JFP **15** (2): 293–349, 2005. © 2005 Cambridge University Press
DOI: 10.1017/S0956796804005477 Printed in the United Kingdom

Noninterference through flow analysis

KOHEI HONDA
Queen MaryCollege, University of London, Mile End Road, London, UK
(*e-mail:* kohei@dcs.gmul.ac.uk)

NOBUKO YOSHIDA
Imperial College, University of London, London, UK
(*e-mail:* yoshida@doc.ic.ac.uk)

Abstract

This paper proposes new syntactic inference rules which can directly extract information flow in a given typed process in the π-calculus. In the flow analysis, a flow in a process is captured as a chain of possible interactions which transform differences in behaviours from one part of its interface to another part of its interface. Polarity in types plays a fundamental role in the analysis, which is elucidated via examples. We show that this inductive flow analysis can be used for giving simple proofs of noninterference in the secrecy analyses for the π-calculus with linear/affine typing, including its concurrent, stateful extensions.

Capsule Review

Noninterference is one of the main operational properties that can be established in the framework offered by this paper. The focus is indeed on secure information flow for typed versions of the pi calculus (a first simple typed version is incrementally extended). The required information is extracted from the syntax of processes, in the form of syntactic inference rules.

The key idea is the correspondence between polarity in types and information flow. As a matter of fact, polarities have to do with the directions of information flow: a typed channel is positive if the channel emits information, while a typed channel is negative if the channel receives information. If the system does not infer any insecure flow in the process under analysis, then noninterference is automatically ensured, i.e. no high-level information flows down to low-level channels. The proof of noninterference is therefore carried in a simple and elegant way. The paper is well written and is notable the way notions are rearranged to give new insights.

1 Introduction

This paper introduces new syntactic inference rules which can directly extract possible flows of information from a given typed process. A flow in a process is, intuitively speaking, a chain of possible interactions which may transform differences in behaviour from one part of its interface (which is a set of channels) to another part of its interface (which is again a set of channels). In other words, the flows in a process P are the embodiment of the transformation which P would induce when it is connected with another process at designated channels.

Types associated with process channels play an essential role in our framework. The notion of types in which we are particularly interested is the linear/affine type discipline and its extensions (Berger *et al.*, 2001; Yoshida *et al.*, 2001; Yoshida *et al.*, 2002; Honda & Yoshida, 2002; Honda *et al.*, 2004), whose origins are traced back both to the logical traditions including Linear Logic and game semantics (Abramsky *et al.*, 2000; Hyland & Ong, 2000; Honda & Yoshida, 1997; Girard, 1987) and to the process-calculi tradition, sorting (Milner, 1992b; Vasconcelos & Honda, 1993) and its refinement (Honda, 1996; Yoshida, 1996; Kobayashi *et al.*, 1996; Pierce & Sangiorgi, 1996). A central feature of this type discipline is the use of duality, where types designate complementary nature of the possible interactions carried out at channels (for example input and output in the simplest case). Thus types constrain both a process and its environment. This two-way constraint inductively controls the ways by which two processes influence each other in a sequence of mutual interactions, or, in other words, the ways by which information flows between two parties, without sacrificing essential expressiveness. This articulated form of mutual effects leads to a tractable way to extract information flows following the syntax of processes.

As is well-known, the dynamics of diverse language constructs in programming languages are representable as those in processes via encoding (as Milner showed in the case of the λ-calculus (Milner, 1992a) and Walker showed in the case of an object-oriented language (Walker, 1995)). The use of duality-based types is significant in this context again, since it allows these embeddings semantically accurate in the sense that they become equationally fully abstract (Berger *et al.*, 2001; Yoshida *et al.*, 2001; Berger *et al.*, 2003; Honda *et al.*, 2004). This makes it possible to carry over the flow analysis on processes to that on programs via simple encodings.

The well-organised form of information flow via duality types has a clean representation in the notion of *polarity*, one of the basic themes of this paper. Types, or typed channels, come with essentially two polarities, *positive* and *negative*. Intuitively, a type (or a typed channel) which emits, or generates, information is said to be positive, while a type which receives, or consumes, information is said to be negative (these polarities are *not* directly related to the distinction between input and output). Whether a given type is positive or not can be easily verified using a simple behavioural characterisation, which we shall present in section 3.2. The polarity gives basic insight on information flow in typed processes. For example, in processes representing pure functional behaviours (including those based on call-by-name and call-by-value evaluations), types are strongly polarised in the sense that each type is classified exclusively as either positive or negative; thus information always flows in one direction, and its analysis is relatively simple. However when processes contain stateful behaviours, such as those of imperative variables, types can be simultaneously both positive and negative. As a basic example, when an imperative variable is read, it surely emits information; whereas, when it is written, it receives one. Thus an imperative variable has mixed polarities, making the associated flow, hence its analysis, more complex. This observation sheds new light on the difficulty of analysing programs with imperative effects, especially in the presence of higher-order references. The polarities in the present work are closely related with those in Polarised Linear Logic (Laurent, 2002a; Laurent, 2002b; Laurent & Regnier, 2003); Yoshida *et al.* (2002) first discussed this notion in the context of the linear π-calculus.

Polarities directly suggest how the flow inference rules can be derived following the syntactic structure of typed processes. Since polarities represent directions of information flow, they indicate how we can incrementally extract a flow from a typed process starting from flows of its subterms. This is so even with types of mixed polarities (as in types for stateful behaviours). Viewing a type of mixed polarities with a specific polarity determines how we may position a typed channel in a flow, guiding the construction of flow inference rules. The chosen polarity also indicates how we measure secrecy levels of typed channels. The resulting analysis yields, in its most informative form, a sequence of typed channels through which information is transmitted within a process; or, in a simpler case (on which we focus in this paper), the source and the target of a flow of a process. In either way, the flow inference traces a sequence of primitive name passing interactions in its derivation, so that the analysis can transparently be related to the operational behaviour of a process.

We use this transparency in the application of the flow analysis in the present paper, where we demonstrate how the flow analysis can be used for giving simple proofs of noninterference (Denning & Denning, 1977; Goguen & Meseguer, 1982) for the type-based secrecy analysis introduced in Honda & Yoshida (2002). In a type-based secrecy analysis, we use typing rules for statically *ensuring* safety in information flow, so that no high-level information flows down to low-level channels. In contrast, a flow analysis *extracts* information flow from a given process. Thus, all we need to do for justifying a secrecy analysis via a flow analysis, is to show securely typed processes never own an insecure flow (i.e. a flow from high-level channels to a low-level one). Since we can translate the lack of insecure flow into the noninterference property (i.e. that there is no influence from the high-level behaviour to the low-level behaviour) in an organised way, we can conclude that the secrecy analysis is sound.

Summary of the technical contributions. The main technical contributions of the present work may be summarised as follows:

- The elucidation of the correspondence between polarity in types and flows of information in processes in the duality-based type discipline.
- The inductive flow inference rules for the linear/affine π-calculus π^{LA} and its stateful extension π^{LAR}, and the establishment of operational properties of flow-secure processes, among others noninterference.
- The application of the flow inference system to the noninterference results in the secrecy type discipline of Honda & Yoshida (2002).

The key concepts are illustrated with examples to offer intuition as far as possible.

Structure of the paper. The remainder of the paper is organised as follows. Section 2 gives a brief summary of the syntax and reduction of the π-calculus, the linear/affine type discipline, and its secrecy enhancement. Section 3 introduces basic ideas of the flow analysis through examples, and discusses their connection to the notion of polarity. Section 4 introduces the flow analysis for the unary linear/affine π-calculus, π^{LA}. Section 5 proves noninterference for the basic secrecy typing for π^{LA}. Section 6 discusses extensions to inflation (which allows relaxing of the secrecy typing as far as global flows remain safe) and branching/selection constructs. Section 7 extends the flow analysis to state, concurrency and non-determinism. Finally, section 8 concludes the paper with discussions on related works and further issues. Appendices present supplementary materials omitted in the main sections.

2 Preliminaries

This section briefly summarises the linear/affine π-calculus and its secrecy enhancement, first introduced in Honda & Yoshida (2002). Although this summary is technically self-contained, the reader may refer to Honda & Yoshida (2002) for detailed illustration and more examples.

2.1 *Syntax and reduction*

The π-calculus used in this paper is the asynchronous π-calculus (Honda & Tokoro, 1991). The following gives the reduction rule of the asynchronous π-calculus:

$$x(\vec{y}).P \mid \overline{x}\langle\vec{v}\rangle \longrightarrow P\{\vec{v}/\vec{y}\} \tag{1}$$

Here $\vec{y}$ denotes a potentially empty vector $y_1 \ldots y_n$ of names, $\mid$ denotes parallel composition, $x(\vec{y}).P$ is input, and $\overline{x}\langle\vec{v}\rangle$ is asynchronous output. Operationally, this reduction represents the consumption of an asynchronous message by a receptor. The idea extends to a receptor with a replication, $!\,x(\vec{y}).P$:

$$!\,x(\vec{y}).P \mid \overline{x}\langle\vec{v}\rangle \longrightarrow\ !\,x(\vec{y}).P \mid P\{\vec{v}/\vec{y}\}, \tag{2}$$

where the replicated process remains in the configuration after reduction.

Types for processes prescribe usage of names. To be able to do this with precision, it is important to control dynamic sharing of names. For this purpose, it is useful to restrict name passing to *bound (private) name passing*, where only bound names are passed in interaction. This allows tighter control of sharing without losing essential expressiveness, making it easier to administer name usage in more stringent ways. The resulting calculus is sometimes called the asynchronous πI-calculus in the literature (Sangiorgi, 1996) and has the equivalent expressive power with free name passing (as proved in section 6 of Yoshida *et al.* (2001) in the typed case). Although we can easily treat free name passing directly, in the present study, using bound name passing leads to a simple and precise flow analysis, as well as to a clean fully abstract translation of the functional calculi into the π-calculus. Syntactically, we restrict outputs to the form $(\boldsymbol{\nu}\,\vec{y})(\overline{x}\langle\vec{y}\rangle|P)$ (where names in $\vec{y}$ are pairwise distinct), which we henceforth write $\overline{x}(\vec{y})P$. For dynamics, we have the following forms of reduction by the restriction to the bound output. The reduction relation $\longrightarrow$ is generated from the following rules, closed under output prefix, restriction and parallel composition (taking processes modulo the standard structural congruence).

$$\begin{array}{rcl} x(\vec{y}).P \mid \overline{x}(\vec{y})Q & \longrightarrow & (\boldsymbol{\nu}\, \vec{y})(P \mid Q) \\ !x(\vec{y}).P \mid \overline{x}(\vec{y})Q & \longrightarrow & !x(\vec{y}).P \mid (\boldsymbol{\nu}\, \vec{y})(P \mid Q) \end{array}$$

"$\overline{x}(\vec{y})Q$" indicates that $\overline{x}(\vec{y})$ is an asynchronous output exporting $\vec{y}$ which are originally local to Q. After communication, $\vec{y}$ are shared between P and Q.

The formal grammar of the calculus is defined below. Below and henceforth $x, y, \ldots$ range over a countable set of names.

$$P \ ::= \ x(\vec{y}).P \ \mid \ \overline{x}(\vec{y})P \ \mid \ P \mid Q \ \mid \ (\boldsymbol{\nu}\, x)P \ \mid \ \mathbf{0} \ \mid \ !x(\vec{y}).P$$

$x(\vec{y}).P$ (resp. $\overline{x}(\vec{y})P$) is an input (resp. output). $P \mid Q$ is a parallel composition, $(\boldsymbol{\nu}\, x)P$ is a restriction and $!x(\vec{y}).P$ is a replicated input. We omit the empty vector: for example, $\overline{a}$ stands for $\overline{a}()$ and $a.P$ stands for $a().P$. The bound/free names are defined as usual. $\mathsf{fn}(P)$ denotes the set of free names in P. We assume that names in a vector $\vec{y}$ are pairwise distinct. The definitions of structural equality $\equiv$, given in Appendix A, is standard (Berger *et al.*, 2001; Yoshida *et al.*, 2002; Yoshida *et al.*, 2001).

The following simple examples of processes are used throughout the rest of the paper.

Example 2.1 (processes)

1. A *unit process*, defined as $[\![()]\!]_x \overset{\text{def}}{=} !x(w).\overline{w}$, immediately emits a signal to the received channel w.
2. A *copycat*, $[x \to y] \overset{\text{def}}{=} !x(c).\overline{y}(c')c'.\overline{c}$, is a link between two locations x and y; when asked at x, it asks back at y, then, on receiving the answer at c' from y, forwards it back at c as an answer to the initial name invocation. Inserting this agent between two processes does not change their original behaviour. For example:

$$\begin{array}{rlcl} & [\![()]\!]_y | [x \to y] | \overline{x}(c)P & \longrightarrow & [\![()]\!]_y | [x \to y] | (\boldsymbol{\nu}\, c)(\overline{y}(c')c'.\overline{c} \mid P) \\ \longrightarrow & [\![()]\!]_y | [x \to y] | (\boldsymbol{\nu}\, c')(\overline{c'} | c'.\overline{c} \mid P) & \longrightarrow & [\![()]\!]_y | [x \to y] | (\boldsymbol{\nu}\, c)(\overline{c} \mid P) \end{array}$$

which is the same as $[\![()]\!]_y | \overline{y}(c)P \longrightarrow [\![()]\!]_y | (\boldsymbol{\nu}\, c)(\overline{c} \mid P)$ save some internal reductions. Using the copycat, we can translate free name passing into bound name passing as: $\overline{x}\langle v \rangle \overset{\text{def}}{=} \overline{x}(v')[v' \to v]$. For example, assuming $c, e \notin \mathsf{fn}(P)$, we have, in free name passing:

$$x(e)\overline{e}(c)c.P \mid \overline{x}\langle v \rangle \longrightarrow \overline{v}(c)c.P$$

This reduction can be simulated using the copycat as follows:

$$\begin{array}{rlcl} & x(e)\overline{e}(c)c.P \mid \overline{x}(e)[e \to v] & \longrightarrow & (\boldsymbol{\nu}\, e)(\overline{e}(c)c.P \mid [e \to v]) \\ \longrightarrow & (\boldsymbol{\nu}\, c)(\overline{v}(c')c'.\overline{c} \mid c.P) \mid (\boldsymbol{\nu}\, e)[e \to v] & \approx & \overline{v}(c')c'.P \end{array}$$

where $\approx$ is the standard weak bisimilarity (Milner *et al.*, 1992; Honda & Tokoro, 1991). See Section 6 in (Yoshida *et al.*, 2001) for the formal embedding result between free and bound name passing based on the copycat encoding.

3. An *omega agent*, $\Omega_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)([u \rightarrow y] | [y \rightarrow u])$, immediately diverges after the initial invocation at u; we can check $\Omega_u|\overline{u}(c)P \longrightarrow\!\!\!\!\longrightarrow \Omega_u|\overline{u}(c)P \longrightarrow\!\!\!\!\longrightarrow \Omega_u|\overline{u}(c)P \longrightarrow\!\!\!\!\longrightarrow \ldots$.

2.2 *Basic idea of types and typings*

In this subsection, we review the basic idea of the linear/affine type discipline (Honda & Yoshida, 2002). This type discipline allows a precise embedding of functional computation in the π-calculus by restricting process behaviour to be a *confluent* one. To realise confluence, we use the following idea:

(A) for each replicated name there is a unique stateless replicated input with zero or more dual outputs; or
(B) for each linear (resp. affine) name there are a unique input and a unique (resp. at most one) output

As an example of the first constraint, let us consider the following two processes.

$$P_1 \stackrel{\text{def}}{=} !\,b.\overline{a} \mid !\,b.\overline{c} \qquad\qquad P_2 \stackrel{\text{def}}{=} !\,b.\overline{a} \mid \overline{b} \mid !\,c.\overline{b} \tag{3}$$

P_1 is untypable because b is associated with two replicators: but P_2 is typable since, while output at b appears twice, a replicated input at b appears only once. As an example for the second condition, let us consider:

$$P_3 \stackrel{\text{def}}{=} b.\overline{a} \mid c.\overline{a} \mid a.\mathbf{0} \qquad\qquad P_4 \stackrel{\text{def}}{=} b.\overline{a} \mid c.\overline{b} \mid a.(\overline{c} \mid \overline{e}) \tag{4}$$

Then P_3 is not typable as a appears twice as output, while P_4 is typable since each channel appears at most once as input and output. We can also further ensure the *terminating* behaviour by introducing the following constraint.

(C) channels have no circular dependency.

For example,

$$P_5 \stackrel{\text{def}}{=} !\,b.\overline{a} \mid !\,a.\overline{b} \tag{5}$$

is untypable under this constraint. We can easily observe that if we compose message $\overline{a}$ to the above process, then the computation does not terminate. P_4 is also untypable under this constraint.

The typing discipline which ensures (A) and (B) is called *affine*, while one which ensures (A,B,C) is called *linear*. We make the meaning of these terms more precise below:

- *Affinity.* This denotes possibly diverging behaviour in which a question is given an answer at most once.
- *Linearity.* This denotes terminating behaviour in which a question is always given an answer precisely once.

As a theoretical underpinning, Berger *et al.* (2001) and Yoshida *et al.* (2001) have shown that PCF and strongly normalising λ-calculi are fully abstractly embeddable in the affine and linear π-calculus, respectively. As illustrated above, “linearity” means more than terminating behaviour: it indicates a process always returns an answer at a linear channel. In replicated channels, linearity means convergence while affinity means potential divergence. For example, P_4 and P_5 are typable by the affine typing system, while they are not so by the linear one. We can further mix these two type disciplines. This mixture of nontermination (affinity) and termination (linearity) is fine-grained: even inside a process engaged in a linear interaction, nonterminating computation may take place, and linear interactions themselves can invoke nonterminating computation. For example, take the following process:

$$!b(xc).(\overline{x} \mid \overline{c}) \mid \overline{b}(ac)(!a.\overline{a} \mid c.P)$$

This process has first an interaction at b, then does an infinite series of actions at a while having a linear answer at c: in the above terminology, it is affine at a, while it is linear at c. This fine-grained mixture is vital for flexible embeddings of various programming constructs and plays an essential role in applications, including secure information analysis (Honda & Yoshida, 2002).

2.3 *The linear/affine typing system*

This subsection summarises the linear/affine typing system in Honda & Yoshida (2002), introducing the minimum notations and definitions needed for the flow analysis. While we leave some of the technical details to Appendix (further discussions and examples are also found in Honda & Yoshida (2002)), the knowledge in this subsection suffices to read the rest of the paper.

Action modes. We use the following *action modes* (Berger *et al.*, 2001; Yoshida *et al.*, 2001), which prescribe different modes of interaction at each channel. The L-modes correspond to linear modes in Yoshida *et al.* (2001) while the A-modes to affine modes in Berger *et al.* (2001).

$\downarrow_{\mathrm{L}}$	Linear input	$\uparrow_{\mathrm{L}}$	Linear output
$\downarrow_{\mathrm{A}}$	Affine input	$\uparrow_{\mathrm{A}}$	Affine output
$!_{\mathrm{L}}$	Linear server	$?_{\mathrm{L}}$	Client request to $!_{\mathrm{L}}$
$!_{\mathrm{A}}$	Affine server	$?_{\mathrm{A}}$	Client request to $!_{\mathrm{A}}$

We also use the mode $\updownarrow$ which indicates that a channel is no longer available for *further* composition with the outside; for example, if $x.\mathbf{0}$ has a $\downarrow_{\mathrm{L}}$-mode and $\overline{x}$ has a $\uparrow_{\mathrm{L}}$-mode, then $x.\mathbf{0} \mid \overline{x}$ has $\updownarrow$-mode at x. The $\updownarrow$-mode at x indicates that the process cannot be composed with any process that has x as a free name (e.g. $x.\mathbf{0} \mid \overline{x}$). $p, p', \ldots$ range over action modes. The modes in the left column are *input modes* while those in the right are *output modes*. The pair of modes in each row are *dual* to each other, writing $\overline{p}$ for the dual of p. We set $\mathcal{M}_{\downarrow} = \{\downarrow_{\mathrm{L}}, \downarrow_{\mathrm{A}}\}$ etc. and often write $?$ to denote either $?_{\mathrm{L}}$ or $?_{\mathrm{A}}$. Similarly for $!_{\mathrm{L}}, \uparrow_{\mathrm{L}}, \downarrow_{\mathrm{L}}$.

Types. Let $(\mathcal{L}, \sqsubseteq, \mathbb{H}, \mathbb{L})$ be a complete lattice of secrecy levels (higher means more secret), ranged over by $s, s', \ldots$. Then the set of channel types annotated with secrecy levels, ranged over by $\tau, \rho, \sigma, \ldots$, are generated from the following grammar.

$$\tau \quad ::= \quad \tau_{\mathrm{I}} \mid \tau_{\mathrm{O}} \mid \updownarrow \qquad\qquad \begin{array}{lll} \tau_{\mathrm{I}} & ::= & (\vec{\tau}^{?})^{\downarrow_{\mathrm{L}}} \mid (\vec{\tau}^{?})_{s}^{\downarrow_{\mathrm{A}}} \mid (\vec{\tau}^{?}\tau^{\uparrow})^{!_{\mathrm{L}}} \mid (\vec{\tau}^{?}\tau^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}} \\ \tau_{\mathrm{O}} & ::= & (\vec{\tau}^{!})^{\uparrow_{\mathrm{L}}} \mid (\vec{\tau}^{!})_{s}^{\uparrow_{\mathrm{A}}} \mid (\vec{\tau}^{!}\tau^{\downarrow})^{?_{\mathrm{L}}} \mid (\vec{\tau}^{!}\tau^{\downarrow_{\mathrm{A}}})^{?_{\mathrm{A}}} \end{array}$$

$\vec{\tau}$ is a vector of types and τ^{p} indicates the (outermost) mode of τ (we assume $\updownarrow$ has the mode $\updownarrow$). Since non-trivial information flow occurs only at affine output/input channels, we only annotate them with a secrecy level (Honda & Yoshida, 2002). We also often omit the secrecy level if it is unnecessary. Note that an input only carries an output (and dually), and that only a replicated linear input can carry a (unique) linear output (and dually). The former condition comes from game semantics (Hyland & Ong, 2000; Abramsky *et al.*, 2000; Honda & Yoshida, 1997) (which is an important constraint to embed higher-order functions fully abstractly (Berger *et al.*, 2001; Berger *et al.*, 2003; Honda *et al.*, 2004; Yoshida *et al.*, 2001)), while the latter condition ensures an invocation at linear replication will eventually terminate, firing an associated linear output. We write $\mathsf{sec}(\tau)/\mathsf{md}(\tau)$ for the outermost secrecy level/mode of τ. The *dual of* τ, written $\overline{\tau}$, is the result of dualising all action modes.

We define the least commutative partial operation, $\odot$, which control the composition of channels as:

$$(1)\ \tau^{?} \odot \tau^{?} = \tau^{?} \text{ and } \tau^{!} \odot \overline{\tau}^{?} = \tau^{!} \qquad\qquad (2)\ \tau^{\uparrow} \odot \overline{\tau}^{\downarrow} = \updownarrow$$

(1) and (2) ensure the two constraints (A) and (B) in section 2.2.

An action type is a finite map from names to channel types together with directed edges between names, where edges represent causality among linear (resp. linear replicated) channels. Formally an *action type*, denoted $A, B, \ldots$, is a finite directed graph with nodes of the form $x : \tau$, such that no names occur twice; and *causality edge* $x : \tau \rightarrow y : \tau'$ is of the form: (1) from a linear input $\downarrow_{\mathrm{L}}$ to a linear output $\uparrow_{\mathrm{L}}$ or (2) from a linear replication $!_{\mathrm{L}}$ to linear client output $?_{\mathrm{L}}$. $\overline{A}$ dualises all types in A. We write $A(x)$ for the channel type assigned to x occurring in A. The partial operator $A \odot B$ is defined iff channel types with common names compose and the adjoined graph does not have a cycle. This avoids divergence on linear channels. For example, $a : \tau_1 \rightarrow b : \tau_2$ and $b : \overline{\tau_2} \rightarrow a : \overline{\tau_1}$ are not composable, hence a process such as $P_5 \overset{\mathrm{def}}{=} !a.\overline{b} \mid !b.\overline{a}$ is untypable. We write $A_1 \asymp A_2$ when such composition is possible, while the result of composition is written $A_1 \odot A_2$ (see Appendix B for formal definitions). By this operation, we can guarantee the condition (C) in section 2.2 for linear channels. Non-circular causality between linear channels guarantees liveness at linear output channels, resulting in a distinct treatment of information flow at linear channels.

Tamper Level. In the secrecy typing, we use a function which maps an action type to a secrecy level, called *tamper level* (in the sense that it is the level at which a process may affect, or tamper, the environment). It is first defined on channel types, and then extended to action types.

Definition 2.2 We say τ is *immediately tampering* if $\mathsf{md}(\tau) = \uparrow_{\mathrm{A}}$, while τ is *innocuous* if $\mathsf{md}(\tau) \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}, \updownarrow\}$. Then the *tamper level of* τ, denoted $\mathsf{tamp}(\tau)$, is inductively given by:

$$\begin{array}{rcll} \mathsf{tamp}(\tau) & = & \mathsf{sec}(\tau) & \text{if } \tau \text{ is immediately tampering.} \\ \mathsf{tamp}(\tau) & = & \mathbb{H} & \text{if } \tau \text{ is innocuous.} \\ \mathsf{tamp}((\vec{\tau})^{p}) & = & \sqcap\{\mathsf{tamp}(\tau_i)\} & \text{if } p \in \mathcal{M}_{!,\downarrow} \cup \{\uparrow_{\mathrm{L}}\}. \end{array}$$

We set $\mathsf{tamp}(A) \stackrel{\text{def}}{=} \sqcap\{\mathsf{tamp}(\tau) \mid x : \tau \in A\}$.

Intuitively, a channel type is immediately tampering if it emits non-trivial information at the time of interaction. Even if a type is not immediately tampering, an action of that type can have a non-trivial effect on the environment via an immediately tampering type inside. However an innocuous type does not even have such a latent effect: for example, $?_{\mathrm{L}}$-actions just create a new copy of a resource, leaving the environment as it originally was. Thus the tampering level of $?_{\mathrm{L}}$-types is $\mathbb{H}$. Note this discussion relies on the stateless nature of recursive behaviour at types $!_{\mathrm{L}}$ and $!_{\mathrm{A}}$ and are refined later in section 7 when we incorporate stateful behaviours.

Typing System. The secrecy linear/affine typing uses the judgement of the form:

$$\vdash_{\mathsf{sec}} P \triangleright A$$

The typing rules are given in Appendix B. We also use the standard linear/affine typing (which does not care about secrecy), whose typable processes are the target of the flow analysis. The typing rules are also presented in Appendix B. Typed processes in the linear/affine type discipline are written:

$$\vdash P \triangleright A$$

The key modification of the linear/affine typing for the secrecy typing, is that, when we infer an affine input process $x(\vec{y}).P$, we ensure that the tampering level of P is the same as, or higher than, the secrecy level of channel x. This is because an affine input directly receives information. If the information is received at s, its effects may only be safely shown to the outside at s or above.

We briefly illustrate the idea of tampering levels and secure typing by examples. The types in these examples will be used throughout the remaining sections.

Example 2.3 (tampering and secure typing) Let $\mathbb{L}$ and $\mathbb{H}$ be the lowest and the highest secrecy levels, respectively.

1. $()_s^{\uparrow_{\mathrm{A}}}$ is immediately tampering with level s; it transmits information by having two possibilities: either outputting at that channel, or not doing so at all. On the other hand, $()^{\uparrow_{\mathrm{L}}}$ is not immediately tampering: in fact, its tamper level is $\mathbb{H}$ (note no secrecy level is attached). This is because this type represents a behaviour which necessarily sends an empty output precisely once: its behaviour is completely determined by its type, so no information is transmitted by interaction.

2. Let $\star_s = (()_s^{\uparrow_\mathrm{A}})^{!_\mathrm{A}}$. Then we have $\mathsf{tamp}(\star_s) = s$. $\star_s$ is not immediately tampering, but it affects the environment latently. To see this, take Ω_x and $[\![()]\!]_x$. Then both are assigned the same type $x : \star_s$. As these inhabitants show, the type $\star_s$ does contain information of divergence/termination at level s. Note that we can assign $\tau' = (()^{\uparrow_\mathrm{L}})^{!_\mathrm{L}}$ to $[\![()]\!]_x$, but not to Ω_x, where $\mathsf{tamp}(\tau') = \mathbb{H}$. In fact, a(n essentially unique) process inhabiting $x : \tau'$ is always ready to receive at x; then it necessarily outputs an empty message via that name precisely once. Thus neither interaction at x nor that at c contains (emits) information.
3. $\vdash_{\mathsf{sec}} y.\overline{x} \rhd y : ()_{\mathbb{L}}^{\downarrow_\mathrm{A}}, x : ()_{\mathbb{H}}^{\uparrow_\mathrm{A}}$ is well-typed, but $\vdash_{\mathsf{sec}} y.\overline{x} \rhd y : ()_{\mathbb{H}}^{\downarrow_\mathrm{A}}, x : ()_{\mathbb{L}}^{\uparrow_\mathrm{A}}$ is not. Similarly $\vdash_{\mathsf{sec}} [x \to y] \rhd x : \star_{s'}, y : \overline{\star_s}$ is well-typed iff $s \sqsubseteq s'$. Then $\vdash_{\mathsf{sec}} \Omega_x \rhd x : \star_s$ is always well-typed.

These examples clarify the close connection between the secrecy typing and the linear/affine type discipline. If a channel x has the linear type $(()^{\uparrow_\mathrm{L}})^{!_\mathrm{L}}$, then we do not have to consider its secrecy level because no flow of information at x arises as it surely will terminate. On the other hand, if x is typed by $\star_s$, we cannot predict whether interaction at x terminates or not (as both $[\![()]\!]_x$ and Ω_x are typable by $\star_s$), hence the secrecy level should be considered.

3 Basic ideas of flow analysis

3.1 Definition of flow

In the flow analysis we shall present in this section, we use a *typed name*, which is a pair of a port name and its type, written $x : \tau$, and a *typed process*, which is a pair of a typable process and its action type, written P^A. The flow analysis formally uses the sequent of the following shape.

$$\vdash P^A \blacktriangleright \mathsf{F},$$

which we often write $\vdash P \blacktriangleright \mathsf{F}$, leaving A implicit. In this sequent, F is a *flow set*, which is simply a finite set of *flows*. Each flow in F relates a finite set of typed names with distinct channels, say Γ, which are called *sources*, to a typed name, say $x : \tau$, which is called *target*. In a flow set, we always assume two flows never own distinct target channels. A flow is written using the following syntax.

$$\Gamma \rightsquigarrow x : \tau$$

Note $y_i : \tau_i \in \Gamma$ with $i = 1, 2$ implies $y_1 \neq y_2$. A flow $\Gamma \rightsquigarrow x : \tau$ intuitively says interaction at channels in Γ is needed to produce information that is emitted at x.

For clarity of the notation, we fix the following grammar for flows and flow sets.

$$\mathsf{F} \quad ::= \quad \emptyset \mid \mathsf{F}, \Gamma \rightsquigarrow x : \tau \qquad \Gamma \quad ::= \quad \emptyset \mid \Gamma \cdot x : \tau$$

where we assume "$\rightsquigarrow$" associates stronger than "," while "$\cdot$" does so stronger than "$\rightsquigarrow$". For example,

$$y_1 : \tau_1 \cdot y_2 : \tau_2 \rightsquigarrow x : \tau,\ y_3 : \tau_3 \rightsquigarrow x' : \tau' \text{ means } ((y_1 : \tau_1 \cdot y_2 : \tau_2) \rightsquigarrow x : \tau), (y_3 : \tau_3 \rightsquigarrow x' : \tau')$$

Further we shall implicitly assume commutativity, associativity and idempotence of the operations "," and "·" so that they define sets rather than sequences. We also use the operations on sets such as $\bigcup_i \mathsf{F}_i$ when it is convenient.

3.2 *Polarities and flows*

As we briefly discussed in the Introduction, the types which embody generation/emittance of information are called *positive*, while types which embody consumption/reception of information are called *negative*. This informal idea can be made precise by the following simple operational characterisation. Below "$Q \Downarrow_w$" means there exist $Q \longrightarrow^* (\boldsymbol{\nu}\, \vec{c})(\overline{w}(\vec{b})P|R)$ for some $w \notin \{\vec{c}\}$, $\vec{b}$, P and R. "$Q \Uparrow$" means not $Q \Downarrow_w$ for any w.

Definition 3.1 τ is *behaviourally positive* (or, dually, $\overline{\tau}$ is *behaviourally negative*) iff $P_1|R \Downarrow_w$ and $P_2|R \Uparrow$ for some $\vdash P_{1,2} \triangleright x : \tau$ and $\vdash R \triangleright x : \overline{\tau}, w : ()_s^{\uparrow_{\mathrm{A}}}$.

In Definition 3.1, $P_{1,2}$ are the processes which produce information, or difference in behaviour, at channel x; whereas R is the opponent process which receives information, or gets affected by the difference in behaviour, via x. Then information at x is transformed to an ultimate convergence behaviour at w in R. Here the use of $()^{\uparrow_{\mathrm{A}}}$ is because it is the type for the basic observable in the maximally consistent reduction-based equality in the linear/affine π-calculus (Berger *et al.*, 2001; Berger *et al.*, 2003; Honda & Yoshida, 2002). This equality, $\cong$, is defined as the maximum typed congruence which satisfies, whenever $P_1^A \cong P_2^A$ such that $A = x : ()_s^{\uparrow_{\mathrm{A}}}$, we have:

$$P_1 \Downarrow_x \quad \Leftrightarrow \quad P_2 \Downarrow_x .$$

The relation $\cong$ is maximally consistent in the sense that the congruent closure of adding even a single equation is the universal relation.

Thus Definition 3.1 says that, for a given type τ, if there exist processes which can make a semantic difference via a channel of type τ when composed with a suitable receiver-transmitter to the standard observable type, then we call the type τ positive and $\overline{\tau}$ negative.

We illustrate the idea of behavioural polarities through a couple of concrete examples.

Example 3.2 (behavioural polarities)

1. All types of mode $\uparrow_{\mathrm{A}}$ are behaviourally positive. As an example, take:

$$Q_1 \stackrel{\text{def}}{=} \overline{c} \quad \text{and} \quad Q_2 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, ab)(a.\overline{b} \mid b.(\overline{a} \mid \overline{c})) \quad \text{with } \tau_1 = ()^{\uparrow_{\mathrm{A}}}$$

   and define $R \stackrel{\text{def}}{=} c.\overline{w}$. Then $\vdash Q_{1,2} \triangleright c : \tau_1$ and $\vdash R \triangleright c : \overline{\tau_1}, w : ()_s^{\uparrow_{\mathrm{A}}}$. Now we can check $Q_1 \mid R \Downarrow_w$ but $Q_2 \mid R \Uparrow$. Hence τ_1 is behaviourally positive. This also shows types of the mode $\downarrow_{\mathrm{A}}$ are behaviourally negative.
2. The types of mode $!_{\mathrm{L}}$ and $!_{\mathrm{A}}$ (other than exceptional cases, see **3.** below) are behaviourally positive. Take $Q_3 \stackrel{\text{def}}{=} !d(c).Q_1$ and $Q_4 \stackrel{\text{def}}{=} !d(c).Q_3$ and $\tau_2 = (()^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{L}}}$. Define $R' \stackrel{\text{def}}{=} \overline{d}(c)c.\overline{w}$. Then $\vdash Q_{3,4} \triangleright d : \tau_2$ and $\vdash R' \triangleright d : \overline{\tau_2}, w : ()_s^{\uparrow_{\mathrm{A}}}$. Now we can

check $Q_3 \mid R' \Downarrow_w$ but $Q_4 \mid R' \Uparrow$. In this way, types of the mode $!_{\mathrm{L}}$ are generally positive and those of $?_{\mathrm{L}}$ are negative. $Q_{3,4}$ are also typable by $(()^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}$, which suggest types of the mode $!_{\mathrm{A}}$ are positive and those of $?_{\mathrm{A}}$ are negative.

3. The types of mode $\uparrow_{\mathrm{L}}$ (again other than exceptional cases) are behaviourally positive. Take $Q_5 \stackrel{\mathrm{def}}{=} \overline{e}(d)Q_3$ and $Q_6 \stackrel{\mathrm{def}}{=} \overline{e}(d)Q_4$ with $\tau_3 = (((()^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}})^{\uparrow_{\mathrm{L}}}$. Then define $R' \stackrel{\mathrm{def}}{=} e(d).\overline{d}(c)c.\overline{w}$. By the same routine as the above cases, we can check τ_3 is behaviourally positive. This example also suggests that $\downarrow_{\mathrm{L}}$ is behaviourally negative. The exceptional cases are $()^{\uparrow_{\mathrm{L}}}$ and those types generated from this type inductively, such as $(\vec{\rho}^{\,?}()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$ for arbitrary $\vec{\rho}$, which are, as $\updownarrow$, neither behaviourally positive nor behaviourally negative.

The examples above show that polarities are not directly related to the distinction between input and output: types of mode $!_{\mathrm{L}}, !_{\mathrm{A}}$ are positive even though they are input. Similarly types of mode $?_{\mathrm{L}}, ?_{\mathrm{A}}$ are negative even though they are output.

For syntactic treatment, it is more convenient to neglect the "exceptional cases" noted in Example 3.2 in (3) above. Thus we set:

Definition 3.3 (syntactic polarities of types) τ is *syntactically positive*, or simply *positive*, iff $\mathsf{md}(\tau) \in \{\uparrow_{\mathrm{A}}, \uparrow_{\mathrm{L}}, !_{\mathrm{L}}, !_{\mathrm{A}}\}$. Dually τ is *syntactically negative*, or simply *negative*, iff $\mathsf{md}(\tau) \in \{\downarrow_{\mathrm{A}}, \downarrow_{\mathrm{L}}, ?_{\mathrm{L}}, ?_{\mathrm{A}}\}$.

3.3 *Basic idea of flows*

We can now illustrate how polarities relate to flows using examples. The first example is a very simple flow.

$$y : ()^{\downarrow_{\mathrm{A}}}_{s} \leadsto x : ()^{\uparrow_{\mathrm{A}}}_{s'} \tag{6}$$

The flow says that information a process receives at y flows down to x. Note that a negative type $()^{\downarrow_{\mathrm{A}}}_{s}$ occurs in its source, while a positive type $()^{\uparrow_{\mathrm{A}}}_{s'}$ occurs in its target. This is obeyed in all flows we shall deal with from now on. A simple process which has this flow is (cf. Example 2.3 (3)):

$$y.\overline{x}. \tag{7}$$

In the process, we can see that information flow indeed goes from a negative channel (y) to a positive channel (x). From the viewpoint of secrecy, the flow in (6) is secure iff $s' \sqsupseteq s$, since, if not, a low-level output depends on a high-level input.

Next we consider a slightly more complex flow.

$$y : (()^{\downarrow_{\mathrm{A}}}_{s})^{?_{\mathrm{A}}} \leadsto x : ()^{\uparrow_{\mathrm{A}}}_{s'} \tag{8}$$

Here the source is another output, though its type is again negative. The flow says that information which comes from a negative output channel y, which asks for information and receives it, flows down to a positive output channel. As a simplest process which embodies this flow, we can take (omitting the typing):

$$\overline{y}(c)c.\overline{x}. \tag{9}$$

As noted, while being outputs, $?_{\mathsf{A}}$-actions receive information rather than emit it. The process above intuitively shows this: it asks at y, receives the "real" information at c, then emits the information at x. Again the flow in (8) is secure iff $s' \sqsupseteq s$.

The third example is a flow that contains a replicated input.

$$y : (()_s^{\downarrow_{\mathsf{A}}})^{?_{\mathsf{A}}} \rightsquigarrow x : (()_{s'}^{\uparrow_{\mathsf{A}}})^{!_{\mathsf{A}}} \tag{10}$$

A replicated input emits information by its action at a carried affine output channel when it is asked (this is precisely dual to $?_{\mathsf{A}}$ in (8) above). A process which represents this flow is a copycat, written $[x \to y]$ in Example 2.1 (3):

$$\vdash_{\mathsf{sec}} !x(c).\overline{y}(e)e.\overline{c} \triangleright x : (()_{s'}^{\uparrow_{\mathsf{A}}})^{!_{\mathsf{A}}},\ y : (()_s^{\downarrow_{\mathsf{A}}})^{?_{\mathsf{A}}}. \tag{11}$$

In this process, each action at y can take place only after there is an input action at x. So, from an intuitive idea of causality, one may as well consider there is a flow from x to y, rather than y to x. However, (10) does capture the information flow of the process precisely: when the process is asked at x with a channel c, it asks back at y, and the information it receives as a result finally flows down to c. Thus the flow from y to x in (10) indicates that information which comes from a negative output channel y goes out when a process is asked at a positive input channel x, even though the latter action should temporarily precede the former action.

3.4 Examples of flow analysis

Basic Flow Inference. We now turn to the extraction of a flow from a typed process. Let us take the copycat in (11) and see the main idea of how to extract the flow (10) from this process (note that this process contains (7) and (9) as its sub-terms; thus the derivation in fact includes the flow inferences for their flows, (6) and (8)).

$$(\mathsf{In}^{!_{\mathsf{A}}})\;\dfrac{(\mathsf{Out}^{?})\;\dfrac{(\mathsf{In}^{\downarrow_{\mathsf{A}}})\;\dfrac{(\mathsf{Out}^{\uparrow_{\mathsf{A}}})\;\dfrac{-}{\vdash \overline{c} \blacktriangleright \emptyset \rightsquigarrow c : ()_{s'}^{\uparrow_{\mathsf{A}}}}}{\vdash e.\overline{c} \blacktriangleright e : ()_s^{\downarrow_{\mathsf{A}}} \rightsquigarrow c : ()_{s'}^{\uparrow_{\mathsf{A}}}}}{\vdash \overline{y}(e)e.\overline{c} \blacktriangleright y : (()_s^{\downarrow_{\mathsf{A}}})^{?_{\mathsf{A}}} \rightsquigarrow c : ()_{s'}^{\uparrow_{\mathsf{A}}}}}{\vdash !x(c).\overline{y}(e)e.\overline{c} \blacktriangleright y : (()_s^{\downarrow_{\mathsf{A}}})^{?_{\mathsf{A}}} \rightsquigarrow x : (()_{s'}^{\uparrow_{\mathsf{A}}})^{!_{\mathsf{A}}}}$$

In the inference above, the names in the left-hand side (such as $(\mathsf{Out}^{\uparrow_{\mathsf{A}}})$) correspond to those of the inference rules given in section 4, Figure 1 later. In the first step, the affine output at c is given without any depending actions, hence the flow contains no sources. When we prefix this output with an affine input e, we record a flow from e to c, as expected (this gives the flow in (6) modulo renaming). In the third step, the prefixing action is the output at y, abstracting e: since a question at y indirectly receives information, y is placed in the source part of the flow set, abstracting e (giving the flow in (8) modulo renaming). In the final line, we prefix the process with the replicated input at x, abstracting c. Again x is the channel via which the "real" information, which exists at c, indirectly comes out, so we put x in the target part, simultaneously abstracting c, reaching the flow (10).

Composition of Flows. The following simple example shows how we can extract information flow from a parallel composition of processes. We already know we can extract the following flow (by exchanging x and e with y and w, respectively, from the third step of the above analysis). We write $\star_s$ for $(()_s^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}$.

$$\vdash \overline{x}(c)c.\overline{w} \blacktriangleright x : \overline{\star_{s'}} \rightsquigarrow w : ()_{s''}^{\uparrow_{\mathrm{A}}}$$

If we compose this process with the copycat in (11), we obtain a typed process:

$$\vdash_{\mathsf{sec}} \ [x \to y] \mid \overline{x}(c)c.\overline{w} \triangleright x : \star_{s'},\ y : \overline{\star_s},\ w : ()_{s''}^{\uparrow_{\mathrm{A}}} \tag{12}$$

We now show how we can extract a flow of this composed process from the flows of the two constituent processes.

$$(\mathsf{Par})\ \frac{\vdash\ [x \to y] \blacktriangleright y : \overline{\star_s} \rightsquigarrow x : \star_{s'} \qquad \vdash \overline{x}(c)c.\overline{w} \blacktriangleright x : \overline{\star_{s'}} \rightsquigarrow w : ()_{s''}^{\uparrow_{\mathrm{A}}}}{\vdash [x \to y] \mid \overline{x}(c)c.\overline{w} \blacktriangleright y : \overline{\star_s} \rightsquigarrow x : \star_{s'},\ y : \overline{\star_s} \rightsquigarrow w : ()_{s''}^{\uparrow_{\mathrm{A}}}}$$

In the inference above, the occurrence of $x : \overline{\star_{s'}}$ in the source part of the flow of the second sequent in the antecedent is replaced by $y : \overline{\star_s}$ in the conclusion. This is because the replicated input complementing an output at x is already supplied, and because we know, by the first part of the antecedent, that this replicated input depends on y. Note that the resulting typed names conform to the resulting action type: x is now a replicated input (is positive), typed as $\star_{s'}$, hence it can only occur in the target part of the resulting flow set.

Treatment of Unused Types. So far, typed names occurring in a flow extracted from a typed process always use the same types as occurring in the action type of that process. Channel types, however, can contain unused parts with respect to the given target, so that their use can lead to inaccuracy. Let us illustrate this point by a simple example.

$$\vdash_{\mathsf{sec}} \ x(y_1 y_2).\overline{y_1}(e)e.\overline{f} \triangleright x : (\overline{\star_{\mathbb{L}}}\ \overline{\star_{\mathbb{H}}})^{\downarrow_{\mathrm{L}}}, f : ()_{\mathbb{L}}^{\uparrow_{\mathrm{A}}} \tag{13}$$

This process is secure: to output a low-level f, it is prefixed by e carried by y_1, but this e is again a low-level, so no insecure flow takes place. If the process also asked at y_2 (via which it should receive a high-level datum) before outputting via f, the process would not have been secure. Thus we should *not* extract the following flow:

$$x : (\overline{\star_{\mathbb{L}}}\ \overline{\star_{\mathbb{H}}})^{\downarrow_{\mathrm{L}}} \rightsquigarrow f : ()_{\mathbb{L}}^{\uparrow_{\mathrm{A}}} \tag{14}$$

(14) indicates an insecure flow, saying a high-level input is used to produce a low-level output, which does not reflect the behaviour of (13). This observation leads to the extraction of a flow from (13) which has the following form:

$$x : (\overline{\star_{\mathbb{L}}} *)^{\downarrow_{\mathrm{L}}} \rightsquigarrow f : ()_{\mathbb{L}}^{\uparrow_{\mathrm{A}}}$$

Here $*$ is a place-holder for a type corresponding to an unused action. By using $*$, the type in the source now captures that the process only asks at the first argument

y_1, not the second one y_2. The extraction of this flow is done as follows.

$$
(\mathsf{In}^{\downarrow_{\mathrm{L}}})\;\dfrac{(\mathsf{Weak})\;\dfrac{(\mathsf{Out}^{?})\;\dfrac{(\mathsf{In}^{\downarrow_{\mathrm{A}}})\;\dfrac{(\mathsf{Out}^{\uparrow_{\mathrm{A}}})\;\dfrac{-}{\vdash \overline{f} \blacktriangleright \emptyset \leadsto f:()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}}}}{\vdash e.\overline{f} \blacktriangleright e:()^{\downarrow_{\mathrm{A}}}_{\mathbb{L}} \leadsto f:()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}}}}{\vdash \overline{y_1}(e)e.\overline{f} \blacktriangleright y_1:\overline{\star_{\mathbb{L}}} \leadsto f:()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}}}}{\vdash \overline{y_1}(e)e.\overline{f} \blacktriangleright y_1:\overline{\star_{\mathbb{L}}}\cdot y_2:* \leadsto f:()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}}}}{\vdash x(y_1y_2).\overline{y_1}(e)e.\overline{f} \blacktriangleright x:(\overline{\star_{\mathbb{L}}}\,*)^{\downarrow_{\mathrm{L}}} \leadsto f:()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}}}
$$

In the final line, the type of x does not contain the type of the channel which is not used (here y_2), but instead records its absence by $*$, so that more precise calculation of the effective level of inputting action becomes possible. On the other hand, if we replace y_1 in (13) by y_2, we derive an insecure flow $x:(*\,\overline{\star_{\mathbb{H}}})^{\downarrow_{\mathrm{L}}} \leadsto f:()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}}$.

4 Flow analysis

4.1 Preliminary definitions

Extended Channel Types. As we discussed in section 3.4, a flow needs to use the "wild card" symbol $*$. For uniformity, we annotate the symbol $*$ with action modes $\{!_{\mathrm{L}}, !_{\mathrm{A}}, ?_{\mathrm{L}}, ?_{\mathrm{A}}\}$.[1] We extend the grammar of channel types with these types:

$$\tau_{\mathrm{I}} \;::=\; \ldots \mid *^{!_{\mathrm{L}}} \mid *^{!_{\mathrm{A}}} \qquad \tau_{\mathrm{O}} \;::=\; \ldots \mid *^{?_{\mathrm{L}}} \mid *^{?_{\mathrm{A}}}$$

We usually omit action mode p from $*^p$, writing simply $*$. We call the resulting set of types, *extended channel types*, or just *types* for simplicity. The notions of flows and flow sets which use extended channel types are defined precisely as given in section 3.1. On the extended channel types we define:

Definition 4.1 $\prec$ is given as the smallest precongruence on extended channel types including $*^p \prec \tau^p$ for each τ. $\Gamma_1 \prec \Gamma_2$ is defined as: $\forall x \in \mathtt{dom}(\Gamma_1).\ \Gamma_1(x) \prec \Gamma_2(x)$. Further we write $\mathsf{F} \prec A$ when for each $x:\tau$ in F, $x:\tau' \in A$ and $\tau \prec \tau'$ for some τ'. We write $\tau_1 \asymp \tau_2$ when they have a common upper bound with respect to $\prec$.

Note $\tau \prec \tau'$ essentially means τ is the result of replacing non-$*$-types in τ' with $*$ in zero or more places. Thus we immediately conclude:

Proposition 4.2 *$\prec$ is a partial order. Further $\tau_1 \asymp \tau_2$ implies there is the join of $\tau_{1,2}$.*

We can easily check $\vee$ is commutative and associative.

Composition of Flow Sets. We next introduce (parallel) composition of two flow sets. First we define the operator $\mathsf{F}_1 \asymp \mathsf{F}_2$ as follows (note that $A_1 \asymp A_2$ means two types are composable as defined in Appendix B).

[1] While we seldom – if ever – use the action mode associated with $*$, having it is convenient since it allows us to use this construct just as other types in various definitions.

Definition 4.3 Assume $A_1 \asymp A_2$ and $\mathsf{F}_i \prec A_i$ $(i = 1, 2)$. Then $\mathsf{F}_1^{A_1} \asymp \mathsf{F}_2^{A_2}$, which we usually write $\mathsf{F}_1 \asymp \mathsf{F}_2$ leaving $A_{1,2}$ implicit, iff the following conditions hold: For each x occurring in both F_1 and F_2:

i. If $A_2(x) = \overline{A_1(x)}$ and $x : \tau$ occurs as a source in F_1, then $x : \delta$ occurs in F_2 as a target such that $\tau \prec \overline{\delta}$.
ii. If $A_1(x) = A_2(x)$ and if $x : \tau_i$ occurs in F_i $(i = 1, 2)$, then $\tau_1 \asymp \tau_2$.

A_1 and A_2 are called the *underlying typings* of F_1 and F_2, respectively.

In practice (for example when we work with the encoding of a programming language), it almost always suffices to assume the strict duality for compensating types in (i) above, replacing "$\tau \prec \overline{\delta}$" with "$\tau = \overline{\delta}$."

When defining composition of flow sets, it is convenient to use a decomposition of a flow into its "prime" elements.

Definition 4.4 Given F, a flow $\Gamma_0 \rightsquigarrow x : \tau$ is a *prime flow in* F when $\Gamma \rightsquigarrow x : \tau$ is in F such that either (1) $\Gamma_0 = \{y : \rho\} \subset \Gamma$ or (2) $\Gamma_0 = \emptyset = \Gamma$. We write $\mathsf{F} \vdash \Gamma_0 \rightsquigarrow x : \tau$ if $\Gamma_0 \rightsquigarrow x : \tau$ is a prime flow in F.

Proposition 4.5 *Write* $|\mathsf{F}|$ *for the set of all prime flows in* F. *Then* $|\mathsf{F}_1| = |\mathsf{F}_2|$ *iff* $\mathsf{F}_1 = \mathsf{F}_2$.

Now assume $\mathsf{F}_1^{A_1} \asymp \mathsf{F}_2^{A_2}$. Then a *composed prime flow* of $\mathsf{F}_{1,2}$ is a sequence of prime flows, each being either in F_1 or in F_2 alternately, of the shape:

$$x_0 : \tau_0 \rightsquigarrow x_1 : \tau_1, \quad x_1 : \tau_1' \rightsquigarrow x_2 : \tau_2, \quad \ldots, \quad x_{n-1} : \tau_{n-1}' \rightsquigarrow x_n : \tau_n \quad (n \geqslant 1)$$

where (1) each x_i is distinct and (2) τ_0 and τ_n have the same modes as $A_1 \odot A_2$, i.e. $\tau_0 \prec (A_1 \odot A_2)(x_0)$ and $\tau_n \prec (A_1 \odot A_2)(x_n)$. Above x_0 and x_n are respectively the *source* and *target* of the composed prime flow with *assigned types* τ_0 and τ_n.

We now define the composition of flow sets, $\mathsf{F}_1 \odot \mathsf{F}_2$.

Definition 4.6 (composition of flow sets) Assume $\mathsf{F}_{1,2}$ such that $\mathsf{F}_1 \asymp \mathsf{F}_2$, with the underlying action types A_1 and A_2. Then $\mathsf{F}_1 \odot \mathsf{F}_2$ is given as follows:

i. Let each composed prime flow with source y and target x assign τ_i to y and τ to x, respectively. Then $\mathsf{F}_1 \odot \mathsf{F}_2 \vdash y : \vee_i \tau_i \rightsquigarrow x : \tau$.
ii. If $\mathsf{F}_i \vdash \Gamma \rightsquigarrow x : \tau$ with either $i = 1$ or $i = 2$, but there is no composed prime flow whose target is x, then we set $\mathsf{F}_1 \odot \mathsf{F}_2 \vdash \emptyset \rightsquigarrow x : \tau$.

Example 4.7 (composition of flow sets)

1. Recall $y.\overline{x}$ which has a flow $\mathsf{F}_1 \stackrel{\text{def}}{=} y : ()_s^{\downarrow\mathrm{A}} \rightsquigarrow x : ()_{s'}^{\uparrow\mathrm{A}}$ in (6) in section 3. Similarly $x.\overline{w}$ has a flow $\mathsf{F}_2 \stackrel{\text{def}}{=} x : ()_{s'}^{\downarrow\mathrm{A}} \rightsquigarrow w : ()_{s''}^{\uparrow\mathrm{A}}$. Then we have $\mathsf{F}_1 \asymp \mathsf{F}_2$ and $\mathsf{F}_1 \odot \mathsf{F}_2 = y : ()_s^{\downarrow\mathrm{A}} \rightsquigarrow w : ()_{s''}^{\uparrow\mathrm{A}}$.
2. Recall a copycat $[x \to y]$ which has a flow $y : \star_s \rightsquigarrow x : \overline{\star_{s'}}$ (cf. (10) in section 3). Then $[x \to y] \mid [y \to x]$ has a flow $\{\emptyset \rightsquigarrow x : \star_s,\ \emptyset \rightsquigarrow y : \star_s\}$. Definition 4.6 (ii) can also treat a composition of two flows which creates circular behaviour, as

Ω_x in Example 2.3 (3), which is given a flow $\emptyset \rightsquigarrow x : \star_s$. Note the resulting flow itself is always non-circular.

3. Recall the process (13) in section 3.4. We define $R_i \stackrel{\text{def}}{=} x(y_1 y_2).\overline{y_i}(e)e.\overline{f}$. Let $\tau_1 \stackrel{\text{def}}{=} (\overline{\star_{s_1}}\, *)^{\downarrow \mathrm{L}}$ and $\tau_2 \stackrel{\text{def}}{=} (*\, \overline{\star_{s_2}})^{\downarrow \mathrm{L}}$. Then R_i has a flow $x : \tau_i \rightsquigarrow f : ()_{s_i}^{\uparrow \mathrm{A}}$. Further $W_i \stackrel{\text{def}}{=} !w_i(f).\overline{y}(x)R_i$ has a flow $y : (\tau_i)^{?\mathrm{L}} \rightsquigarrow w_i : (()_{s_i}^{\uparrow \mathrm{A}})^{!\mathrm{L}}$. Now suppose the dual process $Q \stackrel{\text{def}}{=} \overline{w_1}(c_1)c_1.\overline{w_2}(c_2)c_2.\overline{f}$ whose flow is $\{w_1 : (()_{s_1}^{\downarrow \mathrm{A}})^{?\mathrm{L}} \cdot w_2 : (()_{s_2}^{\downarrow \mathrm{A}})^{?\mathrm{L}}\} \rightsquigarrow f : ()_s^{\uparrow \mathrm{A}}$. Then composition of these processes has a flow:
$$\vdash Q \mid W_1 \mid W_2 \blacktriangleright y : (\tau_1 \vee \tau_2)^{?\mathrm{L}} \rightsquigarrow f : ()_s^{\uparrow \mathrm{A}}$$
with $\tau_1 \vee \tau_2 = (\overline{\star_{s_1}}\, \overline{\star_{s_2}})^{\downarrow \mathrm{L}}$. Note this flow is secure iff $s_1 \sqcup s_2 \sqsubseteq s$.

Remark 4.8 (on composition of flow sets)

- In Definition 4.6 (i), $\vee \rho_i$ is well-defined since $\rho_i \prec (A_1 \odot A_2)(y)$ for each i and by Proposition 4.2.
- By Definition 4.6 (i) and (ii), as well as Definition 4.3, if F_1 and F_2 only have positive ($!$ and $\uparrow$) channels as targets and negative ($?$ and $\downarrow$) ones as sources, then the same is also true in $\mathsf{F}_1 \odot \mathsf{F}_2$.

4.2 Inductive flow analysis

We introduce the inference rules for the flow analysis one by one. There are several *prefix rules* and several *composition rules*, as well as additional *structural rules*. Prefix rules and composition rules are in direct correspondence with the typing rules in Appendix B. We assume a process in the conclusion of each rule is well-typed. This allows us to leave out most of the typing information from the inference rules. Figure 1 summarises all inference rules for the flow analysis.

Composition Rules. There are four composition rules in correspondence with the typing rules.

$$\text{(Zero)}\ \frac{-}{\vdash \mathbf{0} \blacktriangleright \emptyset} \qquad \text{(Par)}\ \frac{\vdash P_i \blacktriangleright \mathsf{F}_i\ (i=1,2) \quad \mathsf{F}_1 \asymp \mathsf{F}_2}{\vdash P_1|P_2 \blacktriangleright \mathsf{F}_1 \odot \mathsf{F}_2} \qquad \text{(Res)}\ \frac{\vdash P \blacktriangleright \mathsf{F} \quad x \notin \mathsf{fn}(\mathsf{F})}{\vdash (\boldsymbol{\nu}\, x)P \blacktriangleright \mathsf{F}} \qquad \text{(Weak)}\ \frac{\mathsf{md}(\tau) \in \mathcal{M}_? \quad \vdash P^A \blacktriangleright \Gamma \rightsquigarrow w : \sigma \quad x \notin \Gamma}{\vdash P^{A,x:\tau} \blacktriangleright \Gamma \cdot x : * \rightsquigarrow w : \sigma}$$

(Zero) does not record any information flow since there is none. (Par) connects causal chains in accordance with composition of the underlying action type. The composition generally needs more than one causal map for compensating multiple sources of even a single flow. In (Res) we restrict the name which does not occur in the flow (note that, since $(\boldsymbol{\nu}\, x)P$ is well-typed, x in the rule should have either $!$-mode or $\updownarrow$-mode). Finally, in (Weak), a weakened $?$-mode name can be given a trivial $*$ type (we omit the rule for the weakening of a name with the mode $\updownarrow$ in which case, since $\updownarrow$-channels never occur in flows, no change takes place in the flow set). Note this rule only treats a singleton flow set: this does not lose generality due to the structural rules, which we discuss next.

$$\text{(Zero)}\ \frac{-}{\vdash \mathbf{0} \blacktriangleright \emptyset} \qquad \text{(Par)}\ \frac{\vdash P_i \blacktriangleright \mathsf{F}_i\ (i=1,2) \quad \mathsf{F}_1 \asymp \mathsf{F}_2}{\vdash P_1|P_2 \blacktriangleright \mathsf{F}_1 \odot \mathsf{F}_2} \qquad \text{(Res)}\ x \notin \mathsf{fn}(\mathsf{F})\ \frac{\vdash P \blacktriangleright \mathsf{F}}{\vdash (\boldsymbol{\nu}\, x)P \blacktriangleright \mathsf{F}} \qquad \text{(Weak)}\ \mathsf{md}(\tau) \in \{?_{\mathsf{L}}, ?_{\mathsf{A}}\}\ \frac{\vdash P^{A} \blacktriangleright \Gamma \rightsquigarrow w : \sigma}{\vdash P^{A\cdot x:\tau} \blacktriangleright \Gamma \cdot x : * \rightsquigarrow w : \sigma}$$

$$\text{(Union)}\ \frac{\forall i.\ \vdash P \blacktriangleright \mathsf{F}_i}{\vdash P \blacktriangleright \bigcup_i \mathsf{F}_i} \qquad \text{(Subset)}\ \frac{\vdash P \blacktriangleright \mathsf{F},\ \Gamma \rightsquigarrow x : \tau}{\vdash P \blacktriangleright \mathsf{F}} \qquad \text{(Empty)}\ \frac{A(x)\ \text{positive},\ A(\vec{y})\ \text{negative}}{\vdash P^{A} \blacktriangleright \vec{y} : \vec{*} \rightsquigarrow x : *}$$

$$(\mathsf{In}^{\downarrow_{\mathsf{L}}})\ \frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\rho} \rightsquigarrow w : \sigma}{\vdash x(\vec{y}).P \blacktriangleright \Gamma \cdot x : (\vec{\rho})^{\downarrow_{\mathsf{L}}} \rightsquigarrow w : \sigma} \qquad (\mathsf{Out}^{\uparrow_{\mathsf{L}}})\ \frac{\forall i.\ \vdash P \blacktriangleright \Gamma_i \rightsquigarrow y_i : \rho_i}{\vdash \overline{x}(\vec{y})P \blacktriangleright \ \vee_i \Gamma_i \rightsquigarrow x : (\vec{\rho})^{\uparrow_{\mathsf{L}}}}$$

$$(\mathsf{In}^{\downarrow_{\mathsf{A}}})\ \frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\rho} \rightsquigarrow w : \sigma}{\vdash x(\vec{y}).P \blacktriangleright \Gamma \cdot x : (\vec{\rho})_s^{\downarrow_{\mathsf{A}}} \rightsquigarrow w : \sigma} \qquad (\mathsf{Out}^{\uparrow_{\mathsf{A}}})\ \frac{\forall i.\ \vdash P \blacktriangleright \Gamma_i \rightsquigarrow y_i : \rho_i}{\vdash \overline{x}(\vec{y})P \blacktriangleright \ \vee_i \Gamma_i \rightsquigarrow x : (\vec{\rho})_s^{\uparrow_{\mathsf{A}}}}$$

$$(\mathsf{In}^{!})\ (p \in \mathcal{M}_{!})\ \frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\rho} \rightsquigarrow z : \sigma}{\vdash !\, x(\vec{y}z).P \blacktriangleright \Gamma \rightsquigarrow x : (\vec{\rho}\, \sigma)^{p}} \qquad (\mathsf{Out}^{?})\ (p \in \mathcal{M}_{?})\ \frac{\vdash P \blacktriangleright \bigcup_i \{\Gamma_i \rightsquigarrow y_i : \tau_i\},\ z : \rho \cdot \Gamma \rightsquigarrow w : \sigma}{\vdash \overline{x}(\vec{y}z)P \blacktriangleright \ \vee_i \Gamma_i \vee \Gamma \vee x : (\vec{\tau}\, \rho)^{p} \rightsquigarrow w : \sigma}$$

Fig. 1. Flow analysis for π^{LA}.

Structural Rules. Structural rules are in close connection with (Par).

$$\text{(Union)}\ \frac{\forall i.\ \vdash P \blacktriangleright \mathsf{F}_i}{\vdash P \blacktriangleright \cup_i \mathsf{F}_i} \qquad \text{(Subset)}\ \frac{\vdash P \blacktriangleright \mathsf{F},\ \Gamma \rightsquigarrow x : \tau}{\vdash P \blacktriangleright \mathsf{F}} \qquad \text{(Empty)}\ \frac{A(x)\ \text{positive},\ A(y_i)\ \text{negative}}{\vdash P^{A} \blacktriangleright \vec{y} : \vec{*} \rightsquigarrow x : *}$$

The need for these rules comes from the following reason: (1) as noted above, (Par) in general uses multiple flows for compensating source channels; and (2) in spite of (1), prefix rules infer only a single flow, as we shall see below. Thus we need a rule to collect individual flows into a flow set, which is done by (Union). After (Par) rule is applied, we may have to single out one flow (to which, for example, we can apply a prefix rule). For this we use (Subset). Note that, by repeatedly applying (Subset), we can always infer an empty flow set from a process. Finally (Empty) adds an ineffective flow (which can later be used in prefix rules).

Prefix Rules. We start with the linear input/output rules.

$$(\mathsf{In}^{\downarrow_{\mathsf{L}}})\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\rho} \rightsquigarrow w : \sigma}{\vdash x(\vec{y}).P \blacktriangleright \Gamma \cdot x : (\vec{\rho})^{\downarrow_{\mathsf{L}}} \rightsquigarrow w : \sigma} \qquad (\mathsf{Out}^{\uparrow_{\mathsf{L}}})\frac{\forall i.\ \vdash P \blacktriangleright \Gamma_i \rightsquigarrow y_i : \rho_i}{\vdash \overline{x}(\vec{y})P \blacktriangleright \ \vee_i \Gamma_i \rightsquigarrow x : (\vec{\rho})^{\uparrow_{\mathsf{L}}}}$$

As noted, each prefix rule only treats a single flow. In ($\mathsf{In}^{\downarrow_{\mathsf{L}}}$) we abstract the carried types used for producing the result: some ρ_i can be weakened by (Weak) and (Empty). (Out$^{\uparrow\mathrm{L}}$) is its exact dual, recording the flow information for part of its abstracted names, leaving out some of y_i for which we choose to use $*$. $\vee_i\Gamma_i$ takes the union of $\{\Gamma_i\}$, with the unification of types for coinciding names using $\vee$. The duality between (In$^{\downarrow\mathrm{L}}$) and (Out$^{\uparrow\mathrm{L}}$) in their treatment of $*$ in carried types is essential for a sound notion of composition. For example, from the following flows of two copycats (the right-hand side uses (Empty) above):

$$\vdash [y_1 \to w_1] \blacktriangleright w_1 : \overline{\star_{\mathbb{L}}} \rightsquigarrow y_1 : \star_{\mathbb{L}} \qquad \vdash [y_2 \to w_2] \blacktriangleright \emptyset \rightsquigarrow y_2 : *$$

we can infer, using (Out$^{\uparrow\mathrm{L}}$):

$$\vdash \overline{x}(y_1 y_2)([y_1 \to w_1] \mid [y_2 \to w_2]) \blacktriangleright w_1 : \overline{\star_{\mathbb{L}}} \rightsquigarrow x : (\star_{\mathbb{L}}\ *)^{\uparrow\mathrm{L}} \tag{15}$$

This can be composed with:

$$\vdash x(y_1 y_2).\overline{y_1}(c)c.\overline{f} \blacktriangleright x : (\overline{\star_{\mathbb{L}}}\ *)^{\downarrow\mathrm{L}} \rightsquigarrow f : ()^{\uparrow\mathrm{A}}_{\mathbb{L}}$$

so that we obtain:

$$\vdash x(y_1 y_2).\overline{y_1}(c)c.\overline{f} \mid \overline{x}(y_1 y_2)([y_1 \to w_1] | [y_2 \to w_2]) \blacktriangleright w_1 : \overline{\star_{\mathbb{L}}} \rightsquigarrow f : ()^{\uparrow\mathrm{A}}_{\mathbb{L}} \tag{16}$$

which indicates a safe flow. Note if we cannot neglect y_2 in (15), we would have gotten a flow from w_2, resulting in an insecure flow unnecessarily.

The next prefix rules are for affine input/output. These rules are the precise analogue of (In$^{\downarrow\mathrm{L}}$) and (Out$^{\uparrow\mathrm{L}}$) except for the added secrecy levels.

$$(\mathsf{In}^{\downarrow\mathrm{A}})\ \frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\rho} \rightsquigarrow w : \sigma}{\vdash x(\vec{y}).P \blacktriangleright \Gamma \cdot x : (\vec{\rho})^{\downarrow\mathrm{A}}_{s} \rightsquigarrow w : \sigma} \qquad (\mathsf{Out}^{\uparrow\mathrm{A}})\ \frac{\forall i.\ \vdash P \blacktriangleright \Gamma_i \rightsquigarrow y_i : \rho_i}{\vdash \overline{x}(\vec{y})P \blacktriangleright \vee_i \Gamma_i \rightsquigarrow x : (\vec{\rho})^{\uparrow\mathrm{A}}_{s}}$$

While the tampering level of $(\vec{\tau})^{\uparrow\mathrm{L}}_{s}$ is always s regardless of the carried types $\vec{\tau}$, a carried type becomes sometimes significant in flow extraction. As an example, consider

$$P_1 \stackrel{\mathrm{def}}{=} \overline{x}^{\mathbb{L}}(c)!c(e).\overline{y}(g)g^{\mathbb{H}}.\overline{e}^{\mathbb{M}} \qquad\qquad P_2 \stackrel{\mathrm{def}}{=} x^{\mathbb{L}}(c).\overline{c}(e)e^{\mathbb{M}}.\overline{w}^{\mathbb{L}}$$

where affine channels are annotated with secrecy levels, with $\mathbb{M}$ being a level between $\mathbb{H}$ and $\mathbb{L}$. Then $P \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, x)(P_1 \mid P_2)$ has an insecure flow from y to w, which is extracted as follows. Firstly, we infer for P_1 and P_2 using these two affine rules:

$$(\mathsf{Out}^{\uparrow\mathrm{A}})\ \frac{\vdash !c(e).\overline{y}(g)g^{\mathbb{H}}.\overline{e}^{\mathbb{M}} \blacktriangleright y : \overline{\star_{\mathbb{H}}} \rightsquigarrow c : \star_{\mathbb{M}}}{\vdash \overline{x}^{\mathbb{L}}(c)!c(e).\overline{y}(g)g^{\mathbb{H}}.\overline{e}^{\mathbb{M}} \blacktriangleright y : \overline{\star_{\mathbb{H}}} \rightsquigarrow x : (\star_{\mathbb{M}})^{\uparrow\mathrm{A}}_{\mathbb{L}}}$$

$$(\mathsf{In}^{\downarrow\mathrm{A}})\ \frac{\vdash \overline{c}(e)e^{\mathbb{M}}.\overline{w}^{\mathbb{L}} \blacktriangleright c : \overline{\star_{\mathbb{M}}} \rightsquigarrow w : ()^{\uparrow\mathrm{A}}_{\mathbb{M}}}{\vdash x^{\mathbb{L}}(c).\overline{c}(e)e^{\mathbb{M}}.\overline{w}^{\mathbb{L}} \blacktriangleright x : (\overline{\star_{\mathbb{M}}})^{\downarrow\mathrm{A}}_{\mathbb{L}} \rightsquigarrow w : ()^{\uparrow\mathrm{A}}_{\mathbb{M}}}$$

By (Par), we obtain an insecure flow $y : \overline{\star_{\mathbb{H}}} \rightsquigarrow w : ()^{\uparrow\mathrm{A}}_{\mathbb{M}}$.

Finally, we list the rules for replicated input/output. Since linear/affine distinction does not make a difference in these rules, we list one rule for replicated input and one rule for replicated output.

$$(\mathsf{In}^{!})\quad (p \in \mathcal{M}_{!}) \qquad\qquad (\mathsf{Out}^{?})\quad (p \in \mathcal{M}_{?})$$

$$\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\rho} \rightsquigarrow z : \sigma}{\vdash !\, x(\vec{y}z).P \blacktriangleright \Gamma \rightsquigarrow x : (\vec{\rho}\,\sigma)^{p}} \qquad \frac{\vdash P \blacktriangleright \bigcup_i\{\Gamma_i \rightsquigarrow y_i : \tau_i\},\ z : \rho \cdot \Gamma \rightsquigarrow w : \sigma}{\vdash \overline{x}(\vec{y}z)P \blacktriangleright \vee \Gamma_i \vee \Gamma \vee x : (\vec{\tau}\,\rho)^{p} \rightsquigarrow w : \sigma}$$

(In$^{!}$) is similar to (In$^{\downarrow_{\mathrm{L}}}$), except that we record the resulting type $(\vec{\rho}\sigma)^p$ in the positive position. (Out$^{?}$) is its dual. Note (In$^{!}$) (resp. (Out$^{?}$)) abstracts negative (resp. positive) channels, even though the newly introduced type is positive (resp. negative). A detailed explanation of this point is found in Remark 4.11 later.

A couple of basic properties of the flow analysis are worth noting. Both properties are by easy induction.

Proposition 4.9

1. *If* $\vdash P^A \blacktriangleright \Gamma \rightsquigarrow x:\tau$ *then* τ *is positive and, for each* $y:\rho \in \Gamma$, ρ *is negative.*
2. *If* $\vdash P^A \blacktriangleright \Gamma \rightsquigarrow x:\tau$ *for some* Γ, *then there exists* Γ_0 *such that* $\vdash P^A \blacktriangleright \Gamma_0 \rightsquigarrow x:\tau$ *and, moreover, whenever* $\vdash P^A \blacktriangleright \Gamma' \rightsquigarrow x:\tau$, *we have* $\Gamma_0 \prec \Gamma'$.

Proposition 4.10 *If* $\vdash P \blacktriangleright \Gamma \rightsquigarrow x:\tau$ *and* $P \longrightarrow P'$ *then* $\vdash P' \blacktriangleright \Gamma \rightsquigarrow x:\tau$.

In the proof below $P \xrightarrow{l} P'$ is the standard (bound name passing) labelled transition, whose formal definition is left to Appendix F.

Proof
By easy induction, the proof system is closed under the structural equality $\equiv$. Since $\xrightarrow{\tau}\equiv\ =\ \longrightarrow$ ($\xrightarrow{\tau}$ is the τ-transition) it suffices to show that $\vdash P \blacktriangleright \Gamma \rightsquigarrow x:\tau$ and $P \xrightarrow{\tau} P'$ imply $\vdash P' \blacktriangleright \Gamma \rightsquigarrow x:\tau$. Permuting back all (Weak) to just after each (Zero), we can work with the system which precisely follows the syntax except for the structural rules. We can then show each visible transition changes the flow precisely following the prefix rules. For example, if we have $x(\vec{y}).P \xrightarrow{x(\vec{y})} P$, assuming x is linearly typed, we can infer:

$$\vdash x(\vec{y}).P \blacktriangleright \Gamma\cdot x:(\vec{\rho})^{\downarrow_{\mathrm{L}}} \rightsquigarrow w:\sigma \qquad \text{and} \qquad \vdash P \blacktriangleright \Gamma\cdot \vec{y}:\vec{\rho} \rightsquigarrow w:\sigma.$$

Assume that $P \xrightarrow{x(\vec{y})} P'$. Then this implies:

$$\vdash P \blacktriangleright \Gamma\cdot x:(\vec{\rho})^{\downarrow_{\mathrm{L}}} \rightsquigarrow w:\sigma \qquad \text{and} \qquad \vdash P' \blacktriangleright \Gamma\cdot \vec{y}:\vec{\rho} \rightsquigarrow w:\sigma,$$

Dually we can show $Q \xrightarrow{\overline{x}(\vec{y})} Q'$ implies $\vdash Q \blacktriangleright \Delta \rightsquigarrow x:(\vec{\rho})^{\uparrow_{\mathrm{L}}}$ and, for each y_i, either $\vdash Q' \blacktriangleright \Delta_i \rightsquigarrow y_i:\rho_i$ or $\rho_i = *$ and $\Delta_i = \emptyset$, such that $\Delta = \vee\Delta_i$. Note that, by typing, σ has mode $\uparrow_{\mathrm{L}}$ or $\uparrow_{\mathrm{A}}$, while types in Δ have mode $?_{\mathrm{L}}$ or $?_{\mathrm{A}}$. Now assume $P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu}\,\vec{y})(P' \mid Q')$ from these transitions. Then we have:

$$\vdash P \mid Q \blacktriangleright \Gamma \vee \Delta \rightsquigarrow w:\sigma \quad \text{and} \quad \vdash P' \mid Q' \blacktriangleright \Gamma\vee\Delta \rightsquigarrow w:\sigma,\ \cup_i\Delta_i \rightsquigarrow y_i:\rho_i$$

Then an application of (Res) to the latter gives us

$$\vdash (\boldsymbol{\nu}\,\vec{y})(P' \mid Q') \blacktriangleright \Gamma\vee\Delta \rightsquigarrow w:\sigma$$

as required. Other cases are similar. □

Example 4.11 We illustrate the rules (In$^{!}$) and (Out$^{?}$), focussing on how each rule needs to abstract types with the opposite polarity. Let us consider the following process (we annotate channels with $\mathbb{H}$ and $\mathbb{L}$ and assume all types are affine).

$$P \stackrel{\text{def}}{=} !x(uz).\overline{u}(c)c^{\mathbb{H}}.\overline{z}^{\mathbb{L}} \tag{17}$$

Clearly, this process is insecure: upon invocation, it asks at its first argument u, from which it receives, at c, a high-level effect: this effect is then transmitted to a low-level effect at z, the second argument of the initial invocation. Let us compose P with the following agent, to make explicit this insecurity.

$$Q \stackrel{\text{def}}{=} \overline{x}(uz)([u^{\mathbb{H}} \to v^{\mathbb{H}}]|z^{\mathbb{L}}.\overline{w}^{\mathbb{L}}) \tag{18}$$

In the composite process $P|Q$, Q asks at x, then gets asked by P in return. Q relays this question to v which is an interface with the outside environment. If it receives an answer, Q replies to P, which then returns, via z, an answer to the initial question; which Q transmits to the outside via w. Thus a high-level effect at v is transmitted to a low-level effect at w. This unsafe flow is due to interaction between P and Q. If Q interacts with a different process, the resulting flow can be completely secure. Indeed, take the following process:

$$P' \stackrel{\text{def}}{=} !x(uz).\overline{z}^{\mathbb{L}} \tag{19}$$

Then there is no insecure flow from v to w in $P'|Q$ (note that u is never invoked by P'). We can analyse the difference between $P|Q$ and $P'|Q$ by extracting from them the following two pairs of mutually compatible flows. For P and Q, we obtain:

$$\vdash P \blacktriangleright \emptyset \rightsquigarrow x : (\overline{\star_{\mathbb{H}}}\,()_{\mathbb{L}}^{\uparrow\mathrm{A}})^{!_{\mathrm{A}}}, \qquad \vdash Q \blacktriangleright x : (\star_{\mathbb{H}}\,()_{\mathbb{L}}^{\downarrow\mathrm{A}})^{?_{\mathrm{A}}} \cdot v : \overline{\star_{\mathbb{H}}} \rightsquigarrow w : ()_{\mathbb{L}}^{\uparrow\mathrm{A}}. \tag{20}$$

Whereas, for P' and Q, we have:

$$\vdash P' \blacktriangleright \emptyset \rightsquigarrow x : (*\,()_{\mathbb{L}}^{\uparrow\mathrm{A}})^{!_{\mathrm{A}}}, \qquad \vdash Q \blacktriangleright x : (*\,()_{\mathbb{L}}^{\downarrow\mathrm{A}})^{?_{\mathrm{A}}} \rightsquigarrow w : ()_{\mathbb{L}}^{\uparrow\mathrm{A}}. \tag{21}$$

Observe the difference in the flow of P in (20) and that of P' in (21), which is reflected in the source part of the flow of Q in (20) and the one of Q in (21). It is instructive to see how these flows are inferred. For Q in (20):

$$(\mathsf{Out}^?)\ \frac{\vdash [u^{\mathbb{H}} \to v^{\mathbb{H}}] \mid z.\overline{w}^{\mathbb{L}} \blacktriangleright v : \overline{\star_{\mathbb{H}}} \rightsquigarrow u : \star_{\mathbb{H}},\ \ z : ()_{\mathbb{L}}^{\downarrow\mathrm{L}} \rightsquigarrow w : ()_{\mathbb{L}}^{\uparrow\mathrm{A}}}{\vdash \overline{x}(uz)([u^{\mathbb{H}} \to v^{\mathbb{H}}]|z.\overline{w}^{\mathbb{L}}) \blacktriangleright x : (\star_{\mathbb{H}}\,()_{\mathbb{L}}^{\downarrow\mathrm{A}})^{?_{\mathrm{A}}} \cdot v : \overline{\star_{\mathbb{H}}} \rightsquigarrow w : ()_{\mathbb{L}}^{\uparrow\mathrm{A}}}$$

which assumes the environment would invoke at u to return to the initial question by Q at x. For Q in (21), we infer:

$$(\mathsf{Out}^?)\ \frac{\vdash [u^{\mathbb{H}} \to v^{\mathbb{H}}] \mid z.\overline{w}^{\mathbb{L}} \blacktriangleright \emptyset \rightsquigarrow u : *,\ \ z : ()_{\mathbb{L}}^{\downarrow\mathrm{L}} \rightsquigarrow w : ()_{\mathbb{L}}^{\uparrow\mathrm{A}}}{\vdash \overline{x}(uz)([u^{\mathbb{H}} \to v^{\mathbb{H}}]|z.\overline{w}^{\mathbb{L}}) \blacktriangleright x : (*\,()_{\mathbb{L}}^{\downarrow\mathrm{A}})^{?_{\mathrm{A}}} \rightsquigarrow w : ()_{\mathbb{L}}^{\uparrow\mathrm{A}}}$$

Here, in contrast to the first inference, we assume u is never invoked: we use (Empty) to introduce a vacuous flow to u in the antecedent. Thus the derived flow does not include the high-level source v, resulting in a safe flow. This example shows how the flow analysis adds precision by tacitly using the assumption on the environment.

5 Noninterference via flow analysis

In this section, we show the noninterference result for securely typed processes via the inductive flow analysis introduced in the previous section. We first state the non-interference theorem. Below $P \Downarrow_x$ is defined before Definition 3.1, Page 303.

Theorem 5.1 (non-interference for securely typed processes) *Let* $\vdash_{\mathsf{sec}} R_{1,2} \triangleright A$ *such that* $\mathsf{tamp}(A) \not\sqsubseteq s$ *and let* $\vdash_{\mathsf{sec}} P \triangleright \overline{A}, x:()^{\uparrow_\mathrm{A}}_s$. *Then* $P \mid R_1 \Downarrow_x$ *iff* $P \mid R_2 \Downarrow_x$.

This theorem essentially says that high-level data never interferes with low-level observable behaviour; in other words, it says that two processes with a tamper level strictly higher than (or incompatible with) s, must always give the same s-level observation.

The proof of this theorem is divided into the following two steps.

Step 1: Demonstration of the *syntactic soundness*, which states that only secure flows are inferable from securely typed processes.

Step 2: Demonstration of the *behavioural soundness*, which states that processes from which only secure flows are inferable satisfy noninterference.

More concretely we prove the following main statement for each step.

Step 1: If $\vdash_{\mathsf{sec}} R \triangleright A$ and $\vdash R \blacktriangleright \mathsf{F}$, then F is secure.

Step 2: Suppose $\vdash_{\mathsf{sec}} P \triangleright \overline{A}, x : ()^{\uparrow_\mathrm{A}}_s$. If $\vdash P \blacktriangleright \mathsf{F}$ always implies F is secure, then for all $\vdash_{\mathsf{sec}} R_{1,2} \triangleright A$ such that $\mathsf{tamp}(A) \not\sqsubseteq s$, $P \mid R_1 \Downarrow_x$ iff $P \mid R_2 \Downarrow_x$.

Step 1 and 2 are formally stated and proved in Propositions 5.10 and 5.15, respectively. The main appeal of this two-step method is that Step 2 is independent from individual secrecy analyses and can be done once and for all, while Step 1 can be verified by simple induction on the typing rules.

To relate secrecy analysis and causality analysis, we use the following dual notion of the tampering level, called *receiving level* (in the sense that it is the level at which the process may receive information from the environment).

Definition 5.2 (receiving level) Given τ of mode **?** or $\downarrow$, the *receiving level of* τ, denoted $\mathsf{receive}(\tau)$, is given inductively as follows:

$$\begin{array}{rclrcl} \mathsf{receive}(*) & = & \mathbb{L} & \mathsf{receive}((\vec{\tau})^{\downarrow_\mathrm{A}}_s) & = & s \\ \mathsf{receive}((\vec{\tau}\rho^{\downarrow})^{?}) & = & \mathsf{receive}(\rho) & \mathsf{receive}((\vec{\tau})^{\downarrow_\mathrm{L}}) & = & \sqcup_i\{\mathsf{receive}(\tau_i)\} \end{array}$$

Then $\mathsf{receive}(\Gamma) = \sqcup_i\{\mathsf{receive}(\tau_i) \mid \exists x.\,\Gamma(x) = \tau_i^{?,\downarrow}\}$.

The tamper level of an extended channel type τ is as given in section 2.3 defined as before, except for adding the clause:

$$\mathsf{tamp}(*) = \mathbb{H}$$

Note $\mathsf{tamp}(\emptyset) = \mathbb{H}$ while $\mathsf{receive}(\emptyset) = \mathbb{L}$.

5.1 From secure typing to flow security

We are now going to show, in securely typed processes, each of their flows is such that the receiving level of its sources is lower than, or equal to, the tampering level of its target. To formalise this idea, we need some care for linear output. Consider a process $P \stackrel{\text{def}}{=} x(y_1y_2).\overline{x'}(y_1'y_2')([y_1' \to y_1] \mid [y_2' \to y_2])$ which is typed as:

$$\vdash P \triangleright x:(\overline{\star_{\mathbb{L}}}\,\overline{\star_{\mathbb{H}}})^{\downarrow_\mathrm{L}},\ x':(\star_{\mathbb{L}}\,\star_{\mathbb{H}})^{\uparrow_\mathrm{L}}$$

Intuitively, this process is secure, and is indeed securely typable, since there are two separate flows, one from c'_1 (hence y'_1) to c_1 (hence y_1) and another from c'_2 (hence y'_2) to c_2 (hence y_2): thus the high level of information only flows down to the high level action, not to the low level one. However, we can extract the following flow from this process:

$$\vdash P \blacktriangleright x : (\overline{\star_{\mathbb{L}}}\, \overline{\star_{\mathbb{H}}})^{\downarrow_{\mathrm{L}}} \rightsquigarrow x' : (\star_{\mathbb{L}}\, \star_{\mathbb{H}})^{\uparrow_{\mathrm{L}}} \tag{22}$$

which is intuitively correct, since to give information at both y'_1 and y'_2 via x', the process does need information at both y_1 and y_2 via x, so it cannot forget any of the types using $*$. But the receiving level of the source (x) is $\mathbb{H}$, while the tampering level of the target (x') is $\mathbb{L}$, saying the process is insecure, which is incorrect. This is because we cannot distinguish two flows once we collect them into a single type.

A solution to this problem is to extract flows individually.

$$\vdash P \blacktriangleright x : (\overline{\star_{\mathbb{L}}}\, *)^{\downarrow_{\mathrm{L}}} \rightsquigarrow x' : (\star_{\mathbb{L}}\, *)^{\uparrow_{\mathrm{L}}} \quad \text{and} \quad \vdash P \blacktriangleright x : (*\, \overline{\star_{\mathbb{H}}})^{\downarrow_{\mathrm{L}}} \rightsquigarrow x' : (*\, \star_{\mathbb{H}})^{\uparrow_{\mathrm{L}}},$$

which decomposes (22) into two flows. Note these two flows together capture the whole of the flow in (22), so there is no loss of information. In each of the decomposed flows, the receiving level is equal to the tampering level, hence we conclude this process is secure. We call such decomposed flows *basic*.

Definition 5.3 (basic) *τ of mode $\mathbf{!}$ or $\uparrow$ is* basic *if either:*

i. $\tau = (\vec{\rho}\rho'^{\uparrow})^{!}$ *or* $\tau = (\vec{*}\rho'\vec{*})^{\uparrow_{\mathrm{L}}}$, *where ρ' is basic in each case.*
ii. $\tau = ()^{\uparrow_{\mathrm{L}}}$ *or* $\tau = (\vec{*})_{s}^{\uparrow_{\mathrm{A}}}$

A flow $\Gamma \rightsquigarrow x : \tau$ is basic *if τ is basic and Γ is minimum among the sources of $x : \tau$ (cf. Proposition 4.9 (2)).*

For each basic type with non-trivial information content, its tampering level and the receiving level of its dual coincide. First we delineate those types which only bear trivial information.

Definition 5.4 (cancellable) The set of *cancellable types* (of mode $\mathbf{!}/\uparrow$) are generated from the following induction.

i.	τ	$=$	$*$		
ii.	τ	$=$	$(\rho_1 \mathinner{..} \rho_n)^{\uparrow_{\mathrm{L}}}$	$(n \geqslant 0)$	(each ρ_i is cancellable)
iii.	τ	$=$	$(\vec{\rho}\rho'^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$		(ρ' is cancellable)

We say τ is *co-cancellable* if $\overline{\tau}$ is cancellable. We also say Γ is co-cancellable if $\Gamma(x)$ is co-cancellable for all $x \in \mathtt{dom}(\Gamma)$.

Cancellable types represent a sequence of linear interactions which neither emit nor receive non-trivial information so that they are negligible from the viewpoint of information flow.

Lemma 5.5 *The set of non-cancellable basic types are generated from the following induction:*

i. $\tau = (\vec{*})^{\uparrow_{\mathrm{A}}}_{s}$.
ii. $\tau = (\vec{*}\,\rho\,\vec{*})^{\uparrow_{\mathrm{L}}}$ $(n \geqslant 1)$ *such that* ρ *is generated from (iii) below.*
iii. $\tau = (\vec{\rho}\rho')^{!}$ *such that* ρ' *is generated from (i) or (ii).*

Proof
Immediate by construction. □

Proposition 5.6 1. *If* $\tau > \tau'$ *then* $\mathsf{tamp}(\tau) \sqsubseteq \mathsf{tamp}(\tau')$.
2. *If* τ *is positive and not cancellable, then* $\mathsf{tamp}(\tau) \sqsubseteq \mathsf{receive}(\overline{\tau})$.
3. *If* τ *is basic and not cancellable, then* $\mathsf{receive}(\overline{\tau}) = \mathsf{tamp}(\tau)$.

Proof
(1) is straightforward by $\mathsf{tamp}(\tau^{!}) \sqsubseteq \mathsf{tamp}(*^{!}) = \mathbb{H}$ and $\mathsf{tamp}(\tau^{?}) = \mathsf{tamp}(*^{!}) = \mathbb{H}$. (2) is proved by induction on the types. We infer:

$$\mathsf{tamp}((\vec{\tau})^{\uparrow_{\mathrm{A}}}_{s}) = s = \mathsf{receive}((\vec{\overline{\tau}})^{\downarrow_{\mathrm{A}}}_{s})$$
$$\mathsf{tamp}((\vec{\tau})^{\downarrow_{\mathrm{L}}}) = \sqcap_i\{\mathsf{tamp}(\tau_i)\} \sqsubseteq \sqcup_i\{\mathsf{receive}(\overline{\tau_i})\} = \mathsf{receive}((\vec{\overline{\tau}})^{\downarrow_{\mathrm{L}}})$$
$$\mathsf{tamp}((\vec{\sigma}\rho)^{!}) = \mathsf{tamp}(\rho) \sqsubseteq \mathsf{receive}(\overline{\rho}) = \mathsf{receive}((\overline{\vec{\sigma}}\overline{\rho})^{?})$$

In the last two lines, we use induction hypothesis (note for a cancellable type, say $()^{\uparrow_{\mathrm{L}}}$, the inequation does not hold). For (3), we use the rule induction on the generation of basic, non-cancellable types in Lemma 5.5. The proof is as the same as (2). □

We use the following lemma for cancellable types. For the lemma, we use the following condition.

Definition 5.7 A cancellable type τ is is *well-formed* if, in Definition 5.4 (iii), we strengthen the condition:

$$\text{iii.} \quad \tau \quad = \quad (\vec{\rho}\rho'^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}} \qquad (\rho' \text{ and each } \overline{\rho_i} \text{ cancellable.})$$

Dually a co-cancellable type τ is *well-formed* if $\overline{\tau}$ is a well-formed cancellable type.

Below we simultaneously show, assuming well-formedness for co-cancellable types: (1) derivable cancellable types are always well-formed; and (2) a flow towards a cancellable target only uses co-cancellable sources.

Lemma 5.8 *Assume we restrict derivable flows so that each co-cancellable type in the source is well-formed.*

1. *If* $\vdash P^{A} \blacktriangleright \vec{y}:\vec{\rho} \rightsquigarrow x:\tau$ *such that* τ *is cancellable, then each* τ *is well-formed.*
2. *If* $\vdash P^{A} \blacktriangleright \vec{y}:\vec{\rho} \rightsquigarrow x:\tau$ *such that* τ *is cancellable, then each* ρ_i *is co-cancellable.*

Proof
We simultaneously prove (1) and (2) by rule induction. (1) is related only to $(\mathrm{In}^{!_{\mathrm{L}}})$ and $(\mathrm{Out}^{?_{\mathrm{L}}})$, respectively.

(Zero) Irrelevant.
(Par) Immediate from the definition of $\odot$.

(Res) Immediate.
(Weak) Immediate since $*^?$ is co-cancellable.
(Union, Subset, Empty) Immediate.
($\mathsf{In}^{\downarrow_\mathrm{L}}$) Because if σ is cancellable, by induction, $\vec{\rho}$ are co-cancellable, hence $(\vec{\rho})^{\downarrow_\mathrm{L}}$ is co-cancellable.
($\mathsf{Out}^{\uparrow_\mathrm{L}}$) Because if $\vec{\rho}$ is cancellable, by induction, each Γ_i is co-cancellable, including the case it is empty, hence $\vee_i \Gamma_i$ is co-cancellable (note if $\tau_{1,2}$ are co-cancellable their superposition by $\vee$ is also co-cancellable).
($\mathsf{In}^{\downarrow_\mathrm{A}}$) Irrelevant because σ cannot be cancellable by the linear/affine typing (since an affine input cannot prefix over a linear output by the typing rule).
($\mathsf{Out}^{\uparrow_\mathrm{A}}$) Irrelevant since $(\vec{\tau})_s^{\uparrow_\mathrm{A}}$ is not cancellable.
($\mathsf{In}^{!_\mathrm{L}}$) Because if $(\vec{\tau}\sigma)^{!_\mathrm{L}}$ is cancellable so is σ hence by induction Γ and each τ_i is co-cancellable. Hence Γ is co-cancellable and $(\vec{\tau}\sigma)^{!_\mathrm{L}}$ is well-formed.
($\mathsf{In}^{!_\mathrm{A}}$) Irrelevant since $(\vec{\tau}\,\sigma)^p$ is not cancellable.
($\mathsf{Out}^{?_\mathrm{L}}$) Suppose $(\vec{\tau}\rho)^{?_\mathrm{L}}$ is well-formed. Then τ_i is cancellable. Hence by the inductive hypothesis, Γ_i is co-cancellable. If σ is cancellable, then Γ, ρ are co-cancellable by induction. We note that $(\vec{\tau}\rho)^{?_\mathrm{L}}$ is also co-cancellable since ρ is co-cancellable. Hence $\vee_i \Gamma_i \vee \Gamma \cdot x : (\vec{\tau}\rho)^p$ is co-cancellable.
($\mathsf{Out}^{?_\mathrm{A}}$) Irrelevant as ($\mathrm{In}^{\downarrow_\mathrm{A}}$). □

We now define when a flow is safe.

Definition 5.9 (flow security) We say a basic flow $\Gamma \rightsquigarrow x : \tau$ is *secure* if $\mathsf{receive}(\Gamma) \sqsubseteq \mathsf{tamp}(\tau)$. A typed process P^A is *flow-secure* if whenever $\vdash P^A \blacktriangleright \Gamma \rightsquigarrow x : \tau$ such that $\Gamma \rightsquigarrow x : \tau$ is basic, it is secure.

Proposition 5.10 (secure typing implies flow-security) *If* $\vdash_{\mathsf{sec}} P \triangleright A$ *then* P^A *is flow-secure.*

Proof
Since secure typing is in direct correspondence with the underlying linear/affine typing, and because (Union) and (Subset) do not affect constituting flows, we can argue by rule induction on the flow inference rules. Below symbols for processes, types etc. refer to those occurring in each inference rule in Figure 1.

(Zero) Vacuous.
(Par) Assume we compose $\vdash P \blacktriangleright u : \delta \rightsquigarrow x : \rho$ and $\vdash Q \blacktriangleright x : \overline{\rho'} \rightsquigarrow v : \tau$ such that $\rho \succ \rho'$ to obtain a flow $u : \delta \rightsquigarrow v : \tau$ (the general case is proved precisely by the same reasoning). If ρ is cancellable, δ is co-cancellable by Lemma 5.8, that is $\mathsf{receive}(\delta) = \mathbb{L}$, hence done. If not, by induction hypothesis and by Proposition 5.6, we have $\mathsf{receive}(\delta) \sqsubseteq \mathsf{tamp}(\rho) \sqsubseteq \mathsf{tamp}(\rho') \sqsubseteq \mathsf{receive}(\overline{\rho'}) \sqsubseteq \mathsf{tamp}(\tau)$, as required.
(Res,Weak) Because $\mathsf{receive}(\Gamma) = \mathsf{receive}(\Gamma \cdot x : *)$.
(Sub,Union,Empty) Immediate.
($\mathsf{In}^{\downarrow_\mathrm{L}}$) Because $\mathsf{receive}(\Gamma \cdot x : (\vec{\rho})^{\downarrow_\mathrm{L}}) = \mathsf{receive}(\Gamma \cdot \vec{y} : \vec{\rho})$.
($\mathsf{Out}^{\uparrow_\mathrm{L}}$) Since we only infer a basic flow, and because the case when the type of x is $()^{\downarrow_\mathrm{L}}$ is vacuous, we can safely assume $\rho_i = *$ except for a single ρ_j for which we have $\vdash P \blacktriangleright \Gamma_j \rightsquigarrow y_j : \rho_j$. On this flow we use induction hypothesis to obtain $\mathsf{receive}(\Gamma) \sqsubseteq \mathsf{tamp}(\rho_j) = \mathsf{tamp}((\vec{*}\,\rho_j\,\vec{*})^{\uparrow_\mathrm{L}})$, hence done.

($\mathsf{In}^{\downarrow_{\mathrm{A}}}$) In the inference rule, consider the typing of P is $A, \vec{y} : \vec{\tau}$ while that of $x(\vec{y}).P$ is $A, x : (\vec{\tau})_s^{\downarrow_{\mathrm{A}}}$. By the side condition in the secrecy typing, we have $\mathsf{tamp}((\vec{\tau})_s^{\downarrow_{\mathrm{A}}}) = s \sqsubseteq \mathsf{tamp}(A)$. Since we know $A(w) = \mathsf{tamp}(\sigma)$ when σ is immediate tampering, we conclude $\mathsf{tamp}((\vec{\tau})_s^{\downarrow_{\mathrm{A}}}) \sqsubseteq \mathsf{tamp}(\sigma)$, as desired.

($\mathsf{Out}^{\uparrow_{\mathrm{A}}}$) Since $(\vec{\rho})_s^{\uparrow_{\mathrm{A}}}$ is basic, $\rho_i = *$. Hence Γ_i is co-cancellable, and $\mathsf{receive}(\Gamma_i) = \mathbb{L} \sqsubseteq s = \mathsf{tamp}((\vec{\rho})_s^{\uparrow_{\mathrm{A}}})$, as required.

($\mathsf{In}^{!}$) Because $\mathsf{receive}(\Gamma) \sqsubseteq \mathsf{receive}(\Gamma \cdot \vec{y} : \vec{\rho}) \sqsubseteq \mathsf{tamp}(\sigma) = \mathsf{tamp}((\vec{\rho}\,\sigma)^{!})$.

($\mathsf{Out}^{?}$) Because $\mathsf{receive}(\Gamma \cdot x : (\vec{\rho}\,\rho)^{?}) = \mathsf{receive}(z : \rho \cdot \Gamma) \sqsubseteq \mathsf{tamp}(\sigma)$. $\square$

Note that composition in (Par) has two cases, depending on whether the intermediate node is cancellable. When the cancellable types are involved, we use Lemma 5.8, since if not false dependency could arise (this potential false dependency due to composition at cancellable types becomes particularly significant when we introduce branching, as detailed in section 6.2). Observe also that the reasoning for the case ($\mathrm{Out}^{\uparrow_{\mathrm{L}}}$) goes through because the flow security is only about basic flows.

5.2 *From flow security to noninterference*

The next step is to show that the syntactic notion of flow security as defined in Definition 5.9 entails its semantic counterpart, the noninterference property, stipulated in Theorem 5.1. Along the way we show that the collection of flows of a process given by the flow inference system at least includes those given by a semantic means. The semantic notion of flows we shall use is defined as follows.

Definition 5.11 (semantic flow) Let $\vdash P \triangleright A$ where (1) $A(x_i) = \tau_i$ is negative for each $x_i \in \{\vec{x}\} \neq \emptyset$, (2) $A(y) = \tau$ is positive and (3) $A(u_j) = \rho_j$ is negative with $\vec{u} = \mathsf{fn}(A) \backslash \vec{x}y$. Then we say P^A *has a semantic flow from* $\vec{x}$ *to* y, written $\models P^A \blacktriangleright \vec{x} \rightsquigarrow y$, iff the following two conditions hold.

1. There exist $\vdash R_i \triangleright \vec{x} : \vec{\overline{\tau}}$ $(i = 1, 2)$, $\vdash S \triangleright y : \overline{\tau}, w : ()_s^{\uparrow_{\mathrm{A}}}$ and $\vdash Q \triangleright \vec{u} : \vec{\overline{\rho}}$ such that:
$$Q|R_1|P|S \Downarrow_w \quad \text{and} \quad Q|R_2|P|S \not\Downarrow_w.$$
2. For each $\vdash R \triangleright \vec{x} : \vec{\overline{\tau}}$ and $\vdash S \triangleright y : \overline{\tau}, w : ()_s^{\uparrow_{\mathrm{A}}}$, we have, for arbitrary $\vdash Q_{1,2} \triangleright \vec{u} : \vec{\overline{\rho}}$:
$$Q_1|R|P|S \Downarrow_w \quad \Leftrightarrow \quad Q_2|R|P|S \Downarrow_w.$$

We also write $\models P^A \blacktriangleright \emptyset \rightsquigarrow y$ (with $A(y)$ positive) iff $\models P^A \blacktriangleright \vec{x} \rightsquigarrow y$ never holds for any $\{\vec{x}\} \neq \emptyset$.

Intuitively, $\models P^A \blacktriangleright \vec{x} \rightsquigarrow y$ indicates there is a non-trivial flow (transmission of behaviour) to y from some of $x_i \in \{\vec{x}\}$, and from no other negative names. Note that the notion is defined behaviourally: in fact a semantic flow is invariant under the maximum contextual congruence $\cong$ given in section 3.2.

The following lemma for cancellable types is a key to the noninterference result.

Lemma 5.12 *Let* $\vdash P^{A, w:()_s^{\uparrow_{\mathrm{A}}}} \blacktriangleright \Gamma \rightsquigarrow w : ()_s^{\uparrow_{\mathrm{A}}}$ *such that types in* A *are negative and* Γ *is co-cancellable. Then* $\models P \blacktriangleright \emptyset \rightsquigarrow w$*, i.e.* $P|R_1 \Downarrow_w$ *iff* $P|R_2 \Downarrow_w$ *for any* $\vdash R_{1,2} \triangleright \overline{A}$.

Proof
See Appendix E. □

Intuitively, the lemma says that having the source of a flow which solely consists of co-cancellable types is semantically the same thing as having the empty source. We illustrate the key operational ideas underlying this lemma. By analysing the inference rules which may induce co-cancellable types, we can observe those actions associated with co-cancellable types are of quite specific nature – those which only consist of unary linear input/output and their replicated variants. More concretely, interactions between a process P and its environment interfaced at co-cancellable channels satisfy the following property:

if P has a co-cancellable interface Γ to w, there always exists a finite chain of linear transitions with $R_{1,2}$ without being interfered by nonterminating actions.

Here by linear transitions we mean those which involve only linear replicated actions and unary linear actions. Note that this is intuitively obvious since the derivability of co-cancellable types at the source of a flow means that, by the definition of cancellable types, only replicated actions and unary linear actions are exchanged between the process and the environment starting from the typed channel. In fact, the sequences of linear transitions thus derived have a regular form called *linear call sequence* (l.c.s), which is introduced and studied in Yoshida (2002) and Yoshida *et al.* (2002). We illustrate the notion of l.c.s. by a concrete example. Take:

$$P \equiv \overline{a}(x)x.\overline{b}(y)y.\overline{w} \mid [b \to a] \mid (\boldsymbol{\nu}\, v)(\Omega_v \mid \overline{v}\langle e\rangle)$$

If we assume a target is co-cancellable, we cannot assign a and b to $(()^{\uparrow_{\mathrm{A}}})^{?_{\mathrm{L}}}$ or $(()^{\uparrow_{\mathrm{A}}})^{?_{\mathrm{A}}}$. Hence a flow to w is:

$$\vdash P \blacktriangleright a : (()^{\downarrow_{\mathrm{L}}})^{?_{\mathrm{L}}} \cdot b : (()^{\downarrow_{\mathrm{L}}})^{?_{\mathrm{L}}} \rightsquigarrow w : ()^{\uparrow_{\mathrm{A}}}_{s}$$

Now the l.c.s. reaching w is:

$$\overline{a}(x)^{(()^{\downarrow_{\mathrm{L}}})^{?_{\mathrm{L}}}} \cdot x^{()^{\downarrow_{\mathrm{L}}}} \cdot (\overline{b}(y)^{(()^{\downarrow_{\mathrm{L}}})^{?_{\mathrm{L}}}}, b(y)^{(()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}}) \cdot (\overline{y}^{()^{\uparrow_{\mathrm{L}}}}, y^{()^{\downarrow_{\mathrm{L}}}})$$

where we explicitly annotate each label by its type. Note that the typing $(()^{\downarrow_{\mathrm{L}}})^{?_{\mathrm{L}}}$ of a and b determines the typing of bound names x and y. We note that finiteness of linear call sequences does *not* imply convergence of the whole term (actually we can observe P above has a diverging reduction path even though $P \Downarrow_w$). But we *can* guarantee that the interaction with R_i has the form (except linear call-backs from the environment, which never affects the subsequent behaviour of P):

$$(\overline{a}(x), a(x)) \cdot (x, \overline{x}) \cdot (\overline{b}(y), b(y)) \cdot (\overline{y}, y)$$

without being interfered by affine transitions at intermediate steps. By this and by its shape, a linear call sequence never affects the final convergence action (the output at w in the example above). From this we conclude, for appropriately typed $R_{1,2}$, that we always have $P|R_1 \Downarrow_w$ iff $P|R_2 \Downarrow_w$. For detail, see Appendix E.

We next observe a basic consequence of Lemma 5.12, which elucidates the central status of Lemma 5.12 in the technical machinery of the present flow analysis. Below we say a flow $\Gamma \rightsquigarrow x : \tau$ is *non-trivial* if τ is not cancellable and, for some y, $\Gamma(y) = \rho$ is not co-cancellable.

Corollary 5.13 (completeness of flow inference) *If* $\models P^A \blacktriangleright \vec{x} \rightsquigarrow y$ *with* $\{\vec{x}\} \neq \emptyset$, *then* $\vdash P^A \blacktriangleright \Gamma \rightsquigarrow y : \tau$ *for a non-trivial* $\Gamma \rightsquigarrow y : \tau$ *such that* $\{\vec{x}\} \subset \mathsf{fn}(\Gamma)$.

Proof
If there is a semantic flow from $\vec{x}$ to y, then, taking $P' \stackrel{\text{def}}{=} P|S$ for S as given in Definition 5.11, there is a non-trivial semantic flow from $\vec{x}$ to w in P' by the definition of semantic flow. By Lemma 5.12, P' should have a non-trivial source, say Γ. But in the derivation for this flow inference for P', a derivation for P is included (because the inference rules are compositional), all of whose interface points with S at x cannot be cancellable by Lemma 5.8. $\square$

Corollary 5.13 says that whenever there is a non-trivial semantic flow in a typed process, it also has a non-trivial syntactic flow inferable by the flow inference system. It is notable that we can also derive Lemma 5.12 from Corollary 5.13. In fact, assume Corollary 5.13 holds and let $\vdash P^{A, w:()_s^{\uparrow \mathrm{A}}} \blacktriangleright \Gamma \rightsquigarrow w : ()_s^{\uparrow \mathrm{A}}$ such that types in Γ are co-cancellable. If $\models P \blacktriangleright \vec{y} \rightsquigarrow w$ for non-empty $\vec{y}$, Corollary 5.13 says that $\vdash P \blacktriangleright \Delta \rightsquigarrow w : ()_s^{\uparrow \mathrm{A}}$ where Δ is not co-cancellable. By Proposition 4.9 (2) we have Γ is also not co-cancellable, a contradiction. Hence $\models P \blacktriangleright \emptyset \rightsquigarrow w$, as required.

Another observation is that Corollary 5.13 suggests a basic method for debugging flow inference rules, where each rule should obey the satisfaction of the stated completeness property inductively.

We also use the following syntactic property of co-cancellable types for our noninterference result.

Lemma 5.14 *If* $\vdash P^{A, x:\rho} \blacktriangleright \Gamma \rightsquigarrow x : \tau$ *with* $y \in \mathtt{dom}(\Gamma)$ *where* $\Gamma(y)$ *is not co-cancellable, then* $\mathsf{tamp}(\overline{A}(y)) \sqsubseteq \mathsf{receive}(\Gamma(y))$.

Proof
By $\Gamma(y) \prec A(y)$ and Proposition 5.6, we have: $\mathsf{tamp}(\overline{A(y)}) \sqsubseteq \mathsf{receive}(A(y)) \sqsubseteq \mathsf{receive}(\Gamma(y))$, as required. $\square$

We can now prove noninterference.

Proposition 5.15 (flow security implies noninterference) *If* $\vdash P \triangleright \overline{A}, w : ()_s^{\uparrow \mathrm{A}}$ *is flow-secure,* $\mathsf{tamp}(A) \not\sqsubseteq s$ *and* $\vdash R_{1,2} \triangleright A$, *then* $P|R_1 \Downarrow_w$ *iff* $P|R_2 \Downarrow_w$.

Proof
Let $\vdash P \triangleright \overline{A}, w : ()_s^{\uparrow \mathrm{A}}$ be flow-secure with $\mathsf{tamp}(A) \not\sqsubseteq s$ (note this implies $\mathsf{fn}(A) \neq \emptyset$). We show $\models P \blacktriangleright \emptyset \rightsquigarrow w$. Below let $\{y\} = \mathsf{fn}(A)$. First we show Γ is (i.e. all its types are) co-cancellable. We reason towards a contradiction.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\Gamma(y_i) \text{ not co-cancellable } \wedge \ \mathsf{tamp}(A) \not\sqsubseteq s} \\
\Rightarrow & \mathsf{tamp}(A(y_i)) \sqsubseteq \mathsf{receive}(\Gamma(y_i)) & \text{(Lemma 5.14)} \\
\Rightarrow & \mathsf{receive}(\Gamma(y_i)) \not\sqsubseteq s & (\mathsf{tamp}(A(y_i)) \not\sqsubseteq s)
\end{array}
$$

But this contradicts P^A being flow-secure, hence Γ is co-cancellable. We can now reason:

$$\begin{array}{llll} \vdash P \blacktriangleright \vec{y} : \vec{\rho} \rightsquigarrow w & \Rightarrow & \vec{\rho} \ \ \text{co-cancellable} & \text{(above)} \\ & \Rightarrow & \models P \blacktriangleright \emptyset \rightsquigarrow w & \text{(Lemma 5.12),} \end{array}$$

hence as required. □

By combining Propositions 5.10 and 5.15, we have now reached the non-interference property, establishing Theorem 5.1.

Remark 5.16 Lemma 5.12, as well as the technical development to follow, shows that co-cancellable types in the source are operationally negligible. In particular, if we only consider the cases when all types are affine, we do not have co-cancellable types, hence we do not need Lemma 5.12 (even though we need a weaker version for a flow which has the empty source). One may thus wonder whether there is any significance to have (unary) linear actions in processes. Their incorporation is however useful in many settings, for example when we encode sequential composition of commands, when we need indirection in the encoding of polymorphic programs, when representing the unit type. Thus having co-cancellable types is useful, in spite of its needing an extra step in reasoning. Note however an extracted flow may as well neglect co-cancellable sources.

Remark 5.17 Definition 5.15 does *not* exclude a flow which uses an "unsafe" type, such as $\emptyset \rightsquigarrow x : (\overline{\star_{\mathbb{H}}}()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}})^{!_{\mathrm{A}}}$ (which contains an unsafe flow from $\overline{\star_{\mathbb{H}}}$ to $()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}}$). This does not cause a problem in the present technical development because of the simple shape of the target type in the non-interference result in Proposition 5.15 (in other words, including or excluding these unsafe types does not change the technical development). The flow inference we shall present in section 7.4 later naturally incorporates this consideration.

6 Extensions (1): Inflation and branching

This section and the next extend the flow analysis to broader classes of typed process behaviours. This section treats two extensions which still stay within pure functional behaviour, *inflation* and *branching*. In the next section we treat the extensions which incorporate stateful behaviour.

6.1 Inflation

The inflation operation, which is suggested by Dependency Core Calculus (Abadi *et al.*, 1999), allows relaxing of the secrecy typing by changing secrecy levels as far as global flows remain safe. Write $\tau \sqcup s$ for the result of raising each secrecy level occurring in τ by taking its join with s. For example, $\star_{\mathbb{L}} \sqcup \mathbb{H} = \star_{\mathbb{H}}$ with $\star_s = (()^{\uparrow_{\mathrm{A}}}_s)^{!_{\mathrm{A}}}$. We extend this operation pointwise to A, writing $A \sqcup s$. Using this operation, the typing rule for inflation is given by:

$$(\mathsf{Inf})\,\frac{\vdash P \triangleright \mathsf{inf}(A)}{\vdash P \triangleright A}$$

where we set $\mathsf{inf}(A) = A \sqcup \mathsf{tamp}(A)$.

To show how this extends the secrecy typing without endangering secure information flow, we consider the following process.

$$Q \stackrel{\text{def}}{=} \overline{y}(ab)(!a(c).\overline{z}(c')c'^{\mathbb{H}}.\overline{c}^{\mathbb{L}} \mid b^{\mathbb{H}}.\overline{e}^{\mathbb{H}})$$

where we attach the security levels on channels for readability. Let $B \stackrel{\text{def}}{=} y : (\star_{\mathbb{L}}\, ()^{\downarrow_{\mathrm{A}}}_{\mathbb{H}})^{?_{\mathrm{L}}}, z : \overline{\star_{\mathbb{H}}}, e : ()^{\uparrow_{\mathrm{A}}}_{\mathbb{H}}$, under which Q is untypable in the secrecy typing without inflation because a high-level input c' prefixes a low-level output c. Yet we can argue Q is in fact secure under B, since this violation is not observable to the environment which receives information only at the high-level. More precisely, this process only affects the environment at e, and never at y and z. For example, we may compose Q with:

$$R \stackrel{\text{def}}{=} !y(ab).\overline{a}(c)c^{\mathbb{L}}.\overline{b}^{\mathbb{H}} \mid !z(c').\overline{c}'^{\mathbb{H}}$$

Note that the insecure action at $c^{\mathbb{L}}$ is "nullified" by the high-level action at $b^{\mathbb{H}}$ by R. In other word, the local violation in Q is ineffective as a whole. In fact, as we shall see later, Q is flow-secure, so, in the light of Proposition 5.10, Q is secure. With (Inf), however, we can check Q becomes typable under B, as follows.

$$(\mathsf{Inf})\,\frac{\vdash_{\mathsf{sec}} Q \triangleright y : (\star_{\mathbb{H}}\, ()^{\downarrow_{\mathrm{A}}}_{\mathbb{H}})^{?_{\mathrm{L}}},\ z : \overline{\star_{\mathbb{H}}},\ e : ()^{\uparrow_{\mathrm{A}}}_{\mathbb{H}}}{\vdash_{\mathsf{sec}} Q \triangleright y : (\star_{\mathbb{L}}\, ()^{\downarrow_{\mathrm{A}}}_{\mathbb{H}})^{?_{\mathrm{L}}},\ z : \overline{\star_{\mathbb{H}}},\ e : ()^{\uparrow_{\mathrm{A}}}_{\mathbb{H}}}$$

The flow analysis associated with this operation is simple, adding the following sequent ($\mathsf{F} \sqcup s$ is given as $A \sqcup s$)

$$(\mathsf{Inf})\,\frac{\vdash P^{\mathsf{inf}(A)} \blacktriangleright \mathsf{F} \sqcup \mathsf{tamp}(A)}{\vdash P^{A} \blacktriangleright \mathsf{F}}$$

Intuitively, the addition of this rule (which in fact does not change derivability of flows except inflated secrecy levels in flows) still maintains the flow security, since $\mathsf{tamp}(A)$ is always lower than, or equal to, the tamper level of τ in a basic flow $\Gamma \rightsquigarrow x : \tau$, hence its effect on Γ and τ is negligible from the viewpoint of flow-security. Indeed:

Proposition 6.1 *If $\vdash_{\mathsf{sec}} P \triangleright A$ with inflation then P^A is flow-secure.*

Proof

Assume we derive $\vdash P \blacktriangleright \Gamma' \rightsquigarrow x : \tau'$ from $\vdash P \blacktriangleright \Gamma \rightsquigarrow x : \tau$ by (Inf). By definition, $\mathsf{tamp}(\tau') = \mathsf{tamp}(\tau)$ and, for each y_i in $\mathsf{fn}(\Gamma)$, $\mathsf{receive}(\Gamma'(y_i)) \sqsubseteq \mathsf{receive}(\Gamma(y_i))$. Finally all and only cancellable types in Γ are those in Γ'. $\square$

Note that the same behavioural secrecy (Lemma 5.12) immediately holds in this extension (since the set of untyped processes derived with co-cancellable sources remain identical). Hence by combining Proposition 5.15 and Proposition 6.1 above, we obtain the noninterference theorem (Theorem 5.1) for this system.

6.2 Branching and selection

Branching is used for representing base values as well as conditionals (Berger *et al.*, 2001; Yoshida *et al.*, 2001). We assume the index set I is either countable or finite. While this construct can be simply encoded into the calculus without it, the branching plays a basic role in the typed setting and is essential for the secrecy analysis of complex behaviour. In the system with branching and selection, processes are extended as follows.

$$P \ ::= \ \ldots \ \mid \ x[\&_{i\in I}(\vec{y}_i).P_i] \ \mid \ \overline{x}\mathtt{in}_i(\vec{y})P$$

We also often omit the indexing set I of $x[\&_{i\in I}(\vec{y}_i).P_i]$. Dynamics of branching involves selection of one branch, discarding remaining ones, as well as name passing.

$$x[\&_i(\vec{y}_i).P_i] \,|\, \overline{x}\mathtt{in}_i(\vec{y}_i)P \ \longrightarrow \ (\boldsymbol{\nu}\,\vec{y}_i)(P_i \,|\, P)$$

For types, we extend the grammar by:

$$\tau \ ::= \ \ldots \ \mid \ [\&_i\vec{\tau}_i^{\,?}]_s^{\downarrow_{\mathrm{L}}} \ \mid \ [\oplus_i\vec{\tau}_i^{\,!}]_s^{\uparrow_{\mathrm{L}}}$$

We only add linear branching/section for simplicity. In fact, from the viewpoint of expressiveness, this is enough since the affine branching can be represented by combination of a unary affine type with linear branching. Branching types ($\&$) are negative, while selection types ($\oplus$) are positive, and they are dual to each other. Note that unlike $(\vec{\tau})^{\uparrow_{\mathrm{L}}}$, a selection type $[\oplus_i\vec{\tau}_i]_s^{\uparrow_{\mathrm{L}}}$ is immediately tampering as it transmits information to the received channel. Hence we set:

$$\mathsf{tamp}([\oplus_i\vec{\tau}_i]_s^{\uparrow_{\mathrm{L}}}) = s \qquad \mathsf{receive}([\&_i\vec{\tau}_i]_s^{\downarrow_{\mathrm{L}}}) = s$$

Selections are *never* cancellable, dually for branching. The typing rules are defined in Appendix C. For illustration we give a couple of examples which use branching.

Example 6.2 (branching types)

1. A *natural number agent*, $[\![n]\!]_u \stackrel{\mathrm{def}}{=} !u(c).\overline{c}\mathtt{in}_n$, acts as a server which necessarily returns a fixed answer n. This agent is invoked by sending a channel c through its free channel u, to which an output should be sent. c plays the role of a continuation of interaction. Now let $\mathbb{N}_s^{\bullet} \stackrel{\mathrm{def}}{=} [\oplus_{i\in\mathbb{N}}]_s^{\uparrow_{\mathrm{L}}}$ and $\mathbb{N}_s^{\circ} \stackrel{\mathrm{def}}{=} (\mathbb{N}_s^{\bullet})^{!_{\mathrm{L}}}$ with $\mathbb{N}$ the set of natural numbers. Then $\vdash_{\mathsf{sec}} [\![n]\!]_x \triangleright x : \mathbb{N}_s^{\circ}$ is typable. Note $\mathsf{tamp}(\mathbb{N}_s^{\circ}) = s$ (note this agent does emit information when asked).
2. The process $\overline{u}(c)c[\&_{n\in\mathbb{N}}.\overline{e}\mathtt{in}_{n+1}]$ acts as the successor function for the natural number agent. This successor invokes the natural number with continuation c; if its i-th branch is selected via c, it emits the answer $i+1$ via e.

$$\begin{array}{rcl}
[\![2]\!]_u|\overline{u}(c)c[\&_{n\in\mathbb{N}}.\overline{e}\mathtt{in}_{n+1}] & \equiv & (\boldsymbol{\nu}\,c)([\![2]\!]_u|\overline{u}\langle c\rangle|c[\&_{n\in\mathbb{N}}.\overline{e}\mathtt{in}_{n+1}]) \\
& \longrightarrow & (\boldsymbol{\nu}\,c)([\![2]\!]_u|\overline{c}\mathtt{in}_2|c[\&_{n\in\mathbb{N}}.\overline{e}\mathtt{in}_{n+1}]) \\
& \longrightarrow & (\boldsymbol{\nu}\,c)([\![2]\!]_u|\overline{e}\mathtt{in}_3) \\
& \equiv & [\![2]\!]_u|\overline{e}\mathtt{in}_3
\end{array}$$

The essence of this encoding lies in the precise representation of functional behaviour as an interacting process. Then $\vdash_{\mathsf{sec}} \overline{u}(c)c[\&_{n\in\mathbb{N}}.\overline{e}\mathtt{in}_{n+1}] \triangleright u : \overline{\mathbb{N}^{\circ}_{s}}, e : \mathbb{N}^{\bullet}_{s'}$ is well-typed iff $s \sqsubseteq s'$. To check this, it suffices to know that $\vdash_{\mathsf{sec}} c[\&_{n\in\mathbb{N}}.\overline{e}\mathtt{in}_{n+1}] \triangleright c : \overline{\mathbb{N}^{\bullet}_{s}} \to e : \mathbb{N}^{\bullet}_{s'}$ is well-typed, the condition for which is nothing but $s \sqsubseteq s'$.

For the flow analysis, first we extend the syntax of channel types as in section 4.1. Then we add the four prefix rules for the added constructs.

$$(\mathsf{Bra}^{\downarrow\mathrm{L}})\ \frac{\vdash P_i \blacktriangleright \Gamma \cdot \vec{y}_i : \vec{\rho}_i \rightsquigarrow w : \sigma}{\vdash x[\&_i(\vec{y}_i).P_i] \blacktriangleright \Gamma \cdot x : [\&_i \vec{\rho}_i]^{\downarrow\mathrm{L}}_{s} \rightsquigarrow w : \sigma} \qquad (\mathsf{Sel}^{\uparrow\mathrm{L}})\ \frac{\forall i,j.\ \vdash P \blacktriangleright \Gamma_{ij} \rightsquigarrow y_{ij} : \rho_{ij}}{\vdash \overline{x}\mathtt{in}_i(\vec{y})P \blacktriangleright \ \vee_{ij}\Gamma_{ij} \rightsquigarrow x : [\oplus_i \vec{\rho}_i]^{\uparrow\mathrm{L}}_{s}}$$

Note these rules treat actions which immediately emit/receive information, so have the same shape as $(\mathrm{In}^{\downarrow\mathrm{A}})$ and $(\mathrm{Out}^{\uparrow\mathrm{A}})$ in Figure 1. In branching, there arises a subtle point in composition at (co-)cancellable types, which we illustrate using an example.

Example 6.3 (composition at cancellable types in branching) As we have seen from the previous section, representing (co-)cancellable types in flows are in effect unnecessary in the unary case. They become harmful in the presence of branching/selection. Take, for example, the following process:

$$\vdash f.\overline{w} \triangleright f : ()^{\downarrow\mathrm{L}}, w : ()^{\uparrow\mathrm{A}}_{\mathbb{L}}, \qquad \vdash x[.\overline{f}\&.\overline{f}] \triangleright x : [\&]^{\downarrow\mathrm{L}}_{\mathbb{H}}, f : ()^{\uparrow\mathrm{L}} \tag{23}$$

They respectively have the following flows:

$$\vdash f.\overline{w} \blacktriangleright f : ()^{\downarrow\mathrm{L}} \rightsquigarrow w : ()^{\uparrow\mathrm{A}}_{\mathbb{L}}, \qquad \vdash x[.\overline{f}\&.\overline{f}] \blacktriangleright x : [\&]^{\downarrow\mathrm{L}}_{\mathbb{H}} \rightsquigarrow f : ()^{\uparrow\mathrm{L}} \tag{24}$$

They are both secure. However, a naive composition of these flows leads to the following flow:

$$x[\&]^{\downarrow\mathrm{L}}_{\mathbb{H}} \rightsquigarrow w : ()^{\uparrow\mathrm{A}}_{\mathbb{L}} \tag{25}$$

which is clearly *not* secure.

A solution to the above problem is to prohibit such composition, and to allow the trimming of co-cancellable types from the source of a flow. When incorporating these two elements, the flow analysis induces:

$$\vdash f.\overline{w} \blacktriangleright \emptyset \rightsquigarrow w : ()^{\uparrow\mathrm{A}}_{\mathbb{L}}, \qquad \vdash x[.\overline{f}\&.\overline{f}] \blacktriangleright \emptyset \tag{26}$$

which is composed to give the safe flow $\emptyset \rightsquigarrow w : ()^{\uparrow\mathrm{A}}_{\mathbb{L}}$. Note that having the flow (26) instead of (24) makes sense since causality is in effect nonexistent at the composition.

For formally preventing composition at cancellable types, we first introduce the following structural rule.

$$(\mathsf{Cancel})\ \frac{\vdash P \blacktriangleright \Gamma \cdot y : \rho \rightsquigarrow x : \tau \quad \rho \text{ co-cancellable}}{\vdash P \blacktriangleright \Gamma \rightsquigarrow x : \tau}$$

Note this rule has no substantial effect in the unary setting since co-cancellable types never have any effect, both in inference (because co-cancellable types have the lowest receiving level) and in operation (because they do not transmit information).

By having (Cancel), we do not have to have channels typed with co-cancellable types when composing two flows. We further *require* that no such composition can occur.

Definition 6.4 In the flow analysis in this section and next, we add the condition: "*with τ not co-cancellable*" in Definition 4.3 (i). That is, we replace Definition 4.3 (i) by: *If $A_2(x) = \overline{A_1(x)}$ and $x : \tau$ occurs as a source in F_1 and τ is not co-cancellable, then $x : \delta$ occurs in F_2 as a target such that $\tau \prec \overline{\delta}$.*

Incorporating this refinement does not affect the technical development of the analysis for unary processes since composition at cancellable types *was* of no use in the unary case by Lemma 5.12. But they can prevent the dangerous composition as discussed in Example 6.3.

The extended system satisfies uniqueness of inferable flows (Proposition 4.9 (ii)), modulo the difference of channels with co-cancellable types in their sources. Basic types add linear selection types $[\oplus\vec{*}]_s^{\uparrow_{\mathrm{L}}}$. As before (Proposition 5.6), if τ is basic and is not cancellable, then $\mathsf{receive}(\overline{\tau})$ and $\mathsf{tamp}(\tau)$ coincide. The notion of *flow-security* (Definition 5.9) remains precisely the same.

Lemma 5.8 does not hold since a linear selection can be prefixed by a unary linear input. This does not matter since we do not allow composition of co-cancellable types, which is the only place where Lemma 5.8 is used.

Proposition 6.5 *If $\vdash_{\mathsf{sec}} P \triangleright A$ with branching and selection then P^A is flow-secure.*

Proof

$(\mathsf{Bra}^{\downarrow_{\mathrm{L}}}, \mathsf{Sel}^{\uparrow_{\mathrm{L}}})$ As $(\mathrm{In}^{\downarrow_{\mathrm{A}}})$ and $(\mathrm{Out}^{\uparrow_{\mathrm{A}}})$ in Proposition 5.10, respectively.

(Cancel) Immediate since $\mathsf{receive}(\rho) = \mathbb{L}$. $\square$

To deduce behavioural security from flow security, we need some preparation. Assume we have derived a flow from P in the system which use neither (Cancellable) nor Definition 6.4. Note, because of dangerous composition at co-cancellable types, the derived flow may not be secure. Yet we can always extract a process from P which does not use composition at co-cancellable channels as follows.

If the derivation of a flow does not involve composition at co-cancellable types, we can use P as it is. If it does, then we can find P' and S such that $P \equiv P'|S$ (here we are safely neglecting hiding of composed channels), so that the interface between P' and S is precisely the composition(s) at co-cancellable channel(s). For example, in the case of the processes in Example 6.3, we first have $P \stackrel{\text{def}}{=} x[.\overline{f}\&.\overline{f}]|f.\overline{w}$, which we cut off at the problematic composition at f, to obtain:

$$P' \stackrel{\text{def}}{=} f.\overline{w} \qquad S \stackrel{\text{def}}{=} x[.\overline{f}\&.\overline{f}].$$

If there are n such compositions in distinct branches of a flow, we have $P \equiv P'|S_1|..|S_n$ where each S_i has a distinct cancellable interface, so that we take $S \stackrel{\text{def}}{=} S_1|..|S_n$. If two or more such compositions are located in the same branch, we choose the one which occurs as near to w as possible. The (flow of the) derived P' does not use the composition at co-cancellable types but still retains w. Note that, for such P', Lemma 6.5 is valid due to the lack of problematic compositions.

We can now derive behavioural security from flow security precisely as we did in section 5.2. Take P and $R_{1,2}$ as in Lemma 5.12 and Proposition 5.15. By the decomposition of P into P' and S, we can regard $P|R_{1,2}$ as $P'|(R_{1,2}|S)$. If the flow of P' has co-cancellable types in its source, then by the same linear liveness property in the linear/affine π-calculus with branching (Yoshida, 2002), $R_{1,2}$ do not affect convergence of P, cf. Lemma 5.12. We can then show, via Lemma 5.14 in the present setting, that if $\vdash P' \triangleright \overline{A}, w:()^{\uparrow_{\mathsf{A}}}_{s}$ and $\mathsf{tamp}(A) \not\sqsubseteq s$, and if $\Gamma \rightsquigarrow w:()^{\uparrow_{\mathsf{A}}}_{s}$ is a flow from P', then Γ solely consists of co-cancellable types. Thus we obtain Proposition 5.15, reaching the noninterference, Theorem 5.1, for this extension.

We observe that integrating inflation into the flow analysis with branching/selection does not change any technical development, since the inflation rule is flow secure and because behavioural properties of flow security are independent from individual embeddings. The resulting flow analysis can easily justify the semantic soundness of Dependency Core Calculus (Abadi *et al.*, 1999) via the standard process encoding of functions into processes.

7 Extensions (2): State

7.1 Elementary state (1): Flow analysis

This section discusses flow analysis on stateful extensions of the linear/affine π-calculus (π^{LA}). The stateful extension, designated π^{LAR}, adds a *reference agent*, which embodies stateful computation, to π^{LA}. This addition means that the clear distinction between positive/negative types, which we had in π^{LA}, is lost: some typed channels can be used for both emitting and receiving information. However distinction between positivity and negativity is still essential for our flow analysis – information always flows from negative channels to positive channels. Thus the flow analysis continues to focus on polarities, or directions of information. The only change is that a single channel in a process can be used both positively and negatively, or as a source and as a target, even in a single flow.

To give a cleanly articulated presentation of the subtle points involved in the flow analysis with mixed polarities, we first treat a simpler form of stateful actions in section 7.1 and section 7.2, where we augment π^{LA} with a minimal notion of state; we then treat the flow analysis for the full π^{LAR}-calculus in section 7.3 and section 7.4, where a general notion of state is fully integrated into the original calculus.

A basic stateful process is an encoding of an imperative variable, which we call *reference*. We use the following extension of the grammar for processes, adding references and their duals.

$$P \ ::= \ \ldots \ \mid \ \mathsf{Ref}\langle xv\rangle \ \mid \ \overline{x}\,\mathtt{read}\langle c\rangle \ \mid \ \overline{x}\,\mathtt{write}\langle yc\rangle$$

$\mathsf{Ref}\langle xv\rangle$ is called *reference*, while $\overline{x}\,\mathtt{read}\langle c\rangle$ and $\overline{x}\,\mathtt{write}\langle yc\rangle$ are called *read* and *write* selections, respectively. In $\mathsf{Ref}\langle xv\rangle$, x is the principal channel and v is the stored value (which is also a channel name). This process waits for invocation at x, with one branch for reading and one branch for writing. The read branch receives a single name c as a continuation from the request, which is used to return its content v. In the write branch, it receives two names, v' and c, and uses v' as its new value (thus changing its state) and acknowledges the receipt via c. Dynamics of $\mathsf{Ref}\langle xv\rangle$ is formally defined as follows.[2]

$$\begin{array}{rcl} \mathsf{Ref}\langle xv\rangle \,|\, \overline{x}\,\mathtt{read}\langle c\rangle & \longrightarrow & \mathsf{Ref}\langle xv\rangle \,|\, \overline{c}\langle v\rangle \\ \mathsf{Ref}\langle xv\rangle \,|\, \overline{x}\,\mathtt{write}\langle v'c\rangle & \longrightarrow & \mathsf{Ref}\langle xv'\rangle \,|\, \overline{c} \end{array}$$

The reader can find more examples of the encodings with reference agents in Honda & Yoshida (2002). Introduction of references makes the calculus nondeterministic (as we shall see in later examples). Regarding types, we add reference types and mutable replicated affine types to the grammar so that the type structure simply and minimally increments the original linear/affine types.

$$\begin{array}{lcl} \tau_e & ::= & \text{from section 2} \\ \tau & ::= & \tau_e \mid \mathsf{ref}_s\langle \tau_e^{!}\rangle \mid \mathsf{rw}_s\langle \tau_e^{?}\rangle \mid (\vec{\tau}_e^{\,?}\tau_e^{\uparrow_{\mathrm{A}}})_s^{!_{\mathrm{A}}} \mid (\vec{\tau}_e^{\,!}\tau_e^{\downarrow_{\mathrm{A}}})_s^{?_{\mathrm{A}}} \mid (\vec{\tau}_e^{\,?_{\mathrm{L}}}\tau_e^{\uparrow_{\mathrm{L}}})_s^{?_{\mathrm{L}}} \mid (\vec{\tau}_e^{\,?}\tau_e^{\uparrow_{\mathrm{L}}})_s^{?_{\mathrm{L}}} \end{array}$$

The added types are called *elementary stateful types*, while the types from section 2, written τ_e in this subsection, are called *stateless types*. $\mathsf{ref}_s\langle\tau\rangle$ stands for $[(\tau)^{\uparrow_{\mathrm{L}}}\&\overline{\tau}()^{\uparrow_{\mathrm{L}}}]_s^{!_{\mathrm{R}}}$, while $\mathsf{rw}_s\langle\tau\rangle = \overline{\mathsf{ref}_s\langle\overline{\tau}\rangle}$. Accordingly these types have mode $!_{\mathrm{R}}$ and its dual $?_{\mathrm{R}}$, respectively. $!_{\mathrm{L}}$ and $?_{\mathrm{L}}$ now have their stateful counterpart, distinguished by added annotations of secrecy levels. Similarly for $!_{\mathrm{R}}$ and $?_{\mathrm{R}}$ (these modes are only used for reference types). They are given secrecy levels since they receive/emit information, as we shall discuss in the next subsection. Elementary stateful types are those which do not carry stateful types, even though the types themselves can be stateful. Restricting to elementary stateful types allows us to demonstrate the subtlety of information flow specific to stateful actions in a simplest possible setting with a minimal increment on π^{LA}. The resulting calculus is also expressive enough to encode many stateful behaviours, such as a first-order imperative programming language (the so-called While language (Winskel, 1993)). The tampering levels of the elementary stateful types (for added ones) are given as follows:

$$\begin{array}{rclcrcl} \mathsf{tamp}(\mathsf{ref}_s\langle\tau\rangle) & = & \mathsf{tamp}(\tau) & \qquad & \mathsf{tamp}((\vec{\tau}\tau)_s^{!}) & = & \mathsf{tamp}(\tau) \\ \mathsf{tamp}(\mathsf{rw}_s\langle\tau\rangle) & = & s & & \mathsf{tamp}((\vec{\tau}\tau)_s^{?}) & = & s \end{array}$$

The typing rules for additional constructs are given in Appendix D.[3] We also add the following condition from Honda & Yoshida (2002) for well-formed types.

Definition 7.1 (structural security for elementary stateful types) τ is *structurally secure* if (1) for each occurrence of $\mathsf{ref}_s\langle\tau'\rangle$ in τ, we have $s \sqsubseteq \mathsf{tamp}(\mathsf{ref}_s\langle\tau'\rangle)$ and (2) for each occurrence of $\mathsf{rw}_s\langle\tau'\rangle$ in τ, we have $s \sqsubseteq \mathsf{tamp}(\mathsf{ref}_s\langle\overline{\tau'}\rangle)$.

[2] We can represent the reference by bound name passing by translating $\overline{c}\langle v\rangle$ into the copycat, cf. Example 2.1 (2). We use free name passing for reference agents and their duals, which results in an arguably simpler presentation.

[3] In (Honda & Yoshida, 2002), we also used read-write subtyping for a refined secrecy typing. While we omit this refinement for brevity, the refined secrecy typing can be easily treated using precisely the same technical development.

The significance of structural security becomes apparent when we introduce the flow inference rules for stateful actions (see also Honda & Yoshida (2002, section 6) for further discussions). A similar idea is also used in SLam-Calculus (see transparency types in the Appendix of Heintze & Riecke (1998)).

Using the extended syntax, we illustrate the subtlety in the flows associated with stateful actions. We first show a reference (an imperative variable) is indeed both positive (i.e. giving information) and negative (i.e. receiving information).

Example 7.2 (polarities of a reference agent) First let us show a reference type is behaviourally positive in the sense of Definition 3.1, that is, two references of the same type can induce a difference in the ultimate convergent behaviour. Let $[\![\mathsf{deref}\ x]\!]_w \stackrel{\text{def}}{=} \overline{x}\,\mathtt{read}(e)e(y)\overline{y}(c)c.\overline{w}$ which is a process reading a value from a reference x. This agent has type $x : \mathsf{rw}_s\langle\overline{\star_{s'}}\rangle, w : ()^{\uparrow_{\mathrm{A}}}$. Then consider the following two processes.

$$\mathsf{Ref}\langle x()\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(\mathsf{Ref}\langle xy\rangle \mid [\![()]\!]_y) \qquad \mathsf{Ref}\langle x\Omega\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(\mathsf{Ref}\langle xy\rangle \mid \Omega_y)$$

where $[\![()]\!]_y$ and Ω_y are from Example 2.3 (3) in section 2 (while having the same type, the former converges, while the latter diverges). $\mathsf{Ref}\langle x()\rangle$ (resp. $\mathsf{Ref}\langle x\Omega\rangle$) represents a reference whose stored value is the unit (resp. the omega), with type $\mathsf{ref}_s\langle\star_{s'}\rangle$. Then these two processes do "make difference", since we have $[\![\mathsf{deref}\ x]\!]_w \mid \mathsf{Ref}\langle x()\rangle \Downarrow_w$ while $[\![\mathsf{deref}\ x]\!]_w \mid \mathsf{Ref}\langle x\Omega\rangle \Uparrow$. Hence by letting $P_1 \stackrel{\text{def}}{=} \mathsf{Ref}\langle x()\rangle$, $P_2 \stackrel{\text{def}}{=} \mathsf{Ref}\langle x\Omega\rangle$ and $R = [\![\mathsf{deref}\ x]\!]_w$, by Definition 3.1, $\mathsf{ref}_s\langle\star_{s'}\rangle$ is positive.

Next we show that a reference type is behaviourally negative too. Let us define $R \stackrel{\text{def}}{=} \mathsf{Ref}\langle x\Omega\rangle \mid [\![\mathsf{deref}\ x]\!]_w$, which is typed as $\vdash R \triangleright x : \mathsf{ref}_s\langle\star_s\rangle, w : ()^{\uparrow_{\mathrm{A}}}$, and consider the following two processes.

$$[\![x := ()]\!] \stackrel{\text{def}}{=} \overline{x}\,\mathtt{write}(vg)([\![()]\!]_v \mid g.\mathbf{0}) \qquad [\![x := \Omega]\!] \stackrel{\text{def}}{=} \overline{x}\,\mathtt{write}(vg)(\Omega_v \mid g.\mathbf{0})$$

$[\![x := ()]\!]$ (resp. $[\![x := \Omega]\!]$) represents a process (with type $x : \mathsf{rw}_s\langle\overline{\star_{s'}}\rangle$) which writes the unit (resp. the omega) to the reference $\mathsf{Ref}\langle xy\rangle$. Then $[\![x := ()]\!] \mid R \Downarrow_w$ while $[\![x := \Omega]\!] \mid R \Uparrow$. Hence $\mathsf{ref}_s\langle\star_{s'}\rangle$ is negative. Thus a reference type (hence its dual) is both positive and negative, by Definition 3.1.

Example 7.2 extends to all reference types (except those which carry types which are neither positive nor negative), so that all reference types are both positive and negative. Example 7.2 also clarifies the need of secrecy annotations for reference types and their duals: "s" in $\mathsf{ref}_s\langle\tau\rangle$ indicates the level at which the reference receives information as a negative type, via writing.

A mutable affine replication is also simultaneously both positive and negative: it emits information in the same way as its stateless counterpart. And it also receives information unlike its stateless counterpart, as the following example shows.

Example 7.3 (polarity of mutable affine replication) Let $P_1 \stackrel{\text{def}}{=} \overline{x}(c)c.\mathbf{0}$ and $P_2 \stackrel{\text{def}}{=} \mathbf{0}$. Then we have $\vdash P_{1,2} \triangleright x : \tau$ with $\tau = (()_s^{\downarrow_{\mathrm{A}}})_s^{?_{\mathrm{A}}}$. Now consider the following process:

$$R \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(!\, x(c).([\![y := ()]\!] \mid \overline{c}) \mid \mathsf{Ref}\langle y\Omega\rangle \mid [\![\mathsf{deref}\ y]\!]_w)$$

then $\vdash R \triangleright x : \overline{\tau}, w : ()_s^{\uparrow \mathrm{A}}$ for some s. We then observe: $P_1 \mid R \Downarrow_w$ while $P_2 \mid R \Uparrow$. Hence τ is behavioural positive.

Example 7.3 also shows the need for the secrecy annotation of mutable replication: s in $(\vec{\tau})_s^{!\mathrm{A}}$ is the level at which a mutable replication receives information.

7.2 Elementary state (2): Flow inference

We now introduce flow inference rules for elementary stateful actions, starting from extended channel types.

$$\begin{array}{rcl} \tau_e & ::= & \text{Extended Channel Types from Section 4.1} \\ \tau & ::= & \tau_e \mid \mathsf{r}_s\langle\tau_e\rangle \mid \mathsf{w}_s\langle\tau_e\rangle \mid \mathsf{refr}_s\langle\tau_e\rangle \mid \mathsf{refw}_s\langle\tau_e\rangle \\ & \mid & (\vec{\tau}_e *)_s^{!\mathrm{A}} \mid (\vec{\tau}_e *)_s^{?\mathrm{A}} \mid (\vec{\tau}_e *)_s^{!\mathrm{L}} \mid (\vec{\tau}_e *)_s^{?\mathrm{L}} \end{array}$$

where the four reference types are in fact abbreviations, given as follows.

- $\mathsf{r}_s\langle\tau^?\rangle$ stands for $[(\tau)^{\downarrow \mathrm{L}} \oplus *()^{\downarrow \mathrm{L}}]_s^{?\mathrm{R}}$, while $\mathsf{refr}_s\langle\tau^!\rangle$ stands for $\overline{\mathsf{r}_s\langle\overline{\tau}\rangle}$; and
- $\mathsf{w}_s\langle\tau^?\rangle$ stands for $[(*)^{\downarrow \mathrm{L}} \oplus \overline{\tau}()^{\downarrow \mathrm{L}}]_s^{?\mathrm{R}}$, while $\mathsf{refw}_s\langle\tau^!\rangle$ stands for $\overline{\mathsf{w}_s\langle\overline{\tau}\rangle}$.

Note that the extended affine replicated types contain $*$ with mode $\{\uparrow, \downarrow\}$.

The decomposition of reference types into extended channel types allows us to assign a unique polarity to each form of channel types, as follows.

- τ_e is *positive* (resp. *negative*) if it is so in section 3.2, Definition 3.3.
- $\mathsf{w}_s\langle\tau_e\rangle$, $\mathsf{refr}_s\langle\tau_e\rangle$, $(\vec{\tau}_e)_s^!$ and $(\vec{\tau}_e *)_s^?$ are *positive.*
- $\mathsf{r}_s\langle\tau_e\rangle$, $\mathsf{refw}_s\langle\tau_e\rangle$, $(\vec{\tau}_e)_s^?$, and $(\vec{\tau}_e *)_s^!$ are *negative.*

For the extended set of channel types we set the tampering levels and receiving levels. Below we assume each τ has mode $!$.

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{tamp}((\overline{\vec{\tau}}*)_s^!) & = & \mathbb{H} & \mathsf{receive}((\vec{\tau}*)_s^?) & = & \mathbb{L} \\ \mathsf{tamp}(\mathsf{refr}_s\langle\tau\rangle) & = & \mathsf{tamp}(\tau) & \mathsf{receive}(\mathsf{r}_s\langle\overline{\tau}\rangle) & = & \mathsf{receive}(\overline{\tau}) \\ \mathsf{tamp}(\mathsf{w}_s\langle\overline{\tau}\rangle) & = & s & \mathsf{receive}(\mathsf{refw}_s\langle\tau\rangle) & = & s \end{array}$$

The table above is arranged to clarify that the receiving level of some type is precisely the tampering level of its dual. The receiving level for $(\vec{\tau}\tau)_s^?$ is similarly defined as $(\vec{\tau})_s^{\downarrow \mathrm{A}}$ in section 4, i.e. $\mathsf{receive}((\vec{\tau}\tau)_s^?) = \mathsf{receive}(\tau)$.

Basic types are defined by adding $(\vec{*})_s^?$, $\mathsf{w}_s\langle *\rangle$, $(\vec{\tau}\tau)_s^!$ and $\mathsf{refr}_s\langle\tau\rangle$ with τ basic. The set of cancellable types add those of the form $\mathsf{refr}_s\langle\tau\rangle$ with τ cancellable. We observe:

Proposition 7.4 1. *If* $\tau \succ \tau'$ *then* $\mathsf{tamp}(\tau) \sqsubseteq \mathsf{tamp}(\tau')$.

2. *If* τ *is positive and not cancellable, then* $\mathsf{tamp}(\tau) \sqsubseteq \mathsf{receive}(\overline{\tau})$.
3. *If* τ *is basic and not cancellable, then* $\mathsf{receive}(\overline{\tau}) = \mathsf{tamp}(\tau)$.

Proof

By the same routine as Proposition 5.6. For (3), in addition to the equations in Proposition 5.6, we use the equations in the above table. For example we have two axioms:

$$\mathsf{receive}(\mathsf{refw}_s\langle\overline{\tau}\rangle) = s = \mathsf{tamp}(\mathsf{w}_s\langle\tau\rangle)$$
$$\mathsf{receive}((\vec{*})^{!}_{s}) = s = \mathsf{tamp}((\vec{*})^{?}_{s})$$

as well as the inductive cases. □

The additional inference rules are naturally given by considering the polarities of stateful types, cf. Examples 7.2 and 7.3. We first show the rule for the reference agent, which says that a reference emits information by getting read, and that information comes from either the initial value or the result of writing.

$$(\mathsf{Ref})\ \frac{-}{\vdash \mathsf{Ref}\langle xy\rangle \blacktriangleright y:\overline{\tau}\cdot\ x:\mathsf{refw}_s\langle *\rangle \rightsquigarrow x:\mathsf{refr}_s\langle\tau\rangle}$$

Let us see what this rule tells us about safety in information flow of a reference. Assuming τ is basic, we first observe:

$$\mathsf{receive}(\overline{\tau}) = \mathsf{tamp}(\tau) = \mathsf{tamp}(\mathsf{refr}_s\langle\tau\rangle).$$

Since $\mathsf{receive}(\mathsf{refw}_s\langle *\rangle) = s$, we can see that the sufficient and necessary condition for making the flow above secure, is $s \sqsubseteq \mathsf{tamp}(\mathsf{ref}_s\langle\tau\rangle) = \mathsf{tamp}(\tau)$. This is precisely what the structural security condition (Definition 7.1) dictates.

For the two duals of references, we have the following rules.

$$(\mathsf{Read})\ \frac{-}{\vdash \overline{x}\,\mathtt{read}\langle c\rangle \blacktriangleright x:\mathsf{r}_s\langle\overline{\tau}\rangle \rightsquigarrow c:(\tau)^{\uparrow_{\mathrm{L}}}} \qquad (\mathsf{Write})\ \frac{-}{\vdash \overline{x}\,\mathtt{write}\langle vc\rangle \blacktriangleright v:\tau \rightsquigarrow x:\mathsf{w}_s\langle\tau\rangle}$$

(Read) says the action of reading a datum from the reference that stores that datum (which would return that datum) affects c. (Write) says there is a flow from the datum to the write action and the write action immediately affects a reference (emits information) at the level specified at a reference type. There is also (Write-sig) rule (given in Figure 2), which induces a trivial information flow (note $()^{\uparrow_{\mathrm{L}}}$ is cancellable).

For mutable replication and its dual, we have the same rules as for non-mutable replication and its dual (cf. Figure 1). In addition, since a replicated input (resp. output) can also be negative (resp. positive), we have the following rules:

$$(\mathsf{In}^{!}\text{-}\mathsf{receive})\ \frac{\vdash P \blacktriangleright \Gamma\cdot\vec{y}:\vec{\tau} \rightsquigarrow w:\sigma \quad w\notin\{\vec{y}z\}}{\vdash !x(\vec{y}z).P \blacktriangleright \Gamma\cdot x:(\vec{\tau}*)^{!}_{s} \rightsquigarrow w:\sigma} \qquad (\mathsf{Out}^{?}\text{-}\mathsf{emit})\ \frac{\vdash P \blacktriangleright \bigcup_i\{\Gamma_i \rightsquigarrow y_i:\tau_i\}}{\vdash \overline{x}(\vec{y}z)P \blacktriangleright \vee_i\,\Gamma_i \rightsquigarrow x:(\vec{\tau}*)^{?}_{s}}$$

(In$^{!}$-receive) introduces an immediately receiving type, while (Out$^{?}$-emit) introduces an immediately tampering (emitting) type. These rules can be understood just as (In$^{\downarrow_{\mathrm{A}}}$) and (Out$^{\uparrow_{\mathrm{A}}}$) in section 4.2.

We summarise the additional rules for treating elementary stateful actions in Figure 2, augmenting the original rules for stateless actions in Figure 1 (all original rules stay intact).

The notion of *flow-security* is defined as before (cf. Definition 5.9), considering only basic types in targets. We observe:

(In$^!$-receive)
$$\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\tau} \rightsquigarrow w : \sigma \quad w \notin \{\vec{y}z\}}{\vdash !x(\vec{y}z).P \blacktriangleright \Gamma \cdot x : (\vec{\tau}*)^{!}_{s} \rightsquigarrow w : \sigma}$$

(Out$^?$-emit)
$$\frac{\vdash P \blacktriangleright \bigcup_i \{\Gamma_i \rightsquigarrow y_i : \tau_i\}}{\vdash \overline{x}(\vec{y}z)P \blacktriangleright \vee_i \Gamma_i \rightsquigarrow x : (\vec{\tau}*)^{?}_{s}}$$

(Ref)
$$\frac{-}{\vdash \mathsf{Ref}\langle xy\rangle \blacktriangleright y : \overline{\tau} \cdot x : \mathsf{refw}_s\langle * \rangle \rightsquigarrow x : \mathsf{refr}_s\langle \tau \rangle}$$

(Read)
$$\frac{-}{\vdash \overline{x}\,\mathtt{read}\langle c\rangle \blacktriangleright x : \mathsf{r}_s\langle \overline{\tau}\rangle \rightsquigarrow c : (\tau)^{\uparrow \mathrm{L}}}$$

(Write)
$$\frac{-}{\vdash \overline{x}\,\mathtt{write}\langle vc\rangle \blacktriangleright v : \tau \rightsquigarrow x : \mathsf{w}_s\langle \tau \rangle}$$

(Write-sig)
$$\frac{-}{\vdash \overline{x}\,\mathtt{write}\langle vc\rangle \blacktriangleright \emptyset \rightsquigarrow c : ()^{\uparrow \mathrm{L}}}$$

Fig. 2. Flow analysis for elementary stateful actions.

Proposition 7.5 *If* $\vdash_{\mathsf{sec}} P \triangleright A$ *in the elementary stateful secrecy typing in Appendix D, then* P^A *is flow-secure w.r.t. the inference rules above.*

Proof
By definition we are only interested in basic flows.

(Out$^?$-emit) As (Out$^{\uparrow \mathrm{A}}$) in Proposition 5.10.
(In$^!$-receive) As (In$^{\downarrow \mathrm{A}}$) in Proposition 5.10.
(Ref) Noting $\mathsf{refr}_s\langle\tau\rangle$ is basic if τ is, we have $\mathsf{receive}(\overline{\tau}) = \mathsf{tamp}(\tau)$. Further by structural security we have $s \sqsubseteq \mathsf{tamp}(\mathsf{ref}_s\langle\tau\rangle) = \mathsf{tamp}(\tau)$ hence done.
(Read) By $\mathsf{receive}(\mathsf{r}_s\langle\overline{\tau}\rangle) = \mathsf{receive}(\overline{\tau}) = \mathsf{tamp}((\tau)^{\uparrow \mathrm{L}})$.
(Write) Assume $\mathsf{w}_s\langle\tau\rangle$ is basic. Then $\tau = *$. Hence immediate.
(Write-sig) Trivial.

Further, we observe that the identical behavioural property for co-cancellable sources (Lemma 5.12) holds using the same liveness property of linear outputs (cf. Yoshida (2002, section 4): while the reduction in general becomes nondeterministic with state, interactions between P and R_i (if any), which are linear call sequences, are still deterministic, so this does not affect the reasoning). This also leads to an analogue of Corollary 5.13. By the same reasoning steps as in the proof for Proposition 5.15, we obtain the same noninterference theorem (Theorem 5.1).

7.3 General state (1): Indirect flows in general reference

This subsection studies information flow analysis for a general class of stateful processes, where the stateful and non-stateful types are fully integrated. The syntax of types is now extended as follows.

$$\tau \quad ::= \quad \ldots \mid \mathsf{ref}_s\langle\tau^!\rangle \mid \mathsf{rw}_s\langle\tau^?\rangle \mid (\vec{\tau}^?\tau^{\uparrow \mathrm{A}})^{!\mathrm{A}}_{s} \mid (\vec{\tau}^!\tau^{\downarrow \mathrm{A}})^{?\mathrm{A}}_{s} \mid (\vec{\tau}^?\tau^{\uparrow \mathrm{L}})^{!\mathrm{L}}_{s} \mid (\vec{\tau}^!\tau^{\downarrow \mathrm{L}})^{?\mathrm{L}}_{s}$$

Note that we now allow any type to carry stateful types. The resulting types are called *general stateful types*, or simply *stateful types*. Similarly reference types which can carry stateful types are called *general reference types*.

Having the generalised form of stateful types adds complication due to indirect flows caused by mutable types that are *carried* in other types. In fact, it makes even "purely positive" types (of mode $\uparrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}$,) negative. Similarly for the dual types. The existence of such indirect flow in general reference was, as far as we know, first observed in Volpano *et al.* (1996). We illustrate the added subtlety again by examples.

Example 7.6 (indirect flows in general references) As a typical case of indirect flows coming from general stateful types, we show that writing action, which is usually just positive, can become behaviourally negative when a reference carries another reference. This suggests an extended type $\mathsf{w}_s\langle\tau\rangle$ is not only positive but also negative when τ is mutable. First, let us define $\mathbf{T}\langle e\rangle$ (a truth agent) and $\mathbf{F}\langle e\rangle$ (a falsity agent) as $!e(c).\overline{c}\mathtt{in}_1$ and $!e(c).\overline{c}\mathtt{in}_2$, respectively. We also write $\mathtt{if}\ \mathsf{deref}\ y\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ as $\overline{y}\,\mathtt{read}(c')c'(e).\overline{e}(g)g[.P_1\ \&\ .P_2]$ (which means if a value of the reference y is truth, then behaves as P_1, otherwise behaves as P_2) and $[\![y := \mathbf{T}]\!]$ as $\overline{y}\,\mathtt{write}(ec_0)(\mathbf{T}\langle e\rangle \mid c_0.\mathbf{0})$. Now let us define:

$$\begin{aligned} P_1 &= \mathsf{Ref}\langle xz\rangle \mid \overline{x}\,\mathtt{read}(c)c(y).[\![y := \mathbf{T}]\!] \\ P_2 &= \mathsf{Ref}\langle xz\rangle \mid \overline{x}\,\mathtt{read}(c)c(y).[\![y := \mathbf{F}]\!] \\ R &= \overline{x}\,\mathtt{write}(yc)(\mathsf{Ref}\langle y\mathbf{F}\rangle \mid \mathtt{if}\ \mathsf{deref}\ y\ \mathtt{then}\ \overline{w}\ \mathtt{else}\ (\boldsymbol{\nu}\, v)(\Omega_v | \overline{v}(c)c.\overline{w})) \end{aligned}$$

where $\vdash P_{1,2} \triangleright x : \mathsf{ref}_s\langle\tau\rangle, z : \overline{\tau}$ and $\vdash R \triangleright x : \mathsf{rw}_s\langle\overline{\tau}\rangle, w : ()^{\uparrow_{\mathrm{A}}}_{s'}$ with $\tau = \mathsf{ref}_{s'}\langle([\ \oplus\ ]^{\uparrow_{\mathrm{L}}}_{s})^{!_{\mathrm{L}}}\rangle$ (observe a reference type carries another reference type). In this configuration, after the writing side accesses the reference agent at x by writing a reference storing a boolean value, if P_i first reads at x and writes at the boolean reference, then the latter can, by changing the value of the reference, communicate information to the writing side. This in effect leads to a difference in the ultimate observable at w. Operationally we have $P_1|R \Downarrow_w$ while $P_2|R \Uparrow$. Thus a writing party can indeed receive information. Noting the initial interface is only at x, the channel x should be regarded as negative for the writing party. Since the writing action is also positive just as before, we conclude that writing is of mixed polarities.

In the same way, non-mutable types can now have mixed polarities if they carry mutable types. That is, if a stateful type is carried in another type (directly or indirectly), then it can induce indirect flows of the same kind as noted above. Below we illustrate one typical such example resulting in mixed polarities of a non-mutable type which carries a mutable type, using concrete processes.

Example 7.7 (Indirect flows via carried stateful types) We show the type $\tau \stackrel{\mathrm{def}}{=} (\mathsf{rw}_s\langle\star_{s'}\rangle)^{\downarrow_{\mathrm{L}}}$ is behaviourally positive (the same is true for all types which carry this and other stateful types directly or indirectly). Let:

$$\begin{aligned} P_1 &\stackrel{\mathrm{def}}{=} x(y).[\![y := ()]\!] \qquad\qquad R \stackrel{\mathrm{def}}{=} \overline{x}(y)(\mathsf{Ref}\langle y\Omega\rangle | [\![\mathsf{deref}\ y]\!]_w) \\ P_2 &\stackrel{\mathrm{def}}{=} x(y).[\![y := \Omega]\!] \end{aligned}$$

Note $\vdash P_i \triangleright x : \tau$ while $\vdash R \triangleright x : \overline{\tau}, w : ()^{\uparrow_{\mathrm{A}}}_{s'}$. Then we can easily show $P_1 \mid R \Downarrow_w$ but $P_2 \mid R \Uparrow$, so that τ is behaviourally positive (hence $\overline{\tau}$ is behaviourally negative).

From these observations, we set all types which carry (directly or indirectly) stateful types to be both positive and negative.

The secrecy typing rules stay the same as in Appendix D, though we refine the definition of the tampering levels as follows (all positive types are given non-trivial levels). Others are from Definition 2.2.

$$\begin{array}{rclrcl}
\mathsf{tamp}(\mathsf{ref}_s\langle\tau\rangle) & = & \mathsf{tamp}(\tau) \sqcap \mathsf{tamp}(\overline{\tau}) & \mathsf{tamp}(\mathsf{rw}_s\langle\tau\rangle) & = & s \\
\mathsf{tamp}((\vec{\tau})^{!}_{s}) & = & \sqcap\{\mathsf{tamp}(\tau_i)\} & \mathsf{tamp}((\vec{\tau})^{?}_{s}) & = & s \\
\mathsf{tamp}((\vec{\tau})^{!}) & = & \sqcap_i\{\mathsf{tamp}(\tau_i)\} & \mathsf{tamp}((\vec{\tau})^{\downarrow_{\mathrm{L}}}) & = & \sqcap_i\{\mathsf{tamp}(\tau_i)\}
\end{array}$$

We can check that this definition and the previous one in section 7.1 coincide for elementary stateful types. Some illustrations of the refinement would be due:

- The tamper level of $\mathsf{ref}_s\langle\tau^{!}\rangle$, or the level of information this type produces, is that of τ itself in the elementary case. This is because τ is what a reference of this type returns when it is read. However if τ is mutable (for example is a reference type), there is another way a reference of the type $\mathsf{ref}_s\langle\tau^{!}\rangle$ can produce information as illustrated in Example 7.6.
- For $\mathsf{rw}_s\langle\tau\rangle$, the tamper level of a process which writes to a reference of level s is s.
- Since τ_i in $(\vec{\tau})^{\downarrow_{\mathrm{L}}}$ and $(\vec{\tau}\tau)^{!}$ may be mutable, we need to accumulate the tampering level of τ_i.

The extended channel types now include $*$ with modes $\uparrow$ and $\downarrow$:

$$\tau \quad ::= \quad \ldots. \quad | \quad *^{p} \quad | \quad (\vec{\tau})^{p}$$

where p is any mode. Note that the above syntax includes $(\vec{\tau})^{p}$ with $p \in \{\downarrow_{\mathrm{A}}, \uparrow_{\mathrm{A}}, \mathbf{!}, \mathbf{?}\}$, which are used as glues between two flows (hence we ignore their security levels, and calculate their tampering and receiving levels just like their linear counterparts). We first review polarities of extended types.

- $(\vec{\tau})^{\downarrow_{\mathrm{L}}}$, $(\vec{\tau})^{\uparrow_{\mathrm{L}}}$, $(\vec{\tau})^{\downarrow_{\mathrm{A}}}_{s}$, $(\vec{\tau})^{\uparrow_{\mathrm{A}}}$, $(\vec{\tau})^{!}_{s}$, $(\vec{\tau})^{?}$, $\mathsf{r}_s\langle\tau\rangle$, $\mathsf{w}_s\langle\tau\rangle$, $\mathsf{refr}_s\langle\tau\rangle$ and $\mathsf{refw}_s\langle\tau\rangle$ are negative.
- $(\vec{\tau})^{\downarrow_{\mathrm{L}}}$, $(\vec{\tau})^{\uparrow_{\mathrm{L}}}$, $(\vec{\tau})^{\downarrow_{\mathrm{A}}}$, $(\vec{\tau})^{\uparrow_{\mathrm{A}}}_{s}$, $(\vec{\tau})^{!}$, $(\vec{\tau})^{?}_{s}$, $\mathsf{r}_s\langle\tau\rangle$, $\mathsf{w}_s\langle\tau\rangle$, $\mathsf{refr}_s\langle\tau\rangle$ and $\mathsf{refw}_s\langle\tau\rangle$ are *positive*.

Note most types are both positive and negative; however we distinguish polarities of **!?**-types by the presence/existence of secrecy annotations.

The grammar of basic types becomes simpler (due to decomposition of flow inference rules on which we shall discuss in the next subsection). We say τ is basic iff:

i. $\tau = (\vec{*}\rho\vec{*})^{p}$ where ρ is basic and $p \in \{\uparrow_{\mathrm{L}}, \downarrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}, \downarrow_{\mathrm{A}}, \mathbf{!}, \mathbf{?}\}$; or
ii. $\tau = ()^{\uparrow_{\mathrm{L}}}$, $\tau = (\vec{*})^{\uparrow_{\mathrm{A}}}_{s}$ or $\tau = (\vec{*})^{?}_{s}$ is basic.
iii. $\mathsf{w}_s\langle * \rangle$ and $\mathsf{refr}_s\langle\tau\rangle$ are basic if τ is basic.

A flow $\Gamma \rightsquigarrow x : \tau$ is *basic* if τ is basic and Γ is minimum among the sources of $x : \tau$ (whose existence we can check as before).

The grammar of cancellable types is precisely the same as in elementary stateful types.

As before, we only treat structurally secure types. Then the following tampering/receiving levels are added to the table given in section 7.2.

$$\begin{array}{rclcrcl}
\mathsf{tamp}(\mathsf{refw}_s\langle\tau\rangle) & = & \mathsf{tamp}(\overline{\tau}) & \quad & \mathsf{receive}(\mathsf{w}_s\langle\overline{\tau}\rangle) & = & \mathsf{receive}(\tau) \\
\mathsf{tamp}(\mathsf{r}_s\langle\overline{\tau}\rangle) & = & \mathsf{tamp}(\overline{\tau}) & & \mathsf{receive}(\mathsf{refr}_s\langle\tau\rangle) & = & \mathsf{receive}(\tau)
\end{array}$$

We also define the receiving level for other types as follows.

- $\mathsf{receive}((\vec{\tau})^{\downarrow_{\mathrm{L}}}) = \mathsf{receive}((\vec{\tau})^{!}) = \mathsf{receive}((\vec{\tau})^{\uparrow}) = \mathsf{receive}((\vec{\tau})_s^{\uparrow_{\mathrm{A}}}) = \mathsf{receive}((\vec{\tau})^{?}) = \mathsf{receive}((\vec{\tau})_s^{?}) = \sqcup_i\{\mathsf{receive}(\tau_i)\}$
- $\mathsf{receive}((\vec{\tau})_s^{\downarrow_{\mathrm{A}}}) = \mathsf{receive}((\vec{\tau})_s^{!}) = s$

We can again mechanically check:

Proposition 7.8 1. *If* $\tau \succ \tau'$ *then* $\mathsf{tamp}(\tau) \sqsubseteq \mathsf{tamp}(\tau')$.

2. *If* τ *is positive and not cancellable, then* $\mathsf{tamp}(\tau) \sqsubseteq \mathsf{receive}(\overline{\tau})$.

3. *If* τ *is basic and not cancellable, then* $\mathsf{receive}(\overline{\tau}) = \mathsf{tamp}(\tau)$.

7.4 General state (2): Flow inference

This subsection introduces the flow inference rules for general stateful actions. In this generalised setting, flows can be from arbitrary free names to arbitrary free names in a process, due to mixed polarities. For this reason, it becomes more convenient to present flow inference rules for prefixed agents in a form which is more analytic (or more atomic) than before. The resulting rules can be seen as a uniform decomposition of the original flow rules into more atomic ones.[4]

Because of the decomposition, all prefix rules now have a uniform shape.

$$\begin{array}{ccc}
(\mathsf{In}^{\downarrow_{\mathrm{L}}}\text{-}\mathsf{receive})\ \ (w \notin \{\vec{y}\}) & \qquad & (\mathsf{In}^{\downarrow_{\mathrm{L}}}\text{-}\mathsf{emit}) \\
\dfrac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\rho} \leadsto w : \tau}{\vdash x(\vec{y}).P \blacktriangleright \Gamma \cdot x : (\vec{\rho})^{\downarrow_{\mathrm{L}}} \leadsto w : \tau} & & \dfrac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\rho} \leadsto y_i : \tau}{\vdash x(\vec{y}).P \blacktriangleright \Gamma \cdot x : (\vec{*}\tau\vec{*})^{\downarrow_{\mathrm{L}}}\ \text{}}
\end{array}$$

The first rule should look familiar: it says that if the body of a prefixed process has a flow from $\vec{y}$ (to be abstracted by a prefix) and Γ to the designated target w, then the prefixed process has a flow from the result of abstracting these names at x and Γ, to the same target w. In the second rule, (In$^{\downarrow_{\mathrm{L}}}$-emit), we infer a flow for the same prefixed process but to its own subject (i.e. the initial name in the prefix) as a target,

[4] In fact, we can reconstruct the flow inference rules in π^{LA} and its extension to elementary state from these decomposed rules; we can recover the original set of flows by deleting superfluous ones using the fixed polarities in π^{LA}. See Remark 7.10 for further discussions.

and has a different shape from those rules we have seen so far. In detail, the rule says that:

If the body P has a flow from $\vec{y}$ (to be abstracted) and Γ, to one of the channels, here y_i, to be abstracted, then the prefixed process as a whole has a flow from the result of abstracting $\vec{y}$ at x and Γ, to the result of abstracting y_i at x.

Some observations:

1. Having a common channel (in this case y_i) on both sides of a flow is not only necessary for analysing stateful actions (as we have already seen in section 7.2), but is also consistent with flow analyses in π^{LA} and its stateful extension. Remark 7.10 presents a non-trivial example which relates this double-sided flow inference to the original flow inference in π^{LA}.
2. The rule only treats the case when a target type in the conclusion carries the type of a single component of abstracted names, y_i. This does not lead to a loss of generality since, by the rule for super-imposition (Super) discussed later, we can always super-impose multiple flows to the same target channel, to make its type “bigger”.

Note that, in the first rule, $x : (\vec{\rho})^{\downarrow_{\mathsf{L}}}$ occurs in the source: thus this type is used *negatively*, which indicates we measure this type in terms of its receiving level. On the other hand, in the second rule, $x : (\vec{*}\tau\vec{*})^{\downarrow_{\mathsf{L}}}$ occurs *positively*, indicating we should measure this type in terms of the tampering level of τ. In this way, polarities now arise as the roles of typed channels in flow analyses, giving again essential information.

Reflecting mixed polarities, the rules for linear output are constructed in the way identical to those for linear input.

$$\text{(Out}^{\uparrow_{\mathsf{L}}}\text{-receive)}\ (w \notin \{\vec{y}\})\quad \frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\rho} \rightsquigarrow w : \tau}{\vdash \overline{x}(\vec{y})P \blacktriangleright \Gamma \cdot x : (\vec{\rho})^{\uparrow_{\mathsf{L}}} \rightsquigarrow w : \tau} \qquad \text{(Out}^{\uparrow_{\mathsf{L}}}\text{-emit)}\quad \frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y} : \vec{\rho} \rightsquigarrow y_i : \tau}{\vdash x(\vec{y}).P \blacktriangleright \Gamma \cdot x : (\vec{\rho})^{\uparrow_{\mathsf{L}}} \rightsquigarrow x : (\vec{*}\tau\vec{*})^{\uparrow_{\mathsf{L}}}}$$

These rules are understood just as the rules for linear input. Similarly, we have the rules for affine input and output rules (with appropriate secrecy annotations, as noted in Figure 3). Replicated prefix rules (except those for the reference and its duals, which we treat next) are similarly defined.

For reference agents, we have precisely the same rule as before (cf. Figure 2), as well as the following two additional rules.

$$\text{(Ref-reverse)}\quad \frac{-}{\vdash \mathsf{Ref}\langle xy\rangle \blacktriangleright x : \mathsf{refr}_s\langle\tau\rangle \rightsquigarrow x : \mathsf{refw}_s\langle\tau\rangle} \qquad \text{(Ref-leak)}\quad \frac{-}{\vdash \mathsf{Ref}\langle xy\rangle \blacktriangleright x : \mathsf{refr}_s\langle\tau\rangle \rightsquigarrow y : \overline{\tau}}$$

In (Ref-reverse), we capture the situation where a reference agent emits information when it is written. As we illustrated in Example 7.6, when a process writes to a mutable datum (say a name of another reference) to a reference and another process reads that datum and changes it, then the reference is in effect mediating a flow

(In$^{\downarrow\mathrm{L}}$-receive) $(w \notin \{\vec{y}\})$
$$\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y}:\vec{\rho} \rightsquigarrow w:\tau}{\vdash x(\vec{y}).P \blacktriangleright \Gamma \cdot x:(\vec{\rho})^{\downarrow\mathrm{L}} \rightsquigarrow w:\tau}$$

(In$^{\downarrow\mathrm{L}}$-emit)
$$\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y}:\vec{\rho} \rightsquigarrow y_i:\tau}{\vdash x(\vec{y}).P \blacktriangleright \Gamma \cdot x:(\vec{\rho})^{\downarrow\mathrm{L}} \rightsquigarrow x:(\vec{*}\tau\vec{*})^{\downarrow\mathrm{L}}}$$

(Out$^{\uparrow\mathrm{L}}$-receive) $(w \notin \{\vec{y}\})$
$$\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y}:\vec{\rho} \rightsquigarrow w:\tau}{\vdash \overline{x}(\vec{y})P \blacktriangleright \Gamma \cdot x:(\vec{\rho})^{\uparrow\mathrm{L}} \rightsquigarrow w:\tau}$$

(Out$^{\uparrow\mathrm{L}}$-emit)
$$\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y}:\vec{\rho} \rightsquigarrow y_i:\tau}{\vdash x(\vec{y}).P \blacktriangleright \Gamma \cdot x:(\vec{\rho})^{\uparrow\mathrm{L}} \rightsquigarrow x:(\vec{*}\tau\vec{*})^{\uparrow\mathrm{L}}}$$

(In$^{\downarrow\mathrm{A}}$-receive) $(w \notin \{\vec{y}\})$
$$\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y}:\vec{\rho} \rightsquigarrow w:\tau}{\vdash\ !x(\vec{y}).P \blacktriangleright \Gamma \cdot x:(\vec{\rho})^{\downarrow\mathrm{A}} \rightsquigarrow w:\tau}$$

(In$^{\downarrow\mathrm{A}}$-emit)
$$\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y}:\vec{\rho} \rightsquigarrow y_i:\tau}{\vdash\ !x(\vec{y}).P \blacktriangleright \Gamma \cdot x:(\vec{\rho})^{\downarrow\mathrm{A}} \rightsquigarrow x:(\vec{*}\tau\vec{*})^{\downarrow\mathrm{A}}}$$

(Out$^{\uparrow\mathrm{A}}$-receive) $(w \notin \{\vec{y}\})$
$$\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y}:\vec{\rho} \rightsquigarrow w:\tau}{\vdash \overline{x}(\vec{y})P \blacktriangleright \Gamma \cdot x:(\vec{\rho})^{\uparrow\mathrm{A}} \rightsquigarrow w:\tau}$$

(Out$^{\uparrow\mathrm{A}}$-emit)
$$\frac{\vdash P \blacktriangleright \Gamma \cdot \vec{y}:\vec{\rho} \rightsquigarrow y_i:\tau}{\vdash \overline{x}(\vec{y})P \blacktriangleright \Gamma \cdot x:(\vec{\rho})^{\uparrow\mathrm{A}} \rightsquigarrow x:(\vec{*}\tau\vec{*})^{\uparrow\mathrm{A}}}$$

(Ref-reverse)
$$\frac{-}{\vdash \mathsf{Ref}\langle xy\rangle \blacktriangleright x:\mathsf{refr}_s\langle\tau\rangle \rightsquigarrow x:\mathsf{refw}_s\langle\tau\rangle}$$

(Ref-leak)
$$\frac{-}{\vdash \mathsf{Ref}\langle xy\rangle \blacktriangleright x:\mathsf{refr}_s\langle\tau\rangle \rightsquigarrow y:\overline{\tau}}$$

(Read-leak)
$$\frac{-}{\vdash \overline{x}\,\mathtt{read}\langle c\rangle \blacktriangleright c:(\tau)^{\uparrow\mathrm{L}} \rightsquigarrow x:\mathsf{r}_s\langle\overline{\tau}\rangle}$$

(Write-reverse)
$$\frac{-}{\vdash \overline{x}\,\mathtt{write}\langle vc\rangle \blacktriangleright x:\mathsf{w}_s\langle\tau\rangle \rightsquigarrow v:\tau}$$

(Super)
$$\frac{\vdash P^A \blacktriangleright \Gamma_i \rightsquigarrow x:\tau_i}{\vdash P^A \blacktriangleright \vee\Gamma_i \rightsquigarrow x:\vee\tau_i}$$

(Weak-$*$)
$$\frac{\vdash P^A \blacktriangleright \Gamma^{-x} \rightsquigarrow y:\tau}{\vdash P^A \blacktriangleright \Gamma \cdot x:* \rightsquigarrow y:\tau}$$

- We also add (Zero, Par, Res, Weak, Union, Subset, Empty) from Figure 1 and (Ref, Read, Write, Write-sig) from Figure 2.

Fig. 3. Flow analysis for general stateful actions.

from the reading party to the writing party. Similarly, (Ref-leak) depicts the situation where the reader of the content, which is a mutable datum, gives information by changing it, emitted via that datum. Note the resulting flows become insignificant when we consider them in the context of elementary stateful agents.

For read/write actions, in addition to the preceding rules, there are two additional rules which treat indirect flows.

(Read-leak)

$$\frac{-}{\vdash \overline{x}\,\mathtt{read}\langle c\rangle \blacktriangleright c:(\tau)^{\uparrow \mathrm{L}} \rightsquigarrow x:\mathsf{r}_s\langle\overline{\tau}\rangle}$$

(Write-reverse)

$$\frac{-}{\vdash \overline{x}\,\mathtt{write}\langle vc\rangle \blacktriangleright x:\mathsf{w}_s\langle\tau\rangle \rightsquigarrow v:\tau}$$

In (Read-leak), we introduce the reading action in the target, rather than in the source. The rule is the dual of (Ref-leak), saying that the reading action may give effects by tampering the datum it has read. Similarly, (Write-reverse) is the dual of (Ref-reverse), saying the write action can be a source of another action since, after writing a name of a mutable datum (say another reference), another process may as well read that name and tampers it, so that the write action is indirectly the source of information change.

The composition rules, (Zero), (Par) and (Res) are as before, together with (Cancel) in section 6.2 and the refinement of (Par) as stipulated in section 6.2 (which prohibits composition at co-cancellable channels).

Finally, for structural rules, we use the same rules (Union), (Subset) and (Empty) in Figure 1: in addition, we also have two additional structural rules, which play an essential role in the present flow analysis.

(Super)

$$\frac{\vdash P^A \blacktriangleright \Gamma_i \rightsquigarrow x:\tau_i}{\vdash P^A \blacktriangleright \ \vee\Gamma_i \rightsquigarrow x:\vee\tau_i}$$

(Weak-$*$)

$$\frac{\vdash P^A \blacktriangleright \Gamma^{-x} \rightsquigarrow y:\tau}{\vdash P^A \blacktriangleright \Gamma\cdot x:* \rightsquigarrow y:\tau}$$

The rule (Super) does super-imposition: if we have two or more flows with the same target channel x, then (noting that all these types are lesser than the type of x in the original typing of the process), we can take their lub (cf. Proposition 4.2). In this way, one can always generate an appropriate target type which is necessary (only) for compensating its dual in the source in (Par). (Weak-$*$) simply adds an insignificant typed channel in the source, which is semantically innocuous. This rule is needed since prefix rules assume all abstracted names to be present in the source.

As in section 7.2, we assume the use of (Cancel) and the refined composition in (Par), as stipulated in section 6.2.

We summarise the additional rules for general stateful actions in Figure 3. We only present the prefix rules for linear input/output and linear replicated input/output: as noted already, their affine versions are identical except that we introduce a secrecy level as appropriate, as noted at the end.

We conclude this section by checking the same properties as before, but for the generalised stateful types. The secrecy typing is given in Appendix D.

Proposition 7.9 *If* $\vdash_{\mathsf{sec}} P \triangleright A$ *in the stateful secrecy typing in Appendix D, then* P^A *is flow-secure.*

Proof
By inspecting each rule. The reasoning for the rules we have checked in Proposition 7.5 remains the same.

($\mathsf{In}^{\downarrow_{\mathrm{L}}}/\mathsf{Out}^{\uparrow_{\mathrm{L}}}$-receive/emit) By inductive hypothesis.
($\mathsf{In}^{\downarrow_{\mathrm{A}}}$-receive) As ($\mathrm{In}^{\downarrow_{\mathrm{A}}}$) in Proposition 5.10.
($\mathsf{In}^{\downarrow_{\mathrm{A}}}$-emit) By inductive hypothesis, $\mathsf{receive}(\Gamma\cdot\vec{y}:\vec{\rho})\sqsubseteq\mathsf{tamp}(\tau)$. Also we note $\mathsf{receive}((\vec{\rho})^{\downarrow_{\mathrm{A}}})=\sqcup_i\{\mathsf{receive}(\rho_i)\}$. Hence $\mathsf{receive}(\Gamma\cdot x:(\vec{\rho})^{\downarrow_{\mathrm{A}}})=\mathsf{receive}(\Gamma)\sqcup(\sqcup_i\{\mathsf{receive}(\rho_i)\})\sqsubseteq\mathsf{tamp}(\tau)=\mathsf{tamp}(x:(\vec{*}\tau\vec{*})^{\downarrow_{\mathrm{A}}}_s)$, as required.
($\mathsf{Out}^{\uparrow_{\mathrm{A}}}$-receive) By inductive hypothesis.
($\mathsf{Out}^{\uparrow_{\mathrm{A}}}$-emit) As ($\mathrm{Out}^{\uparrow_{\mathrm{A}}}$) in Proposition 5.10.
($\mathsf{In}^{!}$-receive) As ($\mathrm{In}^{\downarrow_{\mathrm{A}}}$) in Proposition 5.10.
($\mathsf{In}^{!}$-emit) As ($\mathrm{In}^{\downarrow_{\mathrm{A}}}$-emit).
($\mathsf{Out}^{?}$-receive/emit) As ($\mathrm{Out}^{\uparrow_{\mathrm{A}}}$-receive/emit).
(Ref-reverse) By $\mathsf{receive}(\mathsf{refw}_s\langle\tau\rangle)=\mathsf{receive}(\tau)=\mathsf{tamp}(\overline{\tau})=\mathsf{tamp}(\mathsf{refr}_s\langle\overline{\tau}\rangle)$.
(Ref-leak) Similar to (Ref-reverse).
(Read-leak) By $\mathsf{receive}((\tau)^{\downarrow_{\mathrm{L}}})=\mathsf{receive}(\tau)=\mathsf{tamp}(\overline{\tau})=\mathsf{tamp}(\mathsf{r}_s\langle\overline{\tau}\rangle)$.
(Write-reverse) Similar to (Read-leak).
(Super/Weak-$*$) Trivial.

The same behavioural security as before, as well as an analogue of its corollary, is easily obtained by the linear liveness property in this extension. Hence, by the above proposition, we can finally achieve noninterference (Theorem 5.1) for general stateful processes.

Remark 7.10 (relationship to the preceding systems) Let us see a concrete example of how the equivalent of a non-trivial rule in the flow analysis in Section 4 can be precisely derived in the present system. We take the example from Example 4.11 (at the end of section 4). For the replicated agent, we infer:

$$\frac{\vdash \overline{u}(c)c^{\mathbb{H}}.\overline{z}^{\mathbb{L}} \blacktriangleright u:\overline{\star_{\mathbb{H}}}\cdot c:* \rightsquigarrow z:()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}}}{\vdash !x(uz).\overline{u}(c)c^{\mathbb{H}}.\overline{z}^{\mathbb{L}} \blacktriangleright x:(\overline{\star_{\mathbb{H}}}\,*)^{!_{\mathrm{L}}} \rightsquigarrow x:(*()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}})^{!_{\mathrm{L}}}}\ (\mathsf{In}^{!_{\mathrm{L}}}\text{-emit})$$

whereas, for the dual process, we use ($\mathrm{Out}^{?_{\mathrm{L}}}$-emit) and ($\mathrm{Out}^{?_{\mathrm{L}}}$-receive) for the same process, combined with (Union). First by (Empty), we have: $\vdash z.\overline{w}\blacktriangleright\emptyset\rightsquigarrow z:*$. Then by applying ($\mathrm{Out}^{?_{\mathrm{L}}}$-emit), we have:

$$\frac{\vdash [u^{\mathbb{H}}\rightarrow v^{\mathbb{H}}]\mid z.\overline{w}^{\mathbb{L}} \blacktriangleright v:\overline{\star}_{\mathbb{H}} \rightsquigarrow u:\star_{\mathbb{H}},\emptyset\rightsquigarrow z:*}{\vdash \overline{x}(uz)([u^{\mathbb{H}}\rightarrow v^{\mathbb{H}}]|z.\overline{w}^{\mathbb{L}}) \blacktriangleright v:\overline{\star_{\mathbb{H}}} \rightsquigarrow x:(\star_{\mathbb{H}}\,*)^{?_{\mathrm{L}}}}\ (\mathsf{Out}^{?_{\mathrm{L}}}\text{-emit})$$

For the same output process, we have the following another inference:

$$\frac{\vdash [u^{\mathbb{H}}\rightarrow v^{\mathbb{H}}]\mid z.\overline{w}^{\mathbb{L}} \blacktriangleright u:*\cdot z:()^{\downarrow_{\mathrm{A}}}_{\mathbb{L}} \rightsquigarrow w:()^{\downarrow_{\mathrm{A}}}_{\mathbb{L}}}{\vdash \overline{x}(uz)([u^{\mathbb{H}}\rightarrow v^{\mathbb{H}}]|z.\overline{w}^{\mathbb{L}}) \blacktriangleright x:(*\ ()^{\downarrow_{\mathrm{A}}}_{\mathbb{L}})^{?_{\mathrm{L}}} \rightsquigarrow w:()^{\uparrow_{\mathrm{A}}}_{\mathbb{L}}}\ (\mathsf{Out}^{?_{\mathrm{L}}}\text{-receive})$$

Thus, via (Union), we obtain:

$$\vdash \overline{x}(uz)([u^{\mathbb{H}} \to v^{\mathbb{H}}]|z.\overline{w}^{\mathbb{L}}) \blacktriangleright v:\overline{\star_{\mathbb{H}}} \rightsquigarrow x:(\star_{\mathbb{H}}\,*)^{?_{\mathrm{L}}},\ x:(*\ ()_{\mathbb{L}}^{\downarrow_{\mathrm{A}}})^{?_{\mathrm{L}}} \rightsquigarrow w:()_{\mathbb{L}}^{\uparrow_{\mathrm{A}}}$$

which, when compensated in (Par) with the flow for the input process above, leads to the required (unsafe) flow, $v:\overline{\star_{\mathbb{H}}} \rightsquigarrow w:()_{\mathbb{L}}^{\uparrow_{\mathrm{A}}}$. Note how the flow inference precisely captures individual causal flows together with their composition in a fine-grained fashion. On the other hand, the inference rules for π^{LA} in Figure 1, which are used in the original inference in Example 4.11, give a direct inference of a complicated flow exploiting the fixed polarities in π^{LA}.

8 Discussions

This paper proposes a uniform framework of information flow analysis of stateless and stateful typed π-calculi. Non-interference theorems of secure versions of these calculi are proved based on an inductive flow analysis. The analysis is crucially based on the use of polarities, which signify directions of information flows at given typed channels: in the case of stateless processes, the polarities precisely dictate the position of typed channels in flows (in the source or in the target), determining the shape of flow inference rules. In the case of stateful processes, typed channels have mixed polarities in the sense that a channel can appear both in the source and in the target, which necessitates fine-grained inference rules. Polarities represent, in this case, the role of a typed channel in each flow. In the following, we conclude the paper with comparisons with related work and further issues.

8.1 Comparisons with related work

There are several proof techniques which have been used for proving noninterference. The notion of noninterference was first proposed by Goguen & Meseguer (1982) in the context of sorted algebras. Sabelfeld & Sands (1999) gave a denotational proof method. Abadi and others used a method based on logical relations for proving the noninterference in Dependency Core Calculus (DCC) (Abadi *et al*., 1999). Smith & Volpano (1998) employed operational techniques in a series of their work on imperative secrecy. Hennessy (2003) and Hennessy & Riely (2000) used May and Must behavioural equivalences in the context of secrecy-enriched π-calculi. Pottier & Simonet (2002) proposed a proof method based on an operational analysis of typed reduction, showing the designated low-level value is never affected by any high-level substitutions using extended syntax. Pottier also applied the framework to the interference proofs in the (untyped) π-calculus (Pottier, 2002). We ourselves presented a bisimulation-based method for the linear/affine π-calculus (Yoshida *et al*., 2002). Novel features of the proposed method in comparison with these foregoing studies may be summarised as follows.

- Our method is based on inductive extraction of information flow, and as such, justification of a secrecy analysis can be done via embedding into the flow analysis (essentially simple rule induction), assuming noninterference for the flow analysis has once and for all been established. Further our method does *not* require the target secrecy typing to satisfy subject reduction (unlike many of the preceding methods).

- Noninterference for flow-secure processes is essentially transparent from the shape of the flow analysis rules.
- The flow analysis and noninterference proofs are carried out in the uniform framework of (typed) name passing processes and their dynamics. This makes associated proofs quite elementary, while, via encodings, the results can in principle be carried over to standard programming language constructs.

Our work shares the same target untyped calculus for proving noninterference as that of Hennessy & Riely (2000) and Hennessy (2003). It is notable that the notions of secrecy are close in these two works, while the employed type structures are quite different. A base typing system without secrecy annotations in Hennessy & Riely (2000) and Hennessy (2003) has more typable terms, while it is often the case that processes are secure in the present work but not in Hennessy & Riely (2000) and Hennessy (2003). The inductive flow analysis, the main feature of the present work, is not studied in Hennessy & Riely (2000). In comparison with Pottier (2002), our method differs in that it does not presuppose subject reduction of the target secrecy discipline (for example DCC does not satisfy subject reduction, as discussed in Honda & Yoshida (2002), but still guarantees secrecy). In comparison with Yoshida *et al.* (2002), the main difference is syntactic nature of the analysis, which we discuss in section 8.2.

There are other flow analyses in programming languages such as *control flow analysis*, which traces how a thread of execution goes through a program text including procedure calls, and *data flow analysis*, which traces how a datum propagates within a program text via assignment and procedural calls (Nielson *et al.*, 1999). Bodei and her colleagues first studied a control flow analysis of a untyped π-calculus in Bodei *et al.* (1998) which can determine a superset of the set of names to which a given name may be bound during execution of processes (thus offering a sound approximation of the latter). Later they refined their framework to guarantee that once processes are given levels of security clearance, a process at a high level never sends names to processes at a lower level Bodei *et al.* (1999). The presented flow analysis is more fine-grained due to linear/affine types and refined tracking of causality between channels. More generally, it appears that a thread of interactions traced in our flow analysis would be related with both data and control flows. A detailed comparison with control/data and other notions of flows studied in program analyses community, including algorithmic aspects of the proposed flow analysis for the practice, would be an interesting and valuable topic for further study. As a preliminary observation, a close look reveals that control flow and data flow in programming languages interact in a subtle way in programs, for example when a program assigns a boolean value to a variable used in the guard of a conditional statement. We may ask whether such dependency in programs can be captured uniformly in the flow analysis based on, or extending, the present framework. Regarding this point we share the motivation with Abadi *et al.* (1999) (which uses DCC for capturing various program analyses), expanding the scope by the use of the π-calculus and explicit extraction of information flow.

A notion of information flow for pure functions has been studied under the name of "paths" in the context of Proof Nets (Asperti *et al.*, 1994), whose precise connection with the present framework is another interesting topic.

8.2 Limitations of the present work and further topics

The main limitation of the proposed framework of flow analysis is its syntactic nature. Consider the following popular example of semantically safe flow:

$$\text{if } b^{\mathbb{H}} \text{ then } P \text{ else } P$$

with the obvious translation of conditional into a branching, and assuming P outputs via a low-level channel c. We can check that there is no semantic flow in the sense of Definition 5.11: however the flow inference rules do extract a non-trivial flow from a high-level channel b to a low-level channel c. In this case, even though there *is* a causal flow from b to c, this flow gets blurred by the fact that the dependency is not observable to the outside observers.

To capture such blurring and other semantic effects, one may make resort to arguments based on behavioural equivalences. Our preceding work (Yoshida *et al.*, 2002) presents a semantic framework for guaranteeing and establishing secrecy and noninterference based on a new theory of typed bisimulation, which can be used for reasoning about a process as given above, proving the noninterference theorem up to the secure sensitive contextual congruence for securely typed processes. Semantically based secrecy is also studied in Sabelfeld & Sands (1999) and, more recently, in Mantel & Sabelfeld (2003).

Another limitation of the present approach is that the notion of flow does not involve timing information (Agat, 2000). It is interesting to see if the flow analysis following the present framework can be extended to incorporate this and other refinements, so that it can cover a sufficiently large class of process behaviours.

Finally, in the present framework, once we know that no insecure flow of a given process is inferable, its non-interference property is automatically guaranteed (this is formalised as a completeness property in Corollary 5.13). Thus, from a practical viewpoint, it is worth studying efficient algorithms which produce all possible flows of a given process derivable by the flow inference rules in this paper. Solving these issues will become especially important for applying the proposed framework for the flow analyses of realistic programming constructs via embedding into typed name passing processes (which may also suggest the construction of a flow analysis for a target programming language).

A Reduction

The relation $\equiv$ is the least congruence generated by $\equiv_\alpha$ and the following equations.

$$P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R)$$

$$(\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu}\, xy)P \equiv (\boldsymbol{\nu}\, yx)P \qquad ((\boldsymbol{\nu}\, x)P)|Q \equiv (\boldsymbol{\nu}\, x)(P|Q) \quad (x \notin \mathsf{fn}(Q))$$

The relation $\longrightarrow$ is generated by the following rules.

$$
\begin{array}{rl}
(\textsc{Com}) & x(\vec{y}).P \mid \overline{x}(\vec{y})Q \longrightarrow (\boldsymbol{\nu}\, \vec{y})(P|Q) \\[4pt]
(\textsc{Com}_!) & !\, x(\vec{y}).P \mid \overline{x}(\vec{y})Q \longrightarrow !\, x(\vec{y}).P|(\boldsymbol{\nu}\, \vec{y})(P|Q) \\[4pt]
(\textsc{Res}) & P \longrightarrow Q \implies (\boldsymbol{\nu}\, x)P \longrightarrow (\boldsymbol{\nu}\, x)Q \\[4pt]
(\textsc{Par}) & P \longrightarrow P' \implies P|Q \longrightarrow P'|Q \\[4pt]
(\textsc{Cong}) & P \equiv P' \longrightarrow Q' \equiv Q \implies P \longrightarrow Q
\end{array}
$$

B Linear/affine typing

B.1 *Action types*

$\mathsf{fn}(A)$ and $|A|$ denote the sets of free names and nodes in A, respectively. We write $x \rightarrow y$ if $x : \tau \rightarrow y : \tau'$ for some τ and τ', in a given action type.

We define $A \asymp B$ iff:

- whenever $x : \tau \in A$ and $x : \tau' \in B$, $\tau \odot \tau'$ is defined; and
- whenever $x_1 \rightarrow x_2,\ x_2 \rightarrow x_3,\ \ldots,\ x_{n-1} \rightarrow x_n$ alternately in A and B $(n \geqslant 2)$, we have $x_1 \neq x_n$.

Then $A \odot B$, defined iff $A \asymp B$, is the following action type.

- $x : \tau \in |A \odot B|$ iff either (1) $x \in (\mathsf{fn}(A) \backslash \mathsf{fn}(B)) \cup (\mathsf{fn}(B) \backslash \mathsf{fn}(A))$ and $x : \tau$ occurs in A or B; or (2) $x : \tau' \in A$ and $x : \tau'' \in B$ and $\tau = \tau' \odot \tau''$.
- $x \rightarrow y$ in $A \odot B$ iff $x : \tau_{\text{I}}, y : \tau_{\text{O}} \in |A \odot B|$ and $x = z_1 \rightarrow z_2,\ z_2 \rightarrow z_3, \ldots, z_{n-1} \rightarrow z_n = y$ $(n \geqslant 2)$ alternately in A and B.

We use the following notations.

- $A^{x:\tau}$ indicates A such that $x : \tau \in A$, and A^{-x} indicates A such that $x \notin \mathsf{fn}(A)$.
- $\vec{p}A$ indicates A such that $\mathsf{md}(A) \subset \{\vec{p}\}$, and $\mathbf{?}A$ indicates A such that $\mathsf{md}(A) \subset \mathcal{M}_{?}$.
- A, B denotes the disjoint union of A and B, indicating at the same time $\mathsf{fn}(A) \cap \mathsf{fn}(B) = \emptyset$.
- The *hiding* $A/\vec{x}$ is the result of taking off nodes with names in $\vec{x}$ from A.
- The *prefix* $x : \tau \rightarrow A$ adds an edge from a new node $x : \tau$ to the nodes in A.

B.2 *Typing system*

The linear/affine typing rules are given as follows (the secrecy typing is defined by adding the condition in $\cdots$ and replacing $\vdash$ by $\vdash_{\mathsf{sec}}$).

$$
(\mathsf{Zero})\ \frac{-}{\vdash \mathbf{0} \triangleright \_}
\qquad
(\mathsf{Par})\ \frac{\vdash P_i \triangleright A_i \quad (i=1,2) \qquad A_1 \asymp A_2}{\vdash P_1|P_2 \triangleright A_1 \odot A_2}
\qquad
(\mathsf{Res})\ \frac{\vdash P \triangleright A^{x:\tau} \qquad \mathsf{md}(\tau) \in \mathcal{M}_! \cup \{\updownarrow\}}{\vdash (\boldsymbol{\nu}\, x)P \triangleright A/x}
\qquad
(\mathsf{Weak})\ \frac{\vdash P \triangleright A^{-x} \qquad \mathsf{md}(\tau) \in \mathcal{M}_? \cup \{\updownarrow\}}{\vdash P \triangleright A,\ x : \tau}
$$

$$\text{(In}^{\downarrow_{\mathrm{L}}}\text{)}\quad \frac{\vdash P \triangleright \vec{y}:\vec{\tau},\ \uparrow_{\mathrm{L}}A^{-x}, \uparrow_{\mathrm{A}}\mathbf{?}B^{-x}}{\vdash x(\vec{y}).P \triangleright (x:(\vec{\tau})^{\downarrow_{\mathrm{L}}}\rightarrow A),\ B} \qquad \text{(In}^{!_{\mathrm{L}}}\text{)}\quad \frac{\vdash P \triangleright \vec{y}:\vec{\tau},\ \mathbf{?}_{\mathrm{L}}A^{-x}, \mathbf{?}_{\mathrm{A}}B^{-x}}{\vdash !\,x(\vec{y}).P \triangleright (x:(\vec{\tau})^{!_{\mathrm{L}}}\rightarrow A), B}$$

$$\text{(In}^{\downarrow_{\mathrm{A}}}\text{)}\ \ \boxed{s \sqsubseteq \mathsf{tamp}(A)}\quad \frac{\vdash P \triangleright \vec{y}:\vec{\tau},\ \uparrow_{\mathrm{A}}\mathbf{?}A^{-x}}{\vdash x(\vec{y}).P \triangleright x:(\vec{\tau})_s^{\downarrow_{\mathrm{A}}},\ A} \qquad \text{(In}^{!_{\mathrm{A}}}\text{)}\quad \frac{\vdash P \triangleright \vec{y}:\vec{\tau},\ \mathbf{?}A^{-x}}{\vdash !\,x(\vec{y}).P \triangleright x:(\vec{\tau})^{!_{\mathrm{A}}},\ A} \qquad \text{(Out)}\ \ p \in \mathcal{M}_{?} \cup \mathcal{M}_{\uparrow}\quad \frac{\vdash P \triangleright C^{\vec{y}:\vec{\tau}} \quad C \asymp x:(\vec{\tau})_s^p}{\vdash \overline{x}(\vec{y})P \triangleright C/\vec{y} \odot x:(\vec{\tau})_s^p}$$

We give a brief illustration of typing rules.

- (Zero) starts from the empty action type.
- (Par) uses $\asymp$ for controlling composition. For example, if P has type $x:()^{\uparrow_{\mathrm{L}}}$ and Q has type $x:()^{\uparrow_{\mathrm{L}}}$, then $P \mid Q$ is not typable because $()^{\uparrow_{\mathrm{L}}} \not\asymp ()^{\uparrow_{\mathrm{L}}}$.
- (Res) allows hiding of a name only when its mode is $\updownarrow$ or replicated (so that channels of modes $\uparrow$, $\downarrow$ or $\mathbf{?}$ should be compensated by their duals before restricted). (Weak) weakens $\updownarrow$ and $\mathbf{?}$-nodes since we allow the possibility of having no action at these channels. Formally the weakening of these nodes is necessary for having subject reduction.
- (In$^{\downarrow_{\mathrm{L}}}$) records the causality from linear input type $x:(\vec{\tau})^{\downarrow_{\mathrm{L}}}$ to linear output types. The side condition A^{-x} and B^{-x} ensure linearity (i.e. unique occurrence) of x. (In$^{!_{\mathrm{L}}}$) records the causality from replicated input type to $\mathbf{?}_{\mathrm{L}}$-types. The side condition A^{-x} is required to ensure acyclicity. Note (In$^{!_{\mathrm{L}}}$) and (In$^{!_{\mathrm{A}}}$) never suppress $\uparrow_{\mathrm{L}}$ or $\uparrow_{\mathrm{A}}$ action (except for that which is abstracted) otherwise unicity of affine or linear name would be lost.
- In (In$^{\downarrow_{\mathrm{A}}}$), $\downarrow_{\mathrm{A}}$ never suppresses $\uparrow_{\mathrm{L}}$, which is crucial for integration (suppose that x is affine while y is linear in $x.\overline{y}$: then a message at x may never arrive so that y may not fire, violating linearity (Yoshida, 2002)). In securecy typing, we care the secrecy level since affine input directly receives information (dually to an affine output which directly emits information). If the information is received at s, its effects can only be shown to the outside at s or above.
- (Out$^{\mathbf{?}_{\mathrm{L}}}$) essentially composes the output prefix and the body in parallel.

See (Honda & Yoshida, 2002) for more examples and explanations.

C Typing rules for branching/selection

The typing rules for branching/selection are as follows (the secrecy typing is defined by adding the condition in $\boxed{\cdots}$ and replacing $\vdash$ by $\vdash_{\mathsf{sec}}$).

$$\text{(Bra}^{\downarrow_{\mathrm{L}}}\text{)}\ \ \boxed{s \sqsubseteq \mathsf{tamp}(A,\ B)}\quad \frac{\vdash P_i \triangleright \vec{y}_i:\vec{\tau}_i,\ \uparrow_{\mathrm{L}}A^{-x},\ \uparrow_{\mathrm{A}}\mathbf{?}B^{-x}}{\vdash x[\&_i(\vec{y}_i).P_i] \triangleright (x:[\&_i\vec{\tau}_i]_s^{\downarrow_{\mathrm{L}}}\rightarrow A),\ B} \qquad \text{(Sel}^{\uparrow_{\mathrm{L}}}\text{)}\quad \frac{\vdash P \triangleright A^{\vec{y}:\vec{\tau}_i} \quad x \notin \mathsf{fn}(A)}{\vdash \overline{x}\mathtt{in}_i(\vec{y})P \triangleright A/\vec{y}, x:[\oplus\vec{\tau}_i]_s^{\uparrow_{\mathrm{L}}}}$$

In (Bra$^{\downarrow_{\mathrm{L}}}$), each summand should have an identical action type A (except for abstracted channels $\vec{y}_i : \vec{\tau}_i$). This is similar to the sum type in the λ-calculus and additives in Linear Logic. A linear branching receives information by being invoked at one of its branches. Hence the effect should not be transmitted at levels which are the same as or above the receiving level as (In$^{\downarrow_{\mathrm{A}}}$). The selection can be understood as (Out$^{\uparrow_{\mathrm{A}}}$).

D Typing rules for stateful actions

The typing rules for stateful actions are as follows (the secrecy typing is defined by adding the condition in $\boxed{\cdots}$ and replacing $\vdash$ by $\vdash_{\mathsf{sec}}$, as well as imposing structural security).

$$(\mathsf{In}^{!_{\mathrm{L}}})\ \frac{\boxed{s \sqsubseteq \mathsf{tamp}(A,B)} \quad \vdash P \triangleright \vec{y}:\vec{\tau}, ?_{\mathrm{L}}A^{-x}, ?_{\mathrm{A}}B^{-x}}{\vdash !\,x(\vec{y}).P \triangleright (x:(\vec{\tau})^{!_{\mathrm{L}}}_{s} \rightarrow A), B} \qquad (\mathsf{In}^{!_{\mathrm{A}}})\ \frac{\boxed{s \sqsubseteq \mathsf{tamp}(A)} \quad \vdash P \triangleright \vec{y}:\vec{\tau},\ ?A^{-x}}{\vdash\ !x(\vec{y}).P \triangleright x:(\vec{\tau})^{!_{\mathrm{A}}}_{s},\ A}$$

$$(\mathsf{Ref}) \qquad \vdash \mathsf{Ref}\langle xy\rangle \triangleright x:\mathsf{ref}_s\langle\tau\rangle,\ y:\overline{\tau}$$

$$(\mathsf{Read}) \qquad \vdash \overline{x}\mathtt{read}\langle c\rangle \triangleright x:\mathsf{rw}_s\langle\tau\rangle, c:(\overline{\tau})^{\uparrow_{\mathrm{L}}}$$

$$(\mathsf{Write}) \qquad \vdash \overline{x}\mathtt{write}\langle vc\rangle \triangleright x:\mathsf{rw}_s\langle\tau\rangle, c:()^{\uparrow_{\mathrm{L}}}, v:\tau$$

The typing system for the elementary state is defined by the above last three rules together with (Zero,Par,Res,Weak,In,Out) from Appendix B. We restrict A and B to the immutable affine clients types in $(\mathsf{In}^{!_{\mathrm{L}}})$ and $(\mathsf{In}^{!_{\mathrm{A}}})$.

For the general reference, we use all five rules above. In (Ref), we note $\mathsf{md}(\tau) \in \mathcal{M}_!$ by the well-formedness. (Read) first inquires at a reference, then receives a value typed by τ; (Write) writes to a reference then gets a linear acknowledgement at c. These rules can be understood in the light of the reduction rules given in the main section.

E Proof for Lemma 5.12

In this section, we show the key reasoning steps for establishing Lemma 5.12. While a mechanical inductive proof of this lemma is possible (based on induction of the length of derivation of possibly insecurely typed terms), we present an operational proof which may convey the central reason why the lemma in fact holds. First of all, we can check easily that, because of the shape of each flow inference rule which leads to co-cancellable types in the source, P and R_i as given in the lemma only depend on interactions at either unary linear channels or associated replicated channels for reaching the desired observable. This allows us to analyse the chain of typed transition relations created by interaction between P and R_i, which we call *linear call sequences* following Yoshida (2002) and Yosdhida *et al.* (2002).

For defining a linear call-sequence, we use labelled transition relations. Actions $l, l', \ldots$ are given by the grammar:

$$l \ ::= \ \boldsymbol{\tau} \ \mid \ x(\vec{y}) \ \mid \ \overline{x}(\vec{y}).$$

$t, s, r, \ldots$ range over sequences of labels. $\mathsf{bn}(l)$ denotes bound names in l. If $l \neq \boldsymbol{\tau}$, we write $\mathsf{sbj}(l)$ for the initial free name of l. Given A, we often write l^{τ} if $A(\mathsf{sbj}(l)) = \tau$ and define $\mathsf{md}(l^{\tau}) = \mathsf{md}(\tau)$. We often write l^{p} if $\mathsf{md}(l^{\tau}) = p$. We define $\overline{x(\vec{y})} = \overline{x}(\vec{y})$ and $\overline{\overline{x}(\vec{y})} = x(\vec{y})$ ($\overline{\boldsymbol{\tau}}$ undefined). Using these labels, the typed transition $P^{A} \xrightarrow{l} Q^{B}$ is formally defined in Appendix F.

Set $\curvearrowright = \curvearrowright_{\mathsf{b}} \cup \curvearrowright_{\mathsf{p}}$ where $l \curvearrowright_{\mathsf{b}} l'$ means that the subject of l' is bound by l, and $l \curvearrowright_{\mathsf{p}} l'$ means that l' is under input prefix l. We write $\langle l,\ \overline{l}\rangle \curvearrowright l_2$ if $P \overset{l_2}{\Longrightarrow} Q$ and P has subterms Q_1 and Q_2 such that $Q_1 \xrightarrow{l} Q_1'$ and $Q_2 \xrightarrow{\overline{l}\cdot l_2} Q_2'$ with $\overline{l} \curvearrowright l_2$; similarly we define

$l_1 \curvearrowright \langle l,\, \overline{l} \rangle$. When writing down a sequence of causal chains, we often denote $\langle l,\, \overline{l} \rangle$ by $\boldsymbol{\tau}$, and extend the above chain $l_1 \curvearrowright \boldsymbol{\tau}^* \curvearrowright l_2$ and denote it $l_1 \curvearrowright^+ l_2$.

We can now define a linear chain of co-cancellable actions.

Definition E.1 (linear call-sequence) A *linear call-sequence (l.c.s.) to l under A* is a (possibly infinite) sequence of actions with co-cancellable types whose visible actions have the following shape:

$$(l_0^{\downarrow_{\mathrm{L}}} \curvearrowright^+)\ l_1^{?_{\mathrm{L}}} \curvearrowright_{\mathsf{b}} l_2^{\downarrow_{\mathrm{L}}} \curvearrowright^+ l_3^{?_{\mathrm{L}}} \curvearrowright_{\mathsf{b}} l_4^{\downarrow_{\mathrm{L}}} \curvearrowright^+ \cdots l_{2n-1}^{?_{\mathrm{L}}} \curvearrowright_{\mathsf{b}} l_{2n}^{\downarrow_{\mathrm{L}}} \cdots$$

where $\tau_i, \tau_{ij}, \tau_{i+1}$ are all co-cancellable types in $l_i^{\tau_i} \curvearrowright^* (l_{ij}^{\tau_{ij}}, \overline{l}_{ij}) \curvearrowright^* l_{i+1}^{\tau_{i+1}}$ in the above sequence and if it is finite, we have $l_{2n}^{\downarrow_{\mathrm{L}}} \curvearrowright_{\mathsf{p}} l$ for some $n \lneq \omega$.

The basic lemmas for the linear call sequence follow.

Lemma E.2 1. (permutation) $P \xrightarrow{l}\xrightarrow{l'} P'$ *with* $l \not\curvearrowright l'$ *implies* $P \xrightarrow{l'}\xrightarrow{l} P'$.

2. *Suppose* $\vdash P \blacktriangleright \Gamma \rightsquigarrow x : (\vec{\tau})^{\uparrow_{\mathrm{L}}}$. *Then there is a finite l.c.s.* $l_1 \cdot l_2 \cdots l_n$ *to* l *such that* $\mathsf{sbj}(l) = x$ *and* $P \xrightarrow{l_1 \cdots l_n \cdot l}$.
3. (l.c.s. from $!_{\mathrm{L}}$) *Suppose* $\vdash P \blacktriangleright \Gamma \rightsquigarrow x : (\vec{\tau}\tau)^{!_{\mathrm{L}}}$ *with* $(\vec{\tau}\tau)^{!_{\mathrm{L}}}$ *cancellable. Then* $P \xrightarrow{x(y_1 \ldots y_n z)} P'$ *implies there is a finite l.c.s.* $l_1 \cdot l_2 \cdots l_n$ *to* l *s.t.* $\mathsf{sbj}(l) = z$ *and* $P' \xrightarrow{l_1 \cdots l_n \cdot l}$.

Proof
(1) is obvious.

(2) is proved essentially by the same reasoning as in Lemma 7.1 in Yoshida *et al.* (2002) and Lemma 2(2) in Yoshida (2002). Suppose $\vdash P \blacktriangleright \Gamma \rightsquigarrow x : (\vec{\tau})^{\uparrow_{\mathrm{L}}}$. Then we know $\vdash P \triangleright A$ implies $x : (\vec{\tau})^{\uparrow_{\mathrm{L}}} \in |A|$ by definition. Then we prove the following more general claim:

Claim: $P \xrightarrow{t \cdot l}$ with l linear output iff there is a shortest l.c.s. $l_1 \curvearrowright \cdots \curvearrowright l_n$ to l such that $P \xrightarrow{l_1 \cdots l_n}\xrightarrow{l}$.

Proof of the Claim: The $\Leftarrow$-direction is obvious by letting $t = l_1 \cdots l_n$. For the $\Rightarrow$-direction, by (1), we know there is a finite sequence such that $l_1 \curvearrowright l_2 \cdots l_{n-1} \curvearrowright l_n \curvearrowright l$. Then exactly one of the following must be true: **(a)** $\mathsf{md}(l_n) = \downarrow_{\mathrm{L}}$. **(b)** $l_n = (l_{n0}^{\tau_{n0}}, \overline{l_{n0}})$ with $\mathsf{md}(\tau_{n0}) = \{\downarrow_{\mathrm{L}}, \uparrow_{\mathrm{L}}\}$. This is because $\mathsf{md}(l_n) = \downarrow_{\mathrm{A}}$ is impossible by $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$ (affine inputs never suppress linear outputs), and $\mathsf{md}(l_n) \in \mathcal{M}_!$ is impossible by $(\mathsf{In}^{!_{\mathrm{L}}})$,$(\mathsf{In}^{!_{\mathrm{A}}})$ (replicated inputs never suppress free outputs). Also by IO-alternation, $\mathsf{md}(l_n) \in \mathcal{M}_{?,\uparrow}$ is impossible.

Suppose (a) holds. Then we have either (1) $l_n = l_1$ or (2) $l_{n-1}^{\tau_{n-1}} \curvearrowright_{\mathsf{b}} l_n$ with $\mathsf{md}(l_{n-1}) = ?_{\mathrm{L}}$ since by the definition of the syntax of types, a linear output cannot directly carry linear inputs; also $\mathsf{md}(l_{n-1}) \neq ?_{\mathrm{A}}$ since the affine client outputs cannnot carry a linear output. Note that since l_{n-1} binds l_n, we can set $\tau_{n-1} = (\vec{\tau}_{n-1}\tau_n)^{?_{\mathrm{L}}}$ with $\mathsf{md}(\tau_n) = \downarrow_{\mathrm{L}}$. Now suppose $l_{n-2} \curvearrowright l_{n-1}$. Then $\mathsf{md}(l_{n-2}) = \downarrow_{\mathrm{L}}$ since it is impossible that $\mathsf{md}(l_{n-2}) = !_{\mathrm{L}}$ and $l_{n-2} \curvearrowright l_{n-1} \curvearrowright l_n \curvearrowright l$ by $(\mathsf{In}^{!_{\mathrm{L}}})$ (replication never suppresses free linear outputs, i.e. l). We repeat the same routine by setting $n = n - 2$.

Now suppose (b) holds. Then if $l_{n-1} \curvearrowright_{\mathsf{b}} l_{n0}$, then we just repeat the same routine as above by setting $n = n-1$ in the above case (since $\mathsf{md}(l_{n-1}) = ?_{\mathrm{L}}$). Suppose $l_{n-1} \curvearrowright \overline{l_{n0}}$. By input typing rules, $l_{n-1} \curvearrowright_{\mathsf{b}} l_n$. Then obviously $l_{n-1} = \boldsymbol{\tau} = \langle l'_{n-1}, \overline{l_{n-1}}' \rangle \curvearrowright_{\mathsf{b}} l_n$ with $l'_{n-1} \curvearrowright_{\mathsf{p}} l_n$. Then $\mathsf{md}(l'_{n-1}) = \downarrow_{\mathrm{L}}$ as in the previous reasoning. Hence we repeat the the same routine as (a) by setting $n = n-1$. Then we repeat this procedure until we reach $n = 1$. (end of the proof of the claim) $\square$

Now we only have to prove finiteness of the linear call-sequence to l. Assume by contradiction we have an infinite $l_1 \cdot l_2 \cdots$ to l in $\vdash R \triangleright C, E \to x : \tau$ with $\mathsf{md}(\tau) = \uparrow_{\mathrm{L}}$ and $\mathsf{sbj}(l) = x$. Then there always exist Q and C' such that $\vdash R \mid Q \triangleright C', x : \tau$ with $\mathsf{md}(C') \subset \mathcal{M}_! \cup \{\updownarrow\}$. If a linear call sequence in R is infinite, then $\neg R \mid Q \Downarrow_x$, which contradicts the linear liveness in Theorem 1 in (Yoshida, 2002), hence done.

(3) By assumption, we know $\vdash P \triangleright A$ with $x : (\vec{\tau}\tau)^{!_{\mathrm{L}}} \in |A|$ and $\mathsf{md}(\tau) = \uparrow_{\mathrm{L}}$. Then it is straightforward by (2) because $\vdash P' \triangleright A, \vec{y} : \vec{\tau}, z : \tau$ with $\mathsf{md}(\tau) = \uparrow_{\mathrm{L}}$. $\square$

Lemma E.3 *Let* $\vdash P^{A, w:()_s^{\uparrow_{\mathrm{A}}}} \blacktriangleright \Gamma \rightsquigarrow w : ()_s^{\uparrow_{\mathrm{A}}}$ *such that types in* A *are negative and* Γ *is co-cancellable. Assume* $\vdash R \triangleright \overline{A}$. *Then* (a) *there are finite linear call sequences* t *(note* t *is a sequence of* $\boldsymbol{\tau}$*-actions) such that* $P \mid R \xrightarrow{t} (\boldsymbol{\nu}\, \vec{a})(P' \mid R')$ *with* $P \overset{s}{\Longrightarrow} P'$ *and* $R \overset{\overline{s}}{\Longrightarrow} R'$*; and* (b) $P' \Downarrow_w$ *if* $(P \mid R) \Downarrow_w$*; else* $\neg P' \Downarrow_w$.

Proof
The proof is essentially the same as in Lemma 8 in Yoshida *et al.* (2002) (but this case is simpler since we do not have to prove the closure property). Assume $b : \tau \in |\Gamma|$. If $\mathsf{md}(\tau) = \downarrow_{\mathrm{L}}$, then $b : \overline{\tau} \in |\overline{A}|$, thus we apply (2) in the above lemma; if $\mathsf{md}(\tau) = ?_{\mathrm{L}}$, then we apply (3) in the above lemma and the inductive hypothesis repeatedly. $\square$

Now we prove Lemma 5.12. Assume $(P \mid R_1) \Downarrow_w$. Then by the above lemma, there are finite linear call sequences t and s such that $P | R_1 \xrightarrow{t} (\boldsymbol{\nu}\, \vec{a})(P' | R_1')$ with $P \overset{s}{\Longrightarrow} P'$ and $P' \Downarrow_w$. Then by permutation, there exists a l.c.s s_1 such that $P \overset{s_1}{\Longrightarrow} P_1 \overset{s_1'}{\Longrightarrow} P'$. On the other hand, by the above lemma again, we also know there are finite linear call sequences t' such that $P | R_2 \xrightarrow{t'} (\boldsymbol{\nu}\, \vec{a}')(P'' | R_2')$. Again by the permutation, there exists a l.c.s. s_2 such that $P \overset{s_2}{\Longrightarrow} P_2 \overset{s_2'}{\Longrightarrow} P''$. By the unicity of $s_{1,2}$, we note $P_1 \equiv P_2$. Because the affine output cannot be suppressed by the linear replication, we know $s_1 \curvearrowright^* \overline{w}$, but $s_1' \not\curvearrowright \overline{w}$. Hence $P' \Downarrow_w$ implies $P_1 \Downarrow_w$ (hence $P_2 \Downarrow_w$). Then finally an application of the permutation lemma gives $P'' \Downarrow_w$, as desired.

F Typed transition relation

The labels $l, l', \ldots$ are given by the grammar:

$$l \ ::= \ \boldsymbol{\tau} \ \mid \ x(\vec{y}) \ \mid \ \overline{x}(\vec{y}) \ \mid \ x\mathtt{in}_i(\vec{y}) \ \mid \ \overline{x}\mathtt{in}_i(\vec{y})$$

Using these labels, the typed transition $P^A \xrightarrow{l} Q^B$ is generated from the following rules. We assume all l.h.s. processes are well-typed. We define the relation A *allows*

l as the negation of: (1) $A(\mathsf{sbj}(l)) = \updownarrow$ or (2) l is output and $\mathsf{md}(A(\mathsf{sbj}(l))) \in \mathcal{M}_!$ (Berger *et al.*, 2001). $\mathsf{n}(l)$ is the set of names in l.

$$\begin{array}{rcll}
x(\vec{y}).P^{A} & \xrightarrow{x(\vec{y})} & P^{\vec{y}:\vec{\tau},A/x} & (x:(\vec{\tau})^p \in A,\ p \in \mathcal{M}_{\downarrow}) \\
!x(\vec{y}).P^{A} & \xrightarrow{x(\vec{y})} & !x(\vec{y}).P|P^{\vec{y}:\vec{\tau},A} & (x:(\vec{\tau})^p \in A,\ p \in \mathcal{M}_{!}) \\
\overline{x}(\vec{y})P^{A} & \xrightarrow{\overline{x}(\vec{y})} & P^{\vec{y}:\vec{\tau},A/x} & (x:(\vec{\tau})^p \in A, p \in \mathcal{M}_{\uparrow}) \\
\overline{x}(\vec{y})P^{A} & \xrightarrow{\overline{x}(\vec{y})} & P^{\vec{y}:\vec{\tau},A} & (x:(\vec{\tau})^p \in A, p \in \mathcal{M}_{?}) \\
x[\&_i(\vec{y}_i).P_i]^{A} & \xrightarrow{x\mathtt{in}_i(\vec{y}_i)} & P_i^{\vec{y}_i:\vec{\tau}_i,A/x} & (x:[\&\vec{\tau}_i]^p \in A,\ p \in \mathcal{M}_{\downarrow}) \\
\overline{x}\mathtt{in}_i(\vec{y})P^{A} & \xrightarrow{\overline{x}\mathtt{in}_i(\vec{y})} & P^{\vec{y}:\vec{\tau}_i,A/x} & (x:[\oplus_i\vec{\tau}_i]^p \in A,\ p \in \mathcal{M}_{\uparrow}) \\
\overline{x}\mathtt{in}_i(\vec{y})P^{A} & \xrightarrow{\overline{x}\mathtt{in}_i(\vec{y})} & P^{\vec{y}:\vec{\tau}_i,A} & (x:[\oplus_i\vec{\tau}_i]^p \in A,\ p \in \mathcal{M}_{?})
\end{array}$$

$$\frac{P_1' \equiv_\alpha P_1 \ \ P_1^{A_1} \xrightarrow{l} P_2^{A_2} \ \ P_2 \equiv_\alpha P_2'}{P_1'^{A_1} \xrightarrow{l} P_2'^{A_2}} \qquad \frac{P_1^{A_1} \xrightarrow{l} P_2^{A_2} \quad x \notin \mathsf{n}(l)}{(\boldsymbol{\nu}\, x)P_1^{A_1/x} \xrightarrow{l} (\boldsymbol{\nu}\, x)P_2^{A_2/x}}$$

$$\frac{P_1^{A_1} \xrightarrow{l} P_2^{A_2} \quad A_1 \odot B \text{ allows } l}{P_1|Q^{A_1\odot B} \xrightarrow{l} P_2|Q^{A_2\odot B}} \qquad \frac{P_1^{A_1} \xrightarrow{l} P_2^{A_2} \quad Q_1^{B_1} \xrightarrow{\overline{l}} Q_2^{B_2}}{P_1|Q_1^{A_1\odot B_1} \xrightarrow{\tau} (\boldsymbol{\nu}\, \mathsf{bn}(l))(P_2|Q_2)^{A_2\odot B_2/\mathsf{bn}(l)}}$$

The transition relation for the reference agents are defined as follows.

$$\begin{array}{rcl}
\mathsf{Ref}\langle xv\rangle & \xrightarrow{x\mathtt{inl}_1\langle c\rangle} & \mathsf{Ref}\langle xv\rangle \mid \overline{c}\langle v\rangle \\
\mathsf{Ref}\langle xv\rangle & \xrightarrow{x\mathtt{inl}_2\langle wc\rangle} & \mathsf{Ref}\langle xw\rangle \mid \overline{c}
\end{array}$$

Acknowledgements

The authors thank anonymous referees and Martin Berger for their useful suggestions. Kohei Honda is partially supported by EPSRC grant GR/S55545. Nobuko Yoshida is partially supported by EPSRC grants GR/R33465 and GR/S55538.

References

Abramsky, S., Jagadeesan, R. and Malacaria, P. (2000) Full abstraction for PCF, 1994. *Info. & Comp.* **163**, 409–470.

Abadi, M., Banerjee, A., Heintze, N. and Riecke, J. (1999) A core calculus of dependency. *Proc. POPL'99*. ACM.

Agat, J. (2000) Transforming out timing leaks. *Proc. POPL'00*. ACM Press.

Asperti, A., Danos, V., Laneve, C. and Regnier, L. (1994) Paths in lambda-calculus. *LICS'94*. IEEE.

Berger, M., Honda, K. and Yoshida, N. (2001) Sequentiality and the π-calculus. *TLCA01: Lecture Notes in Computer Science 2044*, pp. 29–45. Springer-Verlag. (A full version available at www.dcs.qmw.ac.uk/˜kohei.)

Berger, M., Honda, K. and Yoshida, N. (2003) Genericity and the π-calculus. *FoSSaCs03: Lecture Notes in Computer Science 2620*, pp. 103–119. Springer-Verlag. (A full version available at: www.dcs.qmul.ac.uk/˜martinb.)

Bodei, C., Degano, P., Nielson, F. and Nielson, H. R. (1999), Static analysis of processes for no read-up and no-write-down. *Proc. FoSSaCS'99: Lecture Notes in Computer Science 1578*, pp. 120–134. Springer-Verlag.

Bodei, C., Degano, P., Nielson, F. and Nielson, H. R. (1998) Control flow analysis for the pi-calculus. *Proc. CONCUR 98: Lecture Notes in Computer Science*, pp. 84–98. Springer-Verlag.

Denning, D. and Denning, P. (1997) Certification of programs for secure information flow. *Comm. ACM*, **20**, 504–513.

Focardi, R., Gorrieri, R. and Martinelli, F. (2000) Non-interference for the analysis of cryptographic protocols. *ICALP00: Lecture Notes in Computer Science 1853*. Springer-Verlag.

Girard, J.-Y. (1987) Linear logic. *Theor. Comput. Sci.* **50**, 1–102.

Goguen, J. and Meseguer, J. (1982) Security policies and security models. *IEEE Symposium on Security and Privacy*, pp. 11–20. IEEE.

Heintze, N. and Riecke, J. (1998) The SLam calculus: programming with secrecy and integrity. *Proc. POPL'98*, pp. 365–377. ACM.

Hennessy, M. (2003) The security picalculus and non-interference, *Proc. MFPS XIX*, Montreal, Canada.

Hennessy, M. and Riely, J. (2000) Information flow vs resource access in the asynchronous pi-calculus. *ICALP00: Lecture Notes in Computer Science 1853*, pp. 415-427. Springer-Verlag.

Honda, K. (1996) Composing processes. *Proc. POPL'96*, pp. 344–357. ACM.

Honda, K., Vasconcelos, V. and Yoshida, N. (2000) Secure information flow as typed process behaviour. *ESOP'00: Lecture Notes in Computer Science 1782*, pp. 180–199.

Honda, K. and Tokoro, M. (1991) An object calculus for asynchronous communication. *ECOOP'91: Lecture Notes in Computer Science 512*, pp. 133–147.

Honda, K. and Yoshida, N. (1999) Game-theoretic analysis of call-by-value computation. *Theor. Comput. Sci.* **221**, 393–456.

Honda, K. and Yoshida, N. (2002) A uniform type structure for secure information flow. *Proc. POPL'02*, pp. 81–92. ACM Press. (The full version: DOC technical report 2002/13, Department of Computing, Imperial College London. Revised in April 2004. 91 pages. Available at: www.dcs.qmul.ac.uk/~kohei.)

Honda, K., Yoshida, N. and Berger, M. (2004) Controls and the π-calculus. *Proc. CW'04*. ACM Press.

Hyland, M. and Ong, L. (2000), Full abstraction for PCF: I, II and III. *Info. & Comp.* **163**, 285–408.

Kobayashi, N., Pierce, B. and Turner, D. (1999) Linear types and the π-calculus. *TOPLAS*, **21**(5), 914–947.

Laurent, O. (2002a) Polarized proof-nets and $\lambda\mu$-calculus. *Theor. Comput. Sci.* **290**(1), 161–188.

Laurent, O. (2002b) Polarized games. *LICS'02*, pp. 265–274. IEEE.

Laurent, O. and Regnier, L. (2003) About translations of classical logic into polarized linear logic. *LICS*, pp. 11–20. IEEE.

Mantel, H. and Sabelfeld, A. (2003) A unifying approach to the security of distributed and multi-threaded programs. *J. Comput. Security*, **11**(4), 615–676.

Milner, R. (1992a) Functions as processes. *MSCS*, **2**(2), 119–141.

Milner, R. (1992b) Polyadic π-Calculus: a tutorial. *Proc. International Summer School on Logic Algebra of Specification*. Marktoberdorf.

Milner, R., Parrow, J. G. and Walker, D. J. (1992) A calculus of mobile processes. *Infor. & Comput.* **100**(1), 1–77.

Nielson, F., Nielson, H. R. and Hankin, C. (1999) *Principles of Program Analysis*. Springer-Verlag.

Pottier, F. and Simonet, V. (2002) Information flow inference for ML. *Proc. POPL'02*, pp. 319–330. ACM Press.

Pierce, B and Sangiorgi, D. (1996) Typing and subtyping for mobile processes. *MSCS*, **6**(5), 409–453.

Pottier, F. (2002) A simple vie of type-secure information flow in the π-calculus. *CSFW02*, pp. 320–330. IEEE.

Sabelfeld, A. and Sands, D. (1999) A per model of secure information flow in sequential programs. *ESOP'99: Lecture Notes in Computer Science 1576*. Springer-Verlag.

Sangiorgi, D. (1996) π-calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.* **167**(2), 235–271.

Smith, G. and Volpano, D. (1998) Secure information flow in a multi-threaded imperative language. *Proc. POPL'98*, pp. 355–364. ACM.

Smith, G. (2001) A new type system for secure information flow. *Proc. CSFW'01*, IEEE.

Vasconcelos, V. and Honda, K. (1993) Principal typing schemes in a polyadic π-calculus. *CONCUR'93: Lecture Notes in Computer Science 715*. Springer-Verlag.

Volpano, D., Smith, G. and Irvine, C. (1996) A sound type system for secure flow analysis. *J. Comput. Security*, **4**(2–3), 167–187.

Walker, D. (1995) Objects in the Pi-calculus. *Info. &. Comp.* **116**(2), 253–271.

Winskel, G. (1993) *The Formal Semantics of Programming Languages*. MIT Press.

Yoshida, N. (1996) Graph types for mobile processes. *FST/TCS'16: Lecture Notes in Computer Science 1180*, pp. 371–386. Springer-Verlag. (Full version as LFCS Technical Report, ECS-LFCS-96-350, 1996.)

Yoshida, N. (2002) *Type-Based Liveness Guarantee in the Presence of Nontermination and Nondeterminism, April 2002*. MCS Technical Report, 2002–20, University of Leicester. (Available at www.doc.ic.ac.uk/~yoshida.)

Yoshida, N., Honda, K. and Berger, M. (2002) Linearity and bisimulation. *Proc. 5th International Conference, Foundations of Software Science and Computer Structures (FoSSaCs 2002): Lecture Notes in Computer Science 2303*, pp. 417–433. Springer-Verlag. (A full version as a MCS technical report, 2001–48, University of Leicester, 2001. Available at www.doc.ic.ac.uk/~yoshida.)

Yoshida, N., Berger, M. and Honda, K. (2001) Strong normalisation in the π-calculus. *LICS'01*, pp. 311–322. IEEE. (Full version: *Infor. & Computation* **191** (2004), 145–202, Elevier.)

Zdancewic, S. and Myers, A. (2001) Secure information flow and CPS. *ESOP01: Lecture Notes in Computer Science 2028*, pp. 46–62, Springer-Verlag.