\![E]\!]\cdot \mathbf{!}x$$

(Command) $(s = \mathsf{sec}(e)_E$ in all cases, $\mathtt{var}_s \overset{\text{def}}{=} \langle \mathtt{var}^{\mathbf{!}}_s, \mathtt{var}^{\mathbf{?}}_s\rangle)$

$$[\![E \vdash \mathtt{skip} : \rho]\!]_f \overset{\text{def}}{=} \overline{f}$$

$$[\![E \vdash c_1; c_2 : \rho]\!]_f \overset{\text{def}}{=} (\boldsymbol{\nu}\, g : \langle\overline{[\![\rho]\!]}, [\![\rho]\!]\rangle)([\![E \vdash c_1 : \rho]\!]_g \mid g.[\![E \vdash c_2 : \rho]\!]_f)$$

$$[\![E \vdash c_1 \mid c_2 : \rho]\!]_f \overset{\text{def}}{=} (\boldsymbol{\nu}\, f_1, f_2 : \langle\overline{[\![\rho]\!]}, [\![\rho]\!]\rangle)([\![E \vdash c_1 : \rho]\!]_{f_1} \mid [\![E \vdash c_2 : \rho]\!]_{f_2} \mid f_1.f_2.\overline{f})$$

$$[\![E \vdash x := e : \rho]\!]_f \overset{\text{def}}{=} \mathbf{eval}[\![e]\!]^E(b^{s'}).\overline{x}\mathtt{inr}(b' : \mathtt{bool}^{\mathbf{!}}_{s'}).([b' \leftarrow b] \mid P) \qquad (s' = E(x))$$

$$[\![E \vdash \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \rho]\!]_f \overset{\text{def}}{=} \mathbf{eval}[\![e]\!]^E(b^s).\mathbf{If}\langle b, [\![E \vdash c_1 : \rho]\!]_f, [\![E \vdash c_2 : \rho]\!]_f\rangle$$

$$[\![E \vdash \mathtt{while}\ e\ \mathtt{do}\ c : \rho]\!]_f \overset{\text{def}}{=} (\boldsymbol{\nu} g : \langle\overline{[\![\rho]\!]}, [\![\rho]\!]\rangle)(\overline{g} \mid \mathcal{E}\langle fg\vec{x}\rangle) \qquad (E = \{\vec{x} : \vec{s}\}, \alpha_i = \mathtt{var}_{s_i})$$

$$\text{where } \mathcal{E} \overset{\text{def}}{=} \mu X(f, g : \langle\overline{[\![\rho]\!]}, [\![\rho]\!]\rangle, \vec{x} : \vec{\alpha}).\ g.\mathbf{eval}[\![e]\!]^E(b^s).\mathbf{If}\langle b, ([\![E \vdash c : \rho]\!]_g \mid X\langle fg\vec{x}\rangle), \overline{f}\rangle$$

(Expression)

$$\mathbf{eval}[\![x]\!]^E(b^s).P \overset{\text{def}}{=} \overline{x}\mathtt{inl}(z : (\mathtt{bool}^{\mathbf{?}}_{s'})^{\downarrow}_{s'}).z(b : \mathtt{bool}^{\mathbf{?}}_{s}).P \qquad (s' = E(x))$$

$$\mathbf{eval}[\![\mathtt{tt}]\!]^E(b^s).P \overset{\text{def}}{=} \mathbf{Link}\langle b^s, [\![\mathtt{b}]\!]^{\bot}, P\rangle \qquad ([\![\mathtt{tt}]\!] \overset{\text{def}}{=} tt, \quad [\![\mathtt{ff}]\!] \overset{\text{def}}{=} ff)$$

$$\mathbf{eval}[\![e_1\ \mathsf{and}\ e_2]\!]^E(b^s).P \overset{\text{def}}{=} \mathbf{eval}[\![e_1]\!]^E(b_1^{s_1}).\mathbf{eval}[\![e_2]\!]^E(b_2^{s_2}).$$
$$\mathbf{If}\langle b_1^{s_1}, \mathbf{Link}\langle b^s, b_2^{s_2}, P\rangle, \mathbf{Link}\langle b^s, b_1^{s_1}, P\rangle\rangle \qquad (s_i = \mathsf{sec}(e_i)_E, s \geq s_1 \sqcup s_2)$$

$$\mathbf{Link}\langle b^s, b'^{s'}, P\rangle \overset{\text{def}}{=} (\boldsymbol{\nu}\, b : \mathtt{var}_s)(P \mid [b \leftarrow b']) \qquad (s' \leq s)$$

(Security of an expression)

$$\mathsf{sec}(x)_E \overset{\text{def}}{=} E(x) \qquad \mathsf{sec}(\mathsf{b})_E \overset{\text{def}}{=} \bot \qquad \mathsf{sec}(e_1\ \mathsf{and}\ e_2)_E \overset{\text{def}}{=} \mathsf{sec}(e_1)_E \sqcup \mathsf{sec}(e_2)_E$$

Fig. 3. Translation of the Smith-Volpano calculus

interest however lies in how *typability* is transformed via the embedding, and how this transformation sheds light on safe information-flow in the original system. The following theorem underpins this point. Below $\overline{A}$ dualises each mode in A which is assigned to a name, taking **?** as the dual of **!**.

Theorem 2 (Soundness). *If* $E \vdash c : \rho$, *then* $[\![E]\!] \cdot f : [\![\rho]\!] \vdash_s [\![E \vdash c : \rho]\!]_f \triangleright \overline{\langle\!\langle E\rangle\!\rangle} \otimes \langle\!\langle\rho\rangle\!\rangle_f$ *with* $s = \mathsf{sec}(\rho)$.

A significant consequence of Theorem 2 is that we obtain, via the non-interference of typed processes mentioned in Section 4, the original non-interference result by Volpano and Smith [33]. The result holds for all terms typable in rules with Figure 2, including typed terms not coming from [33]. As another significant point, the encoding illustrates the reason why the divergent command types cannot be elevated as the convergent ones. Let $E \vdash \mathtt{while}\ e\ \mathtt{do}\ c : s\ \mathtt{cmd}\Uparrow$. In the encoding, the body of the loop, which is at level s, depends on the branching at level $\mathsf{sec}(e)_E \leq s$: lowering s can make this dependency dangerous, hence we cannot degrade $s\,\mathtt{cmd}_{\Uparrow}$ as in the convergent types. Also note this argument does not use the restriction $s = \bot$ in the original type discipline.

5.3 Termination as Observable

After the preceding development, a natural question is whether we obtain any new information by doing such an endeavour or not. In this section we outline a technical development which may answer affirmatively to the question.

We first return to the restriction of the original system that allows only the level $\bot$ for divergent commands. This does seem a strong constraint, especially with multiple security levels. How does this constraint appear in process representation? It means we only assign $\mathtt{sync}^{\Uparrow}_{\bot}$ to a channel for the termination, which makes explicit the notion of termination as an observable, both as types and as behaviours. Once we have this notion, we ask what is the real content of having the observable only at $\bot$. Clearly the answer is: "we allow *everybody* to observe the termination." We may then ask what would be the outcome of *not* allowing everybody to observe the termination. Can this make sense? It seems it does: since the time of Multics and as was recently introduced in a widely known programming language [12], a mechanism by which we can prevent processes from even realising the presence of other processes, depending on assigned security levels, is a well-established idea in security, both from integrity and secrecy concerns.

Further, there is a technically important observation that the encoding in Figure 3 does *not* apparently impose restriction on levels of divergent types: indeed the argument for Theorem 2 hardly depends on it. Thus we generalise the $\mathtt{while}$ rule as follows.

$$(\mathsf{while})\quad \frac{E \vdash e : s \qquad E \vdash c : s\,\mathtt{cmd}_{\Uparrow}}{E \vdash \mathtt{while}\ e\ \mathtt{do}\ c : s\,\mathtt{cmd}_{\Uparrow}}$$

The new rule is significant in its loosened condition on the guard of the loop, allowing us to type, say, (with M being a level between H and L), $\mathtt{while}\ e^{\mathrm{M}}\ \mathtt{do}\ c\ :\ \mathrm{H}\,\mathtt{cmd}_{\Uparrow}$. With exactly the same encoding, we obtain the soundness result for the extended system with a statement identical to Theorem 2.

Further, this new soundness result leads to a non-interference for the extended imperative calculus just as in the original calculus. The formulation is however different since termination behaviours can change between two initial configurations if we set different values at levels lower than the termination observable. Fixing a base E, let s' be a stipulated level of observability of the termination, and assume there are two environments (assignments of truth values to variables) which are *equivalent with respect to* s', i.e. they only differ in variables at levels higher than s'. Suppose also s is the level of the command type of a well-typed c under E and $s \leq s'$ (thus if c includes a while command, its guard is not affected by the content of variables at levels above s'). Then if c terminates under one of these environments, it will also terminate under the other environment, and the two resulting environments are equivalent with respect to s' (hence with respect to s). If we are without the condition $s \leq s'$, we cannot guarantee the same consequence, though observables except the termination at each state are equivalent with respect to s', related in a coinductive way. See [16] for details. Thus we are again guaranteed secure information flow with added typability, by starting from a typed process representation of imperative program behaviour.

References

1. Abadi, M., Secrecy by typing in security protocols. *TACS'97*, LNCS, 611-637, Springer, 1997.
2. Abadi, M., Banerjee, A., Heintze, N. and Riecke, J., A core calculus of dependency, *POPL'99*, ACM, 1999.
3. Abadi, M., Fournet, C. and Gonthier, G., Secure Communications Processing for Distributed Languages. *LICS'98*, 868–883, IEEE, 1998.
4. Abramsky, S., Computational Interpretation of Linear Logic. *TCS*, vol 111, 1993.
5. Bodei, C, Degano, P., Nielson, F., and Nielson, H. Control Flow Analysis of π-Calculus. *CONCUR'98*, LNCS 1466, 84–98, Springer, 1998.
6. Boudol, G., The pi-calculus in direct style, *POPL'97*, 228–241, ACM, 1997.
7. Denning, D. and Denning, P., Certification of programs for secure information flow. *Communication of ACM*, ACM, 20:504–513, 1997.
8. Focardi, R. and Gorrieri, R., The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering,* 23(9), 1997.
9. Gay, S. and Hole, M., Types and Subtypes for Client-Server Interactions, *ESOP'99*, LNCS 1576, 74–90, Springer, 1999.
10. Girard, J.-Y., Linear Logic, *TCS*, Vol. 50, 1–102, 1987.
11. Goguen, J. and Meseguer, J., Security policies and security models. *IEEE Symposium on Security and Privacy*, 11–20, IEEE, 1982.
12. Gong, L., Prafullchandra, H. and Shcemers, R., Going beyond sandbox: an overview of the new security architecture in the Java development kit 1.2. USENIX Symposium on Internet Technologies and Systems, 1997.
13. Gray, J., Probabilistic interference. *Symposium on Security and Privacy*, 170–179, IEEE, 1990.
14. Honda, K., Types for Dyadic Interaction. *CONCUR'93*, LNCS 715, 509-523, 1993.
15. Honda, K., Composing Processes, *POPL'96*, 344-357, ACM, 1996.
16. A full version of the present paper, QMW CS technical report 767, 1999. Available at `http://www.dcs.qmw.ac.uk/~kohei`.
17. Honda, K. and Yoshida, N. On Reduction-Based Process Semantics. *TCS*. 151, 437-486, 1995.
18. Honda, K. and Yoshida, N. Game-theoretic analysis of call-by-value computation. *TCS*, 221 (1999), 393–456, 1999.
19. Hyland, M. and Ong, L., Pi-calculus, dialogue games and PCF, *FPCA'93*, ACM, 1995.
20. Kobayashi, N., Pierce, B., and Turner, D., Linear types and π-calculus, *POPL'96*, 358–371, 1996.
21. Lafont, Y., Interaction Nets, *POPL'90*, 95–108, ACM, 1990.

22. Larsen, K. and Skou, A. Bisimulation through probabilistic testing. *Information and Computation*, 94:1–28, 1991.
23. Lopes, L, Silva, F. and Vasconcelos, F. A Virtual Machine for the TyCO Process Calculus. *PPDP'99*, 244–260, LNCS 1702, Springer, 1999.
24. Lowe, G. Defining Information Flow. MCS technical report, University of Leicester, 1999/3, 1999.
25. McLean, J. Security models and information flow. *IEEE Symposium on Security and Privacy*, 1990.
26. Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989.
27. Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes, *Info. & Comp.* 100(1), pp.1–77, 1992.
28. Pierce, B and Sangiorgi.D, Typing and subtyping for mobile processes, *MSCS* 6(5):409–453, 1996.
29. Roscoe, A. W. Intensional Specifications of Security Protocols, *CSFW'96*, IEEE, 1996.
30. Sangiorgi, D. π-calculus, internal mobility, and agent-passing calculi. *TCS*, 167(2):235–271, 1996.
31. Schneider, S. Security properties and CSP. *Symposium on Security and Privacy*, 174–187, 1996.
32. Vasconcelos, V., Typed concurrent objects. *ECOOP'94*, LNCS 821, pp.100–117. Springer, 1994.
33. Volpano, D. and Smith, G., *Secure information flow in a multi-threaded imperative language*, pp.355–364, *POPL'98*, ACM, 1998.
34. Volpano, D. and Smith, G., Language Issues in Mobile Program Security. to appear in LNCS, Springer, 1999.
35. Volpano, D., Smith, G. and Irvine, C., *A Sound type system for secure flow analysis.* J. Computer Security, 4(2,3):167–187, 1996.
36. Yoshida, N. Graph Types for Mobile Processes. *FST/TCS'16*, LNCS 1180, pp.371–386, Springer, 1996. The full version as LFCS Technical Report, ECS-LFCS-96-350, 1996.