# A Uniform Type Structure for Secure Information Flow

KOHEI HONDA
Queen Mary, University of London
and
NOBUKO YOSHIDA
Imperial College of Science, Technology and Medicine, University of London.

The $\pi$-calculus is a process calculus in which we can compositionally represent dynamics of major programming constructs by decomposing them into a single communication primitive, the name passing. This work reports our experience in using a linear/affine typed $\pi$-calculus for the analysis and development of type-based analyses for programming languages, focussing on secure information flow analysis. After presenting a basic typed calculus for secrecy, we demonstrate its usage by a sound embedding of the dependency core calculus (DCC) and the development of the call-by-value version of DCC. The secrecy analysis is then extended to stateful computation, using which we develop a novel type discipline for imperative programming language which extends a secure multi-threaded imperative language by Smith and Volpano with general references and higher-order procedures. In each analysis, the embedding gives a simple proof of noninterference.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Information flow control*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type Structure*

General Terms: Languages, Security, Theory

Additional Key Words and Phrases: The $\pi$-Calculus, Typing System, Secure Information Flow, Type-Based Program Analysis

Contents



Author's address: K. Honda, Department of Computer Science, Queen Mary, University of London, Mile End, London E1 4NS, UK. N. Yoshida, Department of Computing, Imperial College of Science Technology and Medicine, University of London, Huxley Building, 180 Queen's Gate, London SW7 2BZ, UK.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 1999 ACM 0164-0925/99/0100-0111 $00.75

## 1. INTRODUCTION

**Motivation.** Large software is made up of many different components with different properties. Further it is a norm in modern distributed applications that a number of different programming constructs, or even different languages, are used in a single application. Types for programming offer a primary means to classify and control programs' behaviour with rigour and precision, with well-developed theories and an increasing number of applications. In particular, type structures often play a crucial role as a basis of diverse program analyses [47; 53; 7]. Can we use types for describing and reasoning about such an aggregation of diverse components, forming a basis for specifying, analysing and controlling their behaviour? For this to be effective, it should be possible to type-check one component with a specific type, say $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$ (where $\mathbb{N}$ is a type for a natural number and $\Rightarrow$ is a function type constructor), and combine it with other parts, which may have different type structures, with a guarantee that it behaves as decreed by the original type discipline. For example, if $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$ is inferred in a strongly normalising type discipline, and if we need to ensure this property, then we want the piece of code to behave as a total function producing a natural number. Note a program of this type needs a procedure given by its peer to perform its function: thus we cannot achieve our objective unless we have a consistent integration of multiple type disciplines.

A central technical difficulty in having such an integrated framework, even for basic type structures, comes from different nature of operations each typed formalism deals with. Assignment, function application, controls, method invocation, diverse forms of synchronisation, all have quite different dynamics: we can see this difference clearly when we write down their formal operational semantics and compare them. It is largely due to this difference why it is so hard to consistently merge individually coherent theories for isolated constructs, or to apply what was found in one realm to another realm. A well-known example is issues in transplanting polymorphism, initially developed for pure higher-order functions, to the universe of imperative programming idioms [13; 60; 39; 59; 63]. The different nature of dynamics of assignment commands from that of pure higher-order functions is the culprit of this difficulty. Given this variety, it looks hard to conceive any uniform framework of type structure for different language constructs: unless we have a tool, say syntax, which can represent them on a uniform basis.

**The $\pi$-Calculus.** The $\pi$-calculus [43; 42; 11; 26] is an extension of CCS [41] based on name passing. A basic form of its dynamics can be written down as the following reduction.

$$x(\vec{y}).P|\overline{x}\langle\vec{w}\rangle \longrightarrow P\{\vec{w}/\vec{y}\}$$

Here a vector of names $\vec{w}$ are communicated, via $x$, to an input process, resulting in name instantiation. Perhaps surprisingly, this single operation can compositionally represent dynamics of diverse language constructs, including function application, sequencing, assignment, exception, object, not to speak of communication and concurrency. We are thus prompted by the following question: can we have a foundational type structure for this calculus, similar to those for the $\lambda$-calculus, in which we can precisely capture diverse classes of computational behaviour uniformly? Unlike those for functions, types for interaction is an unexplored realm. More concretely, the preceding studies, cf. [42; 36; 51; 64], have shown that, even though operational encodings of diverse typed calculi into the $\pi$-calculus are possible, they rarely capture the original type structures fully. The issue is visible through, for example, the almost omnipresent lack of full abstraction in such encodings. At a deeper level, this means the encoded types guarantee only a weaker notion of behavioural properties than the original ones: the essential content of types is partially lost through the translation.

Gaining insights from the preceding studies on types for interaction including types for the $\pi$-calculus [42; 51; 22; 64; 36] and game semantics [4; 5; 34; 29], the present authors, with Martin Berger, recently reported [8; 66] that basic type structures for the $\pi$-calculus which precisely capture existing type structure do exist, allowing fully abstract translation of prominent functional typed calculi. In [8; 66], we have presented two type disciplines for the $\pi$-calculus which precisely characterise two classes of sequential higher-order functional behaviours, which we call *affine* and *linear*. These terms are used with the following meaning:

- *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, [8; 66] have shown PCF and strongly normalising $\lambda$-calculi are fully abstractly embeddable in the affine and linear $\pi$-calculus, respectively. In spite of faithfulness in embeddings, the form of types is quite different from that of function types, articulating a broader realm of computational behaviour. In particular, both call-by-value and call-by-name $\lambda$-calculi are embeddable into a single typing system by changing translation of types.

**Using the $\pi$-Calculus.** The uniform embeddability of typed languages in the $\pi$-calculus with linear/affine typing would suggest its potential usage as an integrated basis for multiple type disciplines. As a possible area for experimenting with such possibility, we consider so-called *type-based program analyses*, which are program analyses performed relying on type structures non-trivially. Here we can exploit the faithful representability of language constructs to organise the development of analyses into the following three steps (pictorially illustrated in Figure 1).

*Step 1: Embedding.* If there is a programming language with a faithful embedding

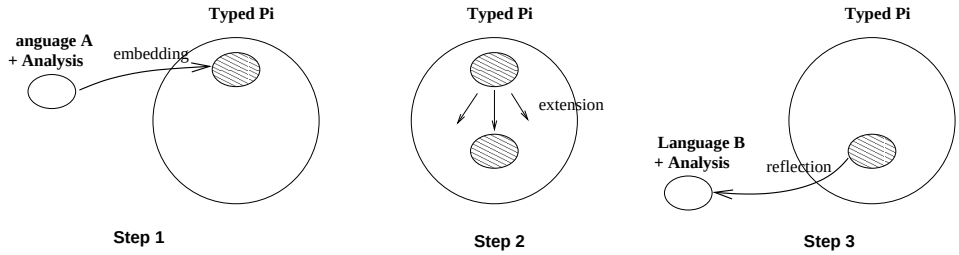


Fig. 1. Type-based Program Analysis using the $\pi$-Calculus

into the typed $\pi$-calculus, and there is a type-based analysis on that language, then we can transfer this analysis into the image of the embedding through syntactic translation.

*Step 2: Extension.* In the next step we extend this analysis, which is initially restricted to the image of the embedding, to the whole collection of typed processes. Basic syntactic and semantic properties of the analysis would be established, whose formulation can be assisted by those in the original analysis through the embedding.

*Step 3: Reflection.* After the completion of Step 2, we can now use the extended analysis for any language embeddable into the $\pi$-calculus. This is done by reflecting the analysis in the $\pi$-calculus onto another programming language through its embedding. The key safety properties of the new analysis are ensured through those of the analysis in the typed $\pi$-calculus.

Among others Step 2 can be challenging, since it is far from clear whether we can soundly extend the original analysis from the image of the embedding to the whole set of typed processes. When this challenge is met, however, the reward is the generalised analysis which not only retains the precision of the original analysis but which is repositioned in a broader realm of interacting processes, amenable to extension and integration.

**Secure Information Flow.** The present paper reports our experience of using the $\pi$-calculus for type-based program analyses, taking secure information flow analysis [3; 19; 49; 50; 56; 57; 58; 62] as an application domain. In this analysis, we use a typing system to ensure the safety of information flow in a given program, i.e. a high-level (secure) data never flows down to low-level (public) channels. Information flow analysis needs precise understanding of observable behaviour of program phrases and their interplay, because of the existence of covert channels [14]. In the $\pi$-calculus representation, computational dynamics is decomposed into interaction, where the notion of observables is made explicit. This makes the $\pi$-calculus a potentially effective tool for analysing subtle information flow among program phrases. Further, in many type-based information flow analyses, distinction between totality and partiality is crucial, both in functional [3] and imperative [62] settings, strongly suggesting its connection to linear/affine type structures. A uniform treatment of call-by-name and call-by-value pure functions as well as stateful computation in

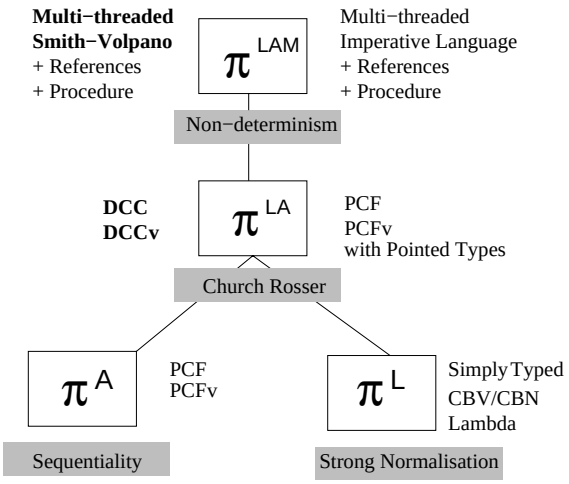


Fig. 2. A Family of Linear/Affine $\pi$-Calculi

secrecy is another motivation for using the $\pi$-calculus.

**Summary of Contributions.** The following summarises the main technical contributions of the present work.

- A typed $\pi$-calculus for secure information flow based on linear/affine type disciplines, which enjoys a basic noninterference property.
- The embeddability of the dependency core calculus (DCC) [3] in the secrecy-enhanced linear/affine $\pi$-calculus, and a simple operational proof of its noninterference property. We also present a novel call-by-value version of DCC.
- A new type system for secrecy in concurrent imperative programs with references and higher-order procedures. Its embeddability in the linear/affine $\pi$-calculus with state again gives a simple proof of non-interference.

A picture of typed calculi used in this paper is given in Figure 2. Each box represents a name of the typed $\pi$-calculus with a specific type structure ("L", "A" and "M" mean linear, affine and state, respectively). The right-hand side of the box shows systems we can embed in the basic typed $\pi$-calculus. The left-hand side shows secure languages we can embed in the secure version of the $\pi$-calculus. The grey box shows a basic property satisfied by the calculus.

**Outline.** This paper is a full version of [30], with complete definitions and detailed proofs. The emphasis is on illustrating, through concrete examples, how the typed $\pi$-calculus can be used for developing and justifying a non-trivial type-based analysis of programming languages. Along this spirit, the presentation of the call-by-value version of DCC and the extended version of Smith-Volpano language is considerably simplified from [30] by reformulation of encodings and a simplification of the linear/affine $\pi$-calculus with state. The present version also gives more com-

parisons with related work, including a formal conservativity result with respect to Smith's recent secure imperative language [57].

In the remainder, Section 2 introduces the $\pi$-calculus with a linear/affine type discipline, integrating type disciplines presented in [8; 66]. Section 3 studies a type-based secrecy analysis for the linear/affine $\pi$-calculus. Section 4 embeds DCC in the secure $\pi$-calculus and develops its call-by-version, both justified via the secure $\pi$-calculus. Section 5 presents a stateful extension of the linear/affine type discipline. Section 6 extends the secrecy analysis in Section 4 to stateful processes. Section 7 develops an extension of the Smith-Volpano's secure multi-threaded imperative calculus with general references and higher-order procedures, based on the secrecy analysis in Section 6. Section 8 concludes the paper with discussions on related work and further topics.

**Acknowledgements.** We thank Martin Berger for our ongoing collaboration on typed $\pi$-calculi and their applications. Stephen Zdancewic pointed out several mistakes in the conference version for which we are particularly grateful. Discussions with Geffeory Smith and Francois Pottier deepened our understanding of secrecy analysis. Discussions with Chris Hankin have broadened our perspective on the use of the $\pi$-calculus for program analyses, part of which is reflected in Introduction. The first author is partially supported by EPSRC grant GR/N/37633. The second author is partially supported by EPSRC grant GR/R33465/01.

## 2. THE $\pi$-CALCULUS WITH LINEAR/AFFINE TYPING

### 2.1 Processes

As a syntax, we use the asynchronous $\pi$-calculus [11; 26] enriched with branching constructs [21; 25; 27]. Let $x, y, \ldots$ and sometimes $a, b, \ldots$ range over a countable set of names (also called channels). We write $\vec{y}$ for a finite, possibly null, string of names. The set of untyped terms, which we often call *processes*, is given by the following grammar.

$$\begin{array}{rcllll} P & ::= & x(\vec{y}).P & \text{input} & \mid P \mid Q & \text{parallel} \\ & \mid & \overline{x}\langle\vec{y}\rangle & \text{output} & \mid (\boldsymbol{\nu}\, x)P & \text{hiding} \\ & \mid & x[\&_{i\in I}(\vec{y}_i).P_i] & \text{branching} & \mid \mathbf{0} & \text{inaction} \\ & \mid & \overline{x}\mathtt{in}_i\langle\vec{z}\rangle & \text{selection} & \mid\ !P & \text{replication} \end{array}$$

In $!P$ we require $P$ to be an input or a branching. We always assume names in $(\vec{v})$ (of input and branching) are pairwise distinct. This and $(\boldsymbol{\nu}\, x)$ act as binders. The sets of free/bound names, written $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$, as well as the alpha equality $\equiv_\alpha$, are defined in the standard way. We write $(\boldsymbol{\nu}\, y_1..y_n)P$ for $(\boldsymbol{\nu}\, y_1)..(\boldsymbol{\nu}\, y_n)P$. $\overline{x}(\vec{y})P$ stands for $(\boldsymbol{\nu}\, \vec{y})(\overline{x}\langle\vec{y}\rangle|P)$ (assuming names in $\vec{y}$ are pairwise distinct), which sends new names $\vec{y}$ local to $P$. We often omit empty parameters (e.g. $x.P$ stands for $x().P$). We often omit the indexing set $I$ (which should be either countable or finite) of $x[\&_{i\in I}(\vec{y}_i).P_i]$. We write $x[(\vec{y}_1).P_1\&(\vec{y}_2).P_2]$ for a binary branching and $\overline{x}\mathtt{inl}\langle\vec{v}\rangle/\overline{x}\mathtt{inr}\langle\vec{v}\rangle$ for binary selections. Branching is used for representing base values as well as conditionals [8; 66]. While this construct can be simply encoded into the calculus without it, the branching plays a basic rôle in the typed setting and is essential for the secrecy analysis we discuss later.

Processes are often considered modulo the *structural congruence* $\equiv$, which is the least congruence generated from the following rules.

- If $P \equiv_\alpha Q$ then $P \equiv Q$.
- $P|\mathbf{0} \equiv P$, $P|Q \equiv Q|P$ and $(P|Q)|R \equiv P|(Q|R)$.
- $(\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0}$, $(\boldsymbol{\nu}\, xy)P \equiv (\boldsymbol{\nu}\, yx)P$ and $(\boldsymbol{\nu}\, x)(P|Q) \equiv ((\boldsymbol{\nu}\, x)P)|Q$ when $x \notin \mathsf{fn}(Q)$.

### 2.2 Reduction

The dynamics of processes is defined by the reduction relation $\longrightarrow$, which is generated by the following rules. First, there are two rules for the unary name passing.

$$\begin{array}{rcl} x(\vec{y}).P \,|\, \overline{x}\langle\vec{v}\rangle & \longrightarrow & P\{\vec{v}/\vec{y}\} \\ !x(\vec{y}).P \,|\, \overline{x}\langle\vec{v}\rangle & \longrightarrow & !x(\vec{y}).P \,|\, P\{\vec{v}/\vec{y}\}. \end{array}$$

The reduction for branching involves selection of one branch, discarding the remaining ones, as well as name passing.

$$\begin{array}{rcl} x[\&_i(\vec{y}_i).P_i] \mid \overline{x}\mathtt{in}_j\langle\vec{v}\rangle & \longrightarrow & P_j\{\vec{v}/\vec{y}_j\} \\ !\, x[\&_i(\vec{y}_i).P_i] \mid \overline{x}\mathtt{in}_j\langle\vec{v}\rangle & \longrightarrow & !\, x[\&_i(\vec{y}_i).P_i] \mid P_j\{\vec{v}/\vec{y}_j\} \end{array}$$

Finally we close the reduction under parallel composition, hiding and $\equiv$.

$$\frac{P \longrightarrow P'}{P|R \longrightarrow P'|R} \qquad \frac{P \longrightarrow P'}{(\boldsymbol{\nu}\, x)P \longrightarrow (\boldsymbol{\nu}\, x)P'} \qquad \frac{P \equiv Q \quad Q \longrightarrow Q' \quad Q' \equiv P'}{P \longrightarrow P'}$$

For representing a sequence of zero or more reductions we use the multi-step reduction $\twoheadrightarrow \stackrel{\text{def}}{=} \equiv \cup \longrightarrow^*$. Simple examples of processes and their reduction follow.

**Example 2.1** (processes and reduction)

(1) A *natural number agent*, $[\![n]\!]_u \stackrel{\text{def}}{=} !u(c)\overline{c}\mathtt{in}_n$, acts as a server which necessarily returns a fixed answer, $n$. As an example, $[\![2]\!]_u|\overline{u}\langle e\rangle \longrightarrow [\![2]\!]_u|\overline{e}\mathtt{in}_2$.

(2) The process $\overline{u}(c)c[\&_{n\in\omega}\overline{e}\mathtt{in}_{n+1}]$ (with $\omega$ the set of natural numbers) behaves as the successor of the natural number agent above as follows (for illustration we follow the use of $\equiv$ in some detail).

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

The essence of this encoding lies in a precisely representation of the functional behaviour as an interaction process, rather than arithmetic calculation itself.

(3) $\overline{f}(ac)([\![1]\!]_a|c[\&_{n\in\omega}\overline{e}\mathtt{in}_n])$ represents an open $\lambda$-term $f : \mathbb{N}\Rightarrow\mathbb{N} \vdash f1 : \mathbb{N}$ with $\mathbb{N}$ denoting the type of natural numbers. The process first inquires at $f$ carrying two new channels. At the first channel it in turn may receive an inquiry, to which it provides the argument 1. At the second channel it receives an answer from the function, which it simply forwards to $e$. We can check, with $[\![\mathsf{succ}]\!]_f \stackrel{\text{def}}{=} !f(ue).\overline{u}(c)c[\&_{n\in\omega}\overline{e}\mathtt{in}_{n+1}]$:

$$[\![\mathsf{succ}]\!]_f|\overline{f}(ac)([\![1]\!]_a|c[\&_{n\in\omega}\overline{e}\mathtt{in}_n]) \twoheadrightarrow [\![\mathsf{succ}]\!]_f|\overline{e}\mathtt{in}_2|(\boldsymbol{\nu}\, a)[\![1]\!]_a$$

Note $(\boldsymbol{\nu}\, a)[\![1]\!]_a$ remains as a garbage without any further potential interaction.

(4) Define $[x \to y] \stackrel{\text{def}}{=} !x(c).\overline{y}(c')c'[\&_{i\in\omega}\overline{c}\mathtt{in}_i]$. This is a *copy-cat agent*, linking two locations $x$ and $y$. Having this agent in-between does not change the whole behaviour. For example: $[\![2]\!]_y|[x \to y]|\overline{x}\langle e\rangle \longrightarrow [\![2]\!]_y|[x \to y]|\overline{y}(c')c'[\&_i\overline{e}\mathtt{in}_i] \longrightarrow [\![2]\!]_y|[x \to y]|(\boldsymbol{\nu}\, c')(\overline{c'}\mathtt{in}_2|c'[\&_i\overline{e}\mathtt{in}_i]) \longrightarrow [\![2]\!]_y|[x \to y]|\overline{e}\mathtt{in}_2$, which is the same as $[\![2]\!]_y|\overline{y}\langle e\rangle \longrightarrow [\![2]\!]_y|\overline{e}\mathtt{in}_2$ save some internal reductions.

(5) Let $\Omega_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)([u \to y]|[y \to u])$. Then we can check $\Omega_u|\overline{u}\langle e\rangle \longrightarrow\longrightarrow\longrightarrow \ldots$. $\Omega_u$ immediately diverges after the initial invocation.

### 2.3 Linear/Affine Typing

Linearity and affinity are fundamental concepts which have long been studied and applied in semantics of computation. Below we show one of the ways to incorporate these ideas into the $\pi$-calculus, integrating the linear type discipline in [66] and the affine type discipline in [8]. On the basis of [8; 66], the following three key steps lead to the coherent integrations of affinity and linearity.

- distinction between linear and affine action modes;
- constraint in channel types such that linear output cannot be carried by affine replication; and
- constraint in prefix typing such that affine input never suppresses linear output.

Since our main objective in this paper is applications, the following presentation of the linear/affine type discipline focusses on the salient features of this calculus in the present context. For detailed discussions of this type discipline, see [67; 65].

**Action Modes.** We use the following *action modes* [8; 22; 27; 66], which prescribe different modes of interaction at each channel.

| | | | |
|---|---|---|---|
| $\downarrow_{\mathrm{L}}$ | 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 $\updownarrow$ to indicate uncomposability at linear/affine channels. $p, p', \ldots$ 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 also write:

$$\mathcal{M}_{\downarrow} = \{\downarrow_{\mathrm{L}}, \downarrow_{\mathrm{A}}\}, \mathcal{M}_{\uparrow} = \{\uparrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}\}, \mathcal{M}_{!} = \{!_{\mathrm{L}}, !_{\mathrm{A}}\}, \text{ and } \mathcal{M}_{?} = \{?_{\mathrm{L}}, ?_{\mathrm{A}}\}.$$

The L-modes correspond to linear modes in [66] while the A-modes to affine modes in [8]. The difference between linearity and affinity in non-replicated channels is that, in a linear channel, an interaction takes place *precisely once*, while it does so *at most once* in an affine channel. In replicated channel, linearity essentially means convergence while affinity means potential divergence.

**Channel Types.** Next we define *channel types* by the following grammar. Below $p_{\mathrm{I}}$ (resp. $p_{\mathrm{O}}$) denotes input (resp. output) modes.

$$\tau \;::=\; \tau_{\mathrm{I}} \mid \tau_{\mathrm{O}} \mid \updownarrow \qquad \tau_{\mathrm{I}} \;::=\; (\vec{\tau})^{p_{\mathrm{I}}} \mid [\&_i\vec{\tau}_i]^{p_{\mathrm{I}}} \qquad \tau_{\mathrm{O}} \;::=\; (\vec{\tau})^{p_{\mathrm{O}}} \mid [\oplus_i\vec{\tau}_i]^{p_{\mathrm{O}}}$$

$(\vec{\tau})^{p_{\mathrm{I}}}$ and $(\vec{\tau})^{p_{\mathrm{O}}}$ are *unary input/output types*, respectively, while $[\&_i\vec{\tau}_i]^p$ and $[\oplus_i\vec{\tau}_i]^p$ are *branching/selection type*. We sometimes write $[\vec{\tau}_1\&\vec{\tau}_2]^p/[\vec{\tau}_1 \oplus \vec{\tau}_2]^p$ for binary branching/selection type. $\mathsf{md}(\tau)$ denotes the outermost mode of $\tau$ except we set $\mathsf{md}(\updownarrow) = \updownarrow$. The *dual of* $\tau$, written $\overline{\tau}$, is the result of dualising all action modes and exchanging & and $\oplus$ in $\tau$. We define $\odot$ is the least commutative partial operation on channel types such that:

$$\begin{array}{lllll} (1) & \tau \odot \overline{\tau} = \updownarrow & (\mathsf{md}(\tau) \in \mathcal{M}_{\downarrow}) & & \\ (2) & \tau \odot \tau = \tau & (\mathsf{md}(\tau) \in \mathcal{M}_{?}) & \tau \odot \overline{\tau} = \tau & (\mathsf{md}(\tau) \in \mathcal{M}_{!}) \end{array}$$

Intuitively, (1) says once we compose input-output linear channels it becomes uncomposable. (2) says that a server should be unique, to which an arbitrary number of clients can request interactions. The following well-formedness condition is an integral part of the present type discipline.

**Definition 2.2** The set of well-formed channel types is inductively generated by the following conditions.

**(C1).** $(\vec{\tau})^p$ with $p \in \mathcal{M}_{\downarrow}$ is well-formed when each $\tau_i$ is well-formed and, moreover, $\mathsf{md}(\tau_i) \in \mathcal{M}_{?}$ for each $i$. Dually when $p \in \mathcal{M}_{\uparrow}$.

**(C2).** $(\vec{\tau})^{!_{\mathrm{L}}}$ is well-formed when each $\tau_i$ is well-formed and, moreover, there exists a unique $j$ s.t. $\mathsf{md}(\tau_j) \in \{\uparrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}\}$, while $\mathsf{md}(\tau_i) \in \mathcal{M}_{?}$ for others. Dually for $(\vec{\tau})^{?_{\mathrm{L}}}$.

**(C3).** $(\vec{\tau})^{!_{\mathrm{A}}}$ is well-formed when each $\tau_i$ is well-formed and, moreover, there is a unique $j$ s.t. $\mathsf{md}(\tau_j) = \uparrow_{\mathrm{A}}$, while $\mathsf{md}(\tau_i) \in \mathcal{M}_{?}$ for others. Dually for $(\vec{\tau})^{?_{\mathrm{A}}}$.

Similarly for branching/selection types, imposing the same constraint for each summand in branching/selection types. *Hereafter we assume all channel types are well-formed unless otherwise stated.*

In a well-formed type, an input only carries an output (and dually), and only a replicated input carries a unique linear/affine output (and dually). These conditions come from game semantics [34; 4]. An additional condition, given in (C2) and (C3) is that a linear output $\uparrow_{\mathrm{L}}$ can only be carried by a linear replicated input $!_{\mathrm{L}}$ (and dually), which is crucial for consistently integrating linearity and affinity. This is because an invocation at linear replication will eventually terminate, firing an associated linear output (see § 2 in [65] for more discussions). Some examples of channel types follow.

**Example 2.3** (channel types)

(1) $()^{\uparrow_{\mathrm{L}}}$ is a type indicating an output without carrying any value which takes place exactly one; $()^{\uparrow_{\mathrm{A}}}$ represents an output which takes at most once. Both $()^{\uparrow_{\mathrm{L}}}$ and $()^{\uparrow_{\mathrm{A}}}$ are (vacuously) well-formed. Further $\overline{()^{\uparrow_{\mathrm{L}}}} = ()^{\downarrow_{\mathrm{L}}}$ and $\overline{()^{\uparrow_{\mathrm{A}}}} = ()^{\downarrow_{\mathrm{A}}}$.

(2) Let $\mathbb{N}^{\bullet} = [\oplus_{i\in\omega}]^{\uparrow_{\mathrm{L}}}$. This type represents an output of a natural number which is done precisely once. $\mathbb{N}^{\circ} = (\mathbb{N}^{\bullet})^{!_{\mathrm{L}}}$ is a type which can repeatedly receive an invocation carrying one name, and through that name necessarily sends a natural number. Note $\overline{\mathbb{N}^{\bullet}} = [\&_{i\in\omega}]^{\downarrow_{\mathrm{L}}}$ and $\overline{\mathbb{N}^{\circ}} = ([\&_{i\in\omega}]^{\downarrow_{\mathrm{L}}})^{?_{\mathrm{L}}}$. The latter is a type which inquires at a replicated channel carrying a unique name as its parameter, and via that name receives a natural number precisely once.

$$\frac{}{\vdash \mathbf{0} \triangleright \_}\ (\mathsf{Zero}) \qquad \frac{\vdash P_i \triangleright A_i \quad (i=1,2) \qquad A_1 \asymp A_2}{\vdash P_1|P_2 \triangleright A_1 \odot A_2}\ (\mathsf{Par})$$

$$\frac{\vdash P \triangleright A^{x:\tau} \qquad \mathsf{md}(\tau) \in \mathcal{M}_! \cup \{\updownarrow\}}{\vdash (\boldsymbol{\nu}\, x)P \triangleright A/x}\ (\mathsf{Res}) \qquad \frac{\vdash P \triangleright A^{-x} \qquad \mathsf{md}(\tau) \in \mathcal{M}_? \cup \{\updownarrow\}}{\vdash P \triangleright A \otimes x{:}\tau}\ (\mathsf{Weak})$$

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

$$\frac{\vdash P \triangleright \vec{y}{:}\vec{\tau} \otimes \uparrow_{\mathrm{A}} ?A^{-x}}{\vdash x(\vec{y}).P \triangleright x{:}(\vec{\tau})^{\downarrow_{\mathrm{A}}} \otimes A}\ (\mathsf{In}^{\downarrow_{\mathrm{A}}}) \qquad \frac{\vdash P \triangleright \vec{y}{:}\vec{\tau} \otimes ?_{\mathrm{L}}?_{\mathrm{A}} A^{-x}}{\vdash\, ! x(\vec{y}).P \triangleright x{:}(\vec{\tau})^{!_{\mathrm{A}}} \otimes A}\ (\mathsf{In}^{!_{\mathrm{A}}}) \qquad \frac{\tau_i = \tau_j \quad (\text{if } y_i = y_j)}{\vdash \overline{x}\langle\vec{y}\rangle \triangleright x{:}(\vec{\tau})^{p_{\mathrm{O}}} \otimes (\odot_i\, y_i{:}\overline{\tau_i})}\ (\mathsf{Out})$$

Fig. 3. Linear/Affine Typing (composition and unary prefix)

(3) $(()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$, $(()^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}$ and $(\overline{\mathbb{N}^{\circ}}()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$ are well-formed, but $(()^{\downarrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$, $(()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{A}}}$ and $(\overline{\mathbb{N}^{\circ}})^{!_{\mathrm{L}}}$ are not.

**Action Types and Typing**. 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. We write $x{:}\tau$ for an assignment of $\tau$ to $x$, and $x{:}\tau \to x'{:}\tau'$ when $x'$ causally depends on $x$. $x{:}\tau \to x'{:}\tau'$ is allowed if either (1) $\mathsf{md}(\tau) = \downarrow_{\mathrm{L}}$ and $\mathsf{md}(\tau) = \uparrow_{\mathrm{L}}$ or (2) $\mathsf{md}(\tau) = !_{\mathrm{L}}$ and $\mathsf{md}(\tau) = ?_{\mathrm{L}}$.

Action types control the formation of typed processes via three operations: *composition* $A \odot B$, with the associated composability relation $A \asymp B$; *prefix* $x{:}\tau \to A$ and *hiding* $A/x$. Intuitively, $A \odot B$ composes two action types after checking, using $\asymp$, the consistency of type assignment and the lack of circularity in causality; $x{:}\tau \to A$ adds an edge from a new node $x{:}\tau$ to the unsuppressed existing nodes; and $A/x$ takes off a node with name $x$ (if any) from $A$. For formal definitions, see Appendix A. $\odot$ is an associative operator whose $n$-ary extension is written $\odot_i A_i$.

The typing rules are given in Figure 3 and Figure 4. In each rule we assume channel types are well-formed, and processes obey the standard bound name condition. The rules use the following notations ($\mathsf{fn}(A)$ and $\mathsf{md}(A)$ denote the sets of free names and modes of $A$, respectively).

- $A^{x:\tau}$ indicates $x{:}\tau$ occurs in $A$. $A^{-x}$ indicates $x \notin \mathsf{fn}(A)$
- $\vec{p}A$ indicates $\mathsf{md}(A) \subset \{\vec{p}\}$, while $?A$ indicates $\mathsf{md}(A) \subset \mathcal{M}_?$.
- $A \otimes B$ denotes the disjoint union of $A$ and $B$, assuming $\mathsf{fn}(A) \cap \mathsf{fn}(B) = \emptyset$.

We give a brief illustration of typing rules.

- (Par) uses $\asymp$ for controlling composition. (Res) allows hiding of a name only when its mode is $\updownarrow$ or replicated (so that channels of modes $\uparrow$, $\downarrow$ or $?$ should be

$$\textsf{(Bra}^{\downarrow_{\mathrm{L}}}\textsf{)}\quad \frac{\vdash P_i \triangleright \vec{y}_i{:}\vec{\tau}_i \otimes \uparrow_{\mathrm{L}} A^{\text{-}x} \otimes \uparrow_{\mathrm{A}} \mathbf{?}B^{\text{-}x}}{\vdash x[\&_i(\vec{y}_i).P_i] \triangleright (x{:}[\&_i\vec{\tau}_i]^{\downarrow_{\mathrm{L}}} \to A) \otimes B} \qquad \textsf{(Bra}^{!_{\mathrm{L}}}\textsf{)}\quad \frac{\vdash P \triangleright \vec{y}_i{:}\vec{\tau}_i \otimes \mathbf{?}_{\mathrm{L}} A^{\text{-}x} \otimes \mathbf{?}_{\mathrm{A}} B^{\text{-}x}}{\vdash! x[\&_i(\vec{y}_i).P_i] \triangleright (x{:}[\&_i\vec{\tau}_i]^{!_{\mathrm{L}}} \to A) \otimes B}$$

$$\textsf{(Bra}^{\downarrow_{\mathrm{A}}}\textsf{)}\quad \frac{\vdash P_i \triangleright \vec{y}_i{:}\vec{\tau}_i \otimes \uparrow_{\mathrm{A}} \mathbf{?} A^{\text{-}x}}{\vdash x[\&_i(\vec{y}_i).P_i] \triangleright x{:}[\&_i\vec{\tau}]^{\downarrow_{\mathrm{A}}} \otimes A} \qquad \textsf{(Bra}^{!_{\mathrm{A}}}\textsf{)}\quad \frac{\vdash P_i \triangleright \vec{y}_i{:}\vec{\tau}_i \otimes \mathbf{?}_{\mathrm{L}}\mathbf{?}_{\mathrm{A}} A^{\text{-}x}}{\vdash! x[\&_i(\vec{y}_i).P_i] \triangleright x{:}[\&_i\vec{\tau}_i]^{!_{\mathrm{A}}} \otimes A}$$

$$\textsf{(Sel)}\quad \frac{\tau_{ij} = \tau_{ij'} \quad (\text{if } y_i = y_j)}{\vdash \overline{x}\mathtt{in}_i\langle\vec{y}\rangle \triangleright x{:}[\oplus_i\vec{\tau}_i]^{p_{\mathrm{O}}} \otimes (\odot_j\, y_j{:}\tau_{ij})}$$

Fig. 4. Typing Branching and Selection

compensated by their duals before restricted). (Weak) weakens $\updownarrow$ and $\mathbf{?}$-nodes since they allow the possibility of no action.

- In $(\textsf{In}^{\downarrow_{\mathrm{L}}})$ and $(\textsf{In}^{!_{\mathrm{L}}})$, we record causality for ensuring linearity. In these rules and $(\textsf{In}^{\downarrow_{\mathrm{A}}})$ and $(\textsf{In}^{!_{\mathrm{A}}})$, an input never suppresses another input. Note $(\textsf{In}^{!_{\mathrm{L}}})$ never suppresses $\uparrow_{\mathrm{L}}/\uparrow_{\mathrm{A}}$ actions (except for that which is abstracted).
- In $(\textsf{In}^{\downarrow_{\mathrm{A}}})$, $\downarrow_{\mathrm{A}}$ never suppresses $\uparrow_{\mathrm{L}}$, which is crucial for integration (suppose $x$ 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 [65]).
- In (Out), the condition $\tau_i = \tau_j$ if $y_i = y_j$ ensures that a duplicated object name is assigned the same type (for example, $\overline{x}\langle yy\rangle$ has type $x : (\tau\tau)^{\uparrow_{\mathrm{L}}} \otimes y : \overline{\tau}$ with $\mathsf{md}(\tau) \in \mathcal{M}_?$). The rule assigns the dual of the corresponding carried type to each $y_i$. This is because a passed name will eventually be used by the dual of its own type (for example, $u$ in $[\![2]\!]_u|\overline{u}\langle e\rangle$ is typed by $\overline{\mathbb{N}^\circ} = ([\&_{i\in\omega}]^{\downarrow_{\mathrm{L}}})^{\mathbf{?}_{\mathrm{L}}}$, while $e$, via which 2 will be outputted, should have a type $[\oplus_{i\in\omega}]^{\uparrow_{\mathrm{L}}}$).
- The typing rules for branching and selection, given in Figure 4, are similar to those for unary prefixes. In the antecedent of each branching rule, 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 $\lambda$-calculus and additives in Linear Logic.

The following variant of (Out), which is specialised for bound output [8; 9; 27; 66; 67], is a permissible rule in the calculus. This rule is often useful for type inference of various encodings (in fact the rule is equipotent to (Out) via simple syntactic translation [23]).

$$\textsf{(Bout)}\quad \frac{\vdash P \triangleright C^{\vec{y}:\vec{\tau}} \quad C \asymp x{:}(\vec{\tau})^{p_{\mathrm{O}}}}{\vdash \overline{x}(\vec{y})P \triangleright C/\vec{y} \odot x{:}(\vec{\tau})^{p_{\mathrm{O}}}} \qquad (1)$$

Similarly for selection with bound output, which we omit.

Some examples of typed terms follow (processes are from Example 2.1).

**Example 2.4** (typed processes)

(1) $\vdash [\![n]\!]_x \triangleright x{:}\mathbb{N}^\circ$ is well-typed. Similarly $\vdash [\![\mathsf{succ}]\!]_x \triangleright x{:}(\overline{\mathbb{N}^\circ}\mathbb{N}^\bullet)^{!_{\mathrm{L}}}$ is well-typed.
(2) $\vdash x.\overline{y} \triangleright x{:}()^{\downarrow_{\mathrm{L}}} \to y{:}()^{\uparrow_{\mathrm{L}}}$ is well-typed, but $\vdash x.\overline{y} \triangleright x{:}()^{\downarrow_{\mathrm{A}}} \otimes y{:}()^{\uparrow_{\mathrm{L}}}$ is not.
(3) The copy-cat (cf. Example 2.1 (3)) can be typed as $\vdash [u \to x] \triangleright u{:}\mathbb{N}^\circ \to x{:}\overline{\mathbb{N}^\circ}$. This sequent can be considered as the encoding of a typed variable, $x{:}\mathbb{N} \vdash x{:}\mathbb{N}$, located at $u$.
(4) Let $\mathbb{N}^\diamond$ be the affine version of $\mathbb{N}^\circ$, i.e. $\mathbb{N}^\diamond \stackrel{\text{def}}{=} ([\oplus_{i\in\omega}]^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}$. We can then type $[x \to x']$ with a different action type: $[x \to x']$: $\vdash [x \to x'] \triangleright x{:}\mathbb{N}^\diamond \otimes x'{:}\overline{\mathbb{N}^\diamond}$.
(5) The divergent agent $\Omega_u$ is typed as $\vdash \Omega_u \triangleright u{:}\mathbb{N}^\diamond$, starting from (4) above.

We write $\pi^{\mathrm{LA}}$ for the resulting typed calculus. Salient features of this calculus include subject reduction and fully abstract embeddability of various functional calculi, such as the simply typed $\lambda$-calculus with products and sums, call-by-name PCF and call-by-value PCF. We use this calculus as a basis of our secrecy analysis in the subsequent sections.

## 3. SECRECY ANALYSIS IN THE LINEAR/AFFINE $\pi$-CALCULUS

### 3.1 Secrecy Annotations and Tampering Level

**Channel Types.** Let $(\mathcal{L}, \sqsubseteq, \top, \bot)$ be a complete lattice of secrecy levels (higher means more secure), whose elements are written $s, s', \ldots$. We first annotate channel types with these secrecy levels. Annotated types are still written $\tau, \tau', \ldots$, generated from the following grammar.

$$\tau ::= \tau_{\mathrm{I}} \mid \tau_{\mathrm{O}} \mid \updownarrow_s \qquad \tau_{\mathrm{I}} ::= (\vec{\tau})^{p_{\mathrm{I}}}_s \mid [\&_i\vec{\tau}_i]^{p_{\mathrm{I}}}_s \qquad \tau_{\mathrm{O}} ::= (\vec{\tau})^{p_{\mathrm{O}}}_s \mid [\oplus_i\vec{\tau}_i]^{p_{\mathrm{O}}}_s$$

Duality is defined respecting secrecy levels (e.g. $()^{\uparrow_{\mathrm{A}}}_s$ and $()^{\downarrow_{\mathrm{A}}}_{s'}$ are dual iff $s = s'$). We write $\mathsf{sec}(\tau)$ for the outermost secrecy level of $\tau$.

**Action Types and Tamper Level.** The secrecy typing uses the same action types as before, except we use the annotated channel types. We still write $A, B, \ldots$ for secrecy-enhanced action types. We use a function which maps an action type to a secrecy level. We call this level a *tamper level* (in the sense that it is the level at which the process may affect, or tamper, the environment). It is first defined on channel types, which is then extended to action types.

**Definition 3.1** $\tau$ is *immediately tampering* if either $\tau = [\oplus_i\vec{\tau}_i]^{\uparrow_{\mathrm{L}}}$ or $\mathsf{md}(\tau) = \uparrow_{\mathrm{A}}$. $\tau$ is *innocuous* if $\mathsf{md}(\tau) \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}, \updownarrow\}$.

**Definition 3.2** The *tamper level of* $\tau$, denoted $\mathsf{tamp}(\tau)$, is inductively given by:

$$\begin{array}{rcll} \mathsf{tamp}(\tau) & = & \mathsf{sec}(\tau) & \text{if } \tau \text{ is immediately tampering.} \\ \mathsf{tamp}(\tau) & = & \top & \text{if } \tau \text{ is innocuous.} \\ \mathsf{tamp}((\vec{\tau})^p_s) & = & \sqcap\{\mathsf{tamp}(\tau_i)\} & \text{with } p \in \mathcal{M}_{!,\downarrow} \cup \{\uparrow_{\mathrm{L}}\}. \\ \mathsf{tamp}([\&_i\vec{\tau}_i]^p_s) & = & \sqcap\{\mathsf{tamp}(\tau_{ij})\} & \text{with } p \in \mathcal{M}_{!,\downarrow}. \end{array}$$

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 the resource, leaving the environment as it originally was. Thus the tampering level of $?_{\mathrm{L}}$-types is always $\top$. Note this discussion relies on stateless nature of recursive behaviour of type $!_{\mathrm{L}}$ and $!_{\mathrm{A}}$. The notion also relates to inhabitation properties of $\pi^{\mathrm{LA}}$ (cf. [67]). A few concrete calculations of tamper levels follow.

**Example 3.3** (tampering level)

(1) $()_s^{\uparrow_{\mathrm{L}}}$ is not immediately tampering: in fact, its tamper level is $\top$. This is because this type represents a behaviour which necessarily sends an empty output precisely once: its behaviour is completely determined, so no information is transmitted by interaction. On the other hand $[\oplus_{i\in\omega}]_s^{\uparrow_{\mathrm{L}}}$ is immediately tampering and does have a non-trivial tamper level, $s$.

(2) Both $()_s^{\uparrow_{\mathrm{A}}}$ and $[\oplus_{i\in\omega}]_s^{\uparrow_{\mathrm{A}}}$ are immediately tampering, with the same tamper level $s$. Intuitively $()_s^{\uparrow_{\mathrm{A}}}$ transmits information by having two possibilities: either outputting at that channel, or not doing so at all.

(3) Let $\tau = ()_s^{\uparrow_{\mathrm{L}}})_{s'}^{!_{\mathrm{L}}}$. Then $\mathsf{tamp}(\tau) = \top$ regardless of $s$ and $s'$. The 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 $c$ contain information.

(4) Let $\mathbb{N}_s^\circ \stackrel{\mathrm{def}}{=} ([\oplus_{i\in\omega}]_s^{\uparrow_{\mathrm{L}}})_{s'}^{!_{\mathrm{L}}}$. Then we have $\mathsf{tamp}(\mathbb{N}_s^\circ) = s$. $\mathbb{N}_s^\circ$ is not immediately tampering, but it affects the environment latently. To see this, take $[\![2]\!]_x \stackrel{\mathrm{def}}{=} !x(c).\overline{c}\mathtt{in}_2$ which, in the typing system presented later, has type $x\!:\!\mathbb{N}_s^\circ$. This process does contain information which it emits after the invocation at $x$. Similarly for $((\,)_s^{\uparrow_{\mathrm{A}}})_{s'}^{!_{\mathrm{A}}}$.

(5) By definition, $\mathsf{tamp}(\tau) = \top$ for any $\tau$ such that $\mathsf{md}(\tau) \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}\}$.

If neither $\tau$ nor $\overline{\tau}$ is immediately tampering (i.e. either $\mathsf{md}(\tau) \in \{\updownarrow, !_{\mathrm{L}}, !_{\mathrm{A}}, ?_{\mathrm{L}}, ?_{\mathrm{A}}\}$ or it is unary and $\mathsf{md}(\tau) \in \{\downarrow_{\mathrm{L}}, \uparrow_{\mathrm{L}}\}$), then $\mathsf{sec}(\tau)$ is not used for calculating the tampering level: nor would it be anyway used in the secrecy typing given next. For this reason we stipulate:

**Convention 3.4** *From now on we usually omit the outermost secrecy level of a channel type if neither the type nor its dual is immediately tampering.*

For example, $\mathbb{N}_s^\circ$ above would be written $([\oplus_{i\in\omega}]_s^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$, omitting the outermost secrecy level.

## 3.2 Secrecy Typing

To distinguish from the underlying linear/affine typing, the secrecy typing uses the sequent of the form $\vdash_{\mathsf{sec}} P \triangleright A$, where $A$ is now an action type which uses secrecy-annotated channel types. The typing rules are the same as those in Figures 3 and 4 (under Convention 3.4), except for the three rules, $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$, $(\mathsf{Bra}^{\downarrow_{\mathrm{L}}})$ and $(\mathsf{Bra}^{\downarrow_{\mathrm{A}}})$,

$$\frac{\vdash_{\mathrm{sec}} P \triangleright \vec{y}{:}\vec{\tau} \otimes \uparrow_{\mathrm{A}} ?A^{\text{-}x} \quad s \sqsubseteq \mathsf{tamp}(A)}{\vdash_{\mathrm{sec}} x(\vec{y}).P \triangleright x{:}(\vec{\tau})^{\downarrow_{\mathrm{A}}}_{s} \otimes A} \ (\mathsf{In}^{\downarrow_{\mathrm{A}}})$$

$$\frac{\vdash_{\mathrm{sec}} P_i \triangleright \vec{y}_i{:}\vec{\tau}_i \otimes \uparrow_{\mathrm{L}} A^{\text{-}x} \otimes \uparrow_{\mathrm{A}} ?B^{\text{-}x} \quad s \sqsubseteq \mathsf{tamp}(A \otimes B)}{\vdash_{\mathrm{sec}} x[\&_i(\vec{y}_i).P_i] \triangleright (x{:}[\&_i \vec{\tau}_i]^{\downarrow_{\mathrm{L}}}_{s} \to A) \otimes B} \ (\mathsf{Bra}^{\downarrow_{\mathrm{L}}})$$

$$\frac{\vdash_{\mathrm{sec}} P_i \triangleright \vec{y}_i{:}\vec{\tau}_i \otimes \uparrow_{\mathrm{A}} ?A^{\text{-}x} \quad s \sqsubseteq \mathsf{tamp}(A)}{\vdash_{\mathrm{sec}} x[\&_i(\vec{y}_i).P_i] \triangleright x{:}[\&_i \vec{\tau}]^{\downarrow_{\mathrm{A}}}_{s} \otimes A} \ (\mathsf{Bra}^{\downarrow_{\mathrm{A}}})$$

Fig. 5. Secrecy Typing (rules with added constraints)

which are now replaced by the rules in Figure 5. Each new rule adds a condition on secrecy levels. We say $\vdash_{\mathrm{sec}} P \triangleright A$ is *well-typed* (or $P$ is *secure under* $A$) when $\vdash_{\mathrm{sec}} P \triangleright A$ is derivable from the secrecy typing rules. Some observations on the secrecy-sensitive rules:

- In $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$, 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.
- In $(\mathsf{Bra}^{\downarrow_{\mathrm{L}}})$, a linear branching receives information by being invoked at one of its branches. In $(\mathrm{Bra}^{\downarrow_{\mathrm{A}}})$, an affine branching receives information both by branching and affinity. Again these effects should not be transmitted at levels which are the same as or above the receiving level.

We illustrate the secrecy typing by a few simple examples (cf. Example 2.4).

**Example 3.5** (securely typed terms)

(1) $\vdash_{\mathrm{sec}} [\![n]\!]_x \triangleright x{:}\mathbb{N}^{\circ}_{s}$ is well-typed for each $s$.

(2) $\vdash_{\mathrm{sec}} x.\overline{y} \triangleright x : ()^{\downarrow_{\mathrm{L}}}_{\top} \to y : ()^{\uparrow_{\mathrm{L}}}_{\bot}$ and $\vdash_{\mathrm{sec}} x.\overline{y} \triangleright x : ()^{\downarrow_{\mathrm{A}}}_{\bot} \otimes y : ()^{\uparrow_{\mathrm{A}}}_{\top}$ are well-typed, but $\vdash_{\mathrm{sec}} x.\overline{y} \triangleright x : ()^{\downarrow_{\mathrm{A}}}_{\top} \otimes y : ()^{\uparrow_{\mathrm{A}}}_{\bot}$ is not. Also $\vdash_{\mathrm{sec}} u[.\overline{x}(y)[\![1]\!]_y \& .\overline{x}(y)[\![2]\!]_y] \triangleright u : [\ \&\ ]^{\downarrow_{\mathrm{L}}}_{\top} \to x{:}(\mathbb{N}^{\circ}_{s})^{\uparrow_{\mathrm{L}}}$ is well-typed iff $s = \top$ (note $\mathsf{tamp}((\mathbb{N}^{\circ}_{s})^{\uparrow_{\mathrm{L}}}) = s$).

(3) $\vdash_{\mathrm{sec}} [\![\mathsf{succ}]\!]_u \triangleright u{:}(\overline{\mathbb{N}^{\circ}_{s}}\mathbb{N}^{\bullet}_{s'})^{!_{\mathrm{L}}}$ is well-typed iff $s \sqsubseteq s'$. Remembering the definition of $[\![\mathsf{succ}]\!]_u$ from Example 2.1, it suffices to check $\vdash_{\mathrm{sec}} c[\&_{n\in\omega}.\overline{e}\mathtt{in}_{n+1}] \triangleright c{:}\overline{\mathbb{N}^{\bullet}_{s}} \to e{:}\mathbb{N}^{\bullet}_{s'}$ is well-typed, the condition for which is nothing but $s \sqsubseteq s'$.

(4) $\vdash_{\mathrm{sec}} [x \to y] \triangleright x{:}\mathbb{N}^{\circ}_{s} \to y{:}\overline{\mathbb{N}^{\circ}_{s'}}$ is well-typed iff $s' \sqsubseteq s$, with the same reasoning as above. Similarly $\vdash_{\mathrm{sec}} [x \to y] \triangleright x{:}\mathbb{N}^{\circ}_{s} \otimes y{:}\overline{\mathbb{N}^{\circ}_{s'}}$ is well-typed iff $s' \sqsubseteq s$.

(5) $\vdash_{\mathrm{sec}} \Omega_x \triangleright x{:}\mathbb{N}^{\circ}_{s}$ is always well-typed.

More substantial examples will be presented in the next section through the embedding of secrecy analysis of call-by-name and call-by-value functions in the secrecy analysis for $\pi^{\mathsf{LA}}$.

### 3.3 Basic Properties of Secure Typing

This subsection discusses a couple of key results in the secrecy analysis for $\pi^{\mathsf{LA}}$. The first property is the following result.

**Proposition 3.6** (subject reduction) *If* $\vdash_{\mathsf{sec}} P \triangleright A$ *and* $P \twoheadrightarrow Q$ *then* $\vdash_{\mathsf{sec}} Q \triangleright A$.

PROOF. First we prove commutativity and associativity of the operator $\odot$ on action types. Secondly we show closure under $\equiv$. The closure under $\longrightarrow$ is proved using the following substitution lemma:

(1) (linear type) If $\vdash_{\mathsf{sec}} P \triangleright x{:}\tau \otimes A$, $\mathsf{md}(\tau) \in \mathcal{M}_{\uparrow}$ and $y \notin \mathsf{fn}(A)$, then $\vdash_{\mathsf{sec}} P\{y/x\} \triangleright y{:}\tau \otimes A$ and $\mathsf{tamp}(x{:}\tau \otimes A) = \mathsf{tamp}(y{:}\tau \otimes A)$.
(2) (client type) If $\vdash_{\mathsf{sec}} P \triangleright x : \tau \otimes A$ and $A(y) = \tau$, then $\vdash_{\mathsf{sec}} P\{y/x\} \triangleright A$ and $\mathsf{tamp}(x{:}\tau \otimes A) = \mathsf{tamp}(A)$.

The rest is a routine, by the rule induction on the reduction rules. See Appendix C.1 for the proof. □

A basic element of the present theory of secrecy analysis is a family of contextual congruences relativised by secrecy levels. We first define the notion of observables.

**Definition 3.7** We write $P \Downarrow_x$ when $P \longrightarrow^* P'$ such that either $P' \equiv (\boldsymbol{\nu}\, \vec{z})(\overline{x}\langle \vec{y} \rangle | R)$ or $P' \equiv (\boldsymbol{\nu}\, \vec{z})(\overline{x}\mathtt{in}_i\langle \vec{y} \rangle | R)$ such that $x \notin \{\vec{z}\}$.

Note we only take an output as an observable [26; 28]. Using this observable, we define the secrecy-sensitive congruence. Below a *typed congruence* is an equivalence relation over $\pi^{\mathsf{LA}}_{\mathsf{sec}}$-terms which always relates two typed terms with the same typing and which is closed under typed contexts.

**Definition 3.8** (secrecy-sensitive contextual congruence) $\cong^{\pi}_{s}$ is the maximum typed congruence which satisfies the following condition: if $\vdash_{\mathsf{sec}} P_1 \cong^{\pi}_{s} P_2 \triangleright x : ()^{\uparrow_{\mathsf{A}}}_{s}$ and $P_1 \Downarrow_x$ then $P_2 \Downarrow_x$.

Intuitively, $\cong_s$ ignores actions which are not observable from the level $s$. By definition we immediately obtain:

**Proposition 3.9** *If* $\vdash_{\mathsf{sec}} P_1 \cong_s P_2 \triangleright A$ *then* $s \sqsupseteq s'$ *implies* $\vdash_{\mathsf{sec}} P_1 \cong_{s'} P_2 \triangleright A$. *In particular,* $\vdash_{\mathsf{sec}} P_1 \cong_{\top} P_2 \triangleright A$ *implies* $\vdash_{\mathsf{sec}} P_1 \cong_s P_2 \triangleright A$ *for each* $s$.

For reasoning about processes using $\cong_s$, one of the basic tools is a context lemma.

**Lemma 3.10** (context lemma) *Let* $\vdash_{\mathsf{sec}} P_{1,2} \triangleright A$. *Then* $P_1 \cong_s P_2$ *if and only if, for each* $\vdash_{\mathsf{sec}} R \triangleright \overline{A} \otimes x{:}()^{\uparrow_{\mathsf{A}}}_{s}$, $(\boldsymbol{\nu}\, \mathsf{fn}(A))(P_1|R) \Downarrow_x$ *iff* $(\boldsymbol{\nu}\, \mathsf{fn}(A))(P_2|R) \Downarrow_x$.

PROOF. The "only if" direction is immediate from the definition. For the "if" direction, let $C[\cdot]$ be a context with hole typed $A$ and the result typed $x : ()^{\uparrow_{\mathsf{A}}}_{s}$ with $x$ fresh (if $x \in \mathsf{fn}(A)$ we can always use a copy-cat to mediate $x$ to a fresh name). Assume the latter condition and $C[P_1] \Downarrow_x$. If the hole of $C[\ \cdot\ ]$ is not under an input prefix, then we already have $C[\ \cdot\ ] \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, \mathsf{fn}(A))(R|[\ \cdot\ ])$. Suppose the hole is under an input prefix. If $C[P_1] \Downarrow_x$ by $C[P_1] \twoheadrightarrow \overline{x}|C'[P_1\sigma]$ keeping $P_1$ under the input prefix along the way (possibly with some substitution $\sigma$) then we have $C[P_2] \twoheadrightarrow \overline{x}|C'[P_2]$, i.e. $C[P_2] \Downarrow_x$. If not, then suppose $C[P_1] \twoheadrightarrow C'[P_1\sigma]$ where $C'[P_1\sigma]$ is the first configuration in which the input prefix is taken off. Using copycats (cf. Appendix B) we can represent $\sigma$ by parallel composition and hiding, so that the former condition gives us $C[P_2] \Downarrow_x$, as required. □

One of the most basic properties in any secrecy analysis is *noninterference*, which essentially says that high-level data never interfere with low-level observable behaviour. Since data and programs are all processes in the present context, and because $P \cong_s Q$ means $P$ and $Q$ have the same $s$-level observable behaviour, the noninterference result in the present context simply says that two processes with a secrecy level incompatible with $s$ can always be equated by $\cong_s$.

**Proposition 3.11** (non-interference in $\pi^{\mathsf{LA}}$) *If* $\vdash_{\mathsf{sec}} P_{1,2} \triangleright A$ *such that* $\mathsf{tamp}(A) = s$ *and* $s \not\sqsubseteq s'$, *then* $\vdash_{\mathsf{sec}} P_1 \cong^{\pi}_{s'} P_2 \triangleright A$.

We have so far developed two methods for proving Proposition 3.11. One is based on the secrecy-sensitive bisimilarity discussed in [67] which sheds a new light on the semantic aspects of the present secrecy analysis enlarging typability. The proof based on this method is outlined in [67] for the secrecy typing in the linear $\pi$-calculus. Another method is based on an inductive analysis of causal chains of actions in secure processes, and offers a basic insight on the operational structure ensured by the presented secrecy typing and its extensions. The proof based on this method is developed in detail in a separate appendix [31].

The following is an immediate corollary of Proposition 3.11.

**Corollary 3.12** *Let* $\vdash_{\mathsf{sec}} P_{1,2} \triangleright A$ *such that* $\mathsf{tamp}(A) \not\sqsubseteq s$. *Let* $C[\,\cdot\,]$ *be a context with its hole typed* $A$ *and its result typed* $B$. *Then* $\vdash_{\mathsf{sec}} C[P_1] \cong_s C[P_2] \triangleright B$.

### 3.4 Refinement (1): Subtyping

The rest of the section discusses two refinements of the basic secrecy analysis. While these refinements are in some way or other reducible to the basic analysis, not only are they useful in practice but also they shed new light on the secrecy analysis in $\pi^{\mathsf{LA}}$ and its extensions itself. Those readers whose main interests lie in applications may safely skip the rest of the section, referring back to it as needed.

The first refinement uses a basic subtyping relation for secrecy [1; 27].

$$\frac{p \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}, \uparrow_{\mathrm{L}}\} \quad \tau_i \leq \tau_i'}{(\vec{\tau})^p \leq (\vec{\tau}')^p} \qquad \frac{\tau_i \leq \tau_i' \quad s \sqsubseteq s'}{(\vec{\tau})^{\uparrow_{\mathrm{A}}}_{s} \leq (\vec{\tau}')^{\uparrow_{\mathrm{A}}}_{s'}} \qquad \frac{p \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}\} \quad \tau_{ij} \leq \tau_{ij}'}{[\oplus_i \vec{\tau}_i]^p \leq [\oplus_i \vec{\tau}_i']^p} \qquad \frac{p \in \{\uparrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}\} \quad \tau_{ij} \leq \tau_{ij}' \quad s \sqsubseteq s'}{[\oplus_i \vec{\tau}_i]^p_s \leq [\oplus_i \vec{\tau}_i']^p_{s'}} \qquad \frac{\overline{\tau_{\mathrm{I}}'} \leq \overline{\tau_{\mathrm{I}}}}{\tau_{\mathrm{I}} \leq \tau_{\mathrm{I}}'}$$

For $\updownarrow$ we set $\updownarrow \leq \updownarrow$. We call this relation, as well as its pointwise extension to action types denoted $A \leq A'$, *secrecy subtyping*. The type-based secrecy analysis now incorporates the standard subsumption rule.

$$(\mathsf{Subs})\ \frac{\vdash_{\mathsf{sec}} P \triangleright A \quad A \leq B}{\vdash_{\mathsf{sec}} P \triangleright B} \tag{2}$$

The resulting system satisfies the subject reduction (the proof follows [27]). A simple example which shows the added typability is the following derivation.

$$(\mathsf{Subs})\ \frac{(\mathsf{Out})\ \dfrac{-}{\vdash_{\mathsf{sec}} \overline{x}\langle y\rangle \triangleright x : ((\,)^{\downarrow_{\mathrm{A}}}_{\bot})^{?_{\mathrm{A}}} \otimes y : (\,)^{\uparrow_{\mathrm{A}}}_{\bot}}}{\vdash_{\mathsf{sec}} \overline{x}\langle y\rangle \triangleright x : ((\,)^{\downarrow_{\mathrm{A}}}_{\bot})^{?_{\mathrm{A}}} \otimes y : (\,)^{\uparrow_{\mathrm{A}}}_{\top}}$$

To see the resulting typed process is in fact secure, we observe that $\overline{x}\langle y\rangle$ behaves precisely as $\overline{x}(u)u.\overline{y}$ up to the contextual equality in $\pi^{\mathsf{LA}}$.

The safety of the refined analysis is established via its reduction to the basic analysis, which is done by filling the gap between the original level and the raised level using secure copy-cats. As an example, let us simulate the subsumption above using this method.

$$\textsf{(Par, Res)}\ \frac{\vdash_{\mathsf{sec}} \overline{x}\langle u\rangle \triangleright x:(()^{\downarrow_{\mathrm{A}}}_{\bot})^{?_{\mathrm{A}}} \otimes u:()^{\uparrow_{\mathrm{A}}}_{\bot} \quad \vdash_{\mathsf{sec}} u.\overline{y} \triangleright u:()^{\downarrow_{\mathrm{A}}}_{\bot} \otimes y:()^{\uparrow_{\mathrm{A}}}_{\bot}}{\vdash_{\mathsf{sec}} \overline{x}(u)u.\overline{y} \triangleright x:(()^{\downarrow_{\mathrm{A}}}_{\bot})^{?_{\mathrm{A}}} \otimes y:()^{\uparrow_{\mathrm{A}}}_{\top}}$$

Here $\vdash_{\mathsf{sec}} u.\overline{y} \triangleright u:()^{\downarrow_{\mathrm{A}}}_{\bot} \otimes y:()^{\uparrow_{\mathrm{A}}}_{\bot}$ acts as a bridge between two types. This transformation can be performed for arbitrary types using generalised copy-cats, written $[x \to y]^{\tau}$, which are defined in Appendix B. The following property is the key to the soundness of such transformation.

**Proposition 3.13** (copycat) *Let $\tau, \tau'$ be input types such that $\tau \leq \tau'$. Then $\vdash_{\mathsf{sec}} [x \to x']^{\tau} \triangleright x:\tau \diamond x':\overline{\tau'}$ where $\diamond = \to$ if $\mathsf{md}(\tau) \in \{!_{\mathrm{L}}, \downarrow_{\mathrm{L}}\}$, and $\diamond = \otimes$ if else.*

By replacing each application of (Subs) by this transformation in a given derivation, we can always reduce the derivation in the subtyped analysis into the equivalent one in the basic analysis. By noting $(\boldsymbol{\nu}\, x)([x \to y]^{\tau}|P) \cong_s P\{y/x\}$ whenever $\vdash_{\mathsf{sec}} P \triangleright A$ such that $A(y) = \tau$ for any $s$, we know any typed process derived using (Subs) is equivalent to a process derived without using it, leading to the same noninterference property of the extended set of typed processes. We also observe that this transformation also indicates that subsumption is in fact redundant for processes which are inferred using (Bout) instead of (Out). To be precise:

**Definition 3.14** A *direct free output in $P$* is $y_i \in \{\vec{y}\}$ in a subterm $\overline{x}\langle\vec{y}\rangle$ of $P$ such that either (1) $y_i$ occurs free in $P$, or (2) $y_i$ is bound by some input in $P$.

**Proposition 3.15** (subsumption in bound output) *If no direct free output occurs in $P$, then* (Subs) *is admissible in the basic secrecy analysis.*

### 3.5 Refinement (2): Inflation

Another refinement of a different nature, suggested by the dependency core calculus [3] discussed in the next section, allows local violation of secure flow while retaining global secrecy. This refinement is used in the embedding of DCC and its call-by-value version in Section 4 (but *not* in the embedding of the Smith-Volpano calculus in Section 7). As a motivation, consider the following process (annotated by secrecy levels, writing $H$ and $L$ for $\top$ and $\bot$).

$$Q \stackrel{\text{def}}{=} \overline{y}(ab)(!a(c).\overline{z}(c')c'^{H}[\&_i.\overline{c}^{L}\mathtt{in}_i] \mid b^{H}.\overline{e}^{H}\mathtt{in}_3)$$

Now let $B \stackrel{\text{def}}{=} y:(\mathbb{N}^{\circ}_{L}\overline{\mathbb{N}^{\bullet}_{H}})^{?_{\mathrm{L}}} \otimes z:\overline{\mathbb{N}^{\circ}_{H}} \otimes e:\mathbb{N}^{\bullet}_{H}$, under which $Q$ is untypable because a high-level input $c'$ suppresses 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.

The refined analysis allows such a process to be well-typed using a simple operation called *inflation*. $\tau \sqcup s$ inflates each secrecy level in $\tau$ by taking the join,

i.e. $(\vec{\tau})^{\uparrow_{\mathrm{A}}}_{s'} \sqcup s \stackrel{\text{def}}{=} (\vec{\tau} \sqcup s)^{\uparrow_{\mathrm{A}}}_{s' \sqcup s}$, $[\&_i \vec{\tau}_i]^p_{s'} \sqcup s \stackrel{\text{def}}{=} [\&_i \vec{\tau}_i \sqcup s]^p_{s' \sqcup s}$ and $[\oplus_i \vec{\tau}_i]^p_{s'} \sqcup s \stackrel{\text{def}}{=} [\oplus_i \vec{\tau}_i \sqcup s]^p_{s' \sqcup s}$ (with $\vec{\tau} \sqcup s$ standing for $\tau_1 \sqcup s .. \tau_n \sqcup s$ with $\vec{\tau} = \tau_1 .. \tau_n$). The operation is pointwise extended to action types, written $A \sqcup s$. Noting $(A \sqcup s) \sqcup s' = A \sqcup (s \sqcup s')$, we know $A \sqcup s$ is idempotent, associative and compatible with $\odot$ (i.e. $A \asymp B$ implies $(A \sqcup s) \asymp (B \sqcup s)$ and $(A \sqcup s) \odot (B \sqcup s) = (A \odot B) \sqcup s$). These properties give us:

**Proposition 3.16** *If* $\vdash_{\text{sec}} P \triangleright A$ *then* $\vdash_{\text{sec}} P \triangleright A \sqcup s$ *for each* $s$.

The extension of the analysis is done by incorporating the converse of Proposition 3.16 in a limited form.

**Definition 3.17** We say $\vdash_{\text{sec}} P \triangleright A$ *is well-typed with inflation* if it is typable with the secrecy typing in §3.3 using (BOut) instead of (Out) augmented with:

$$(\mathsf{Inf})\ \frac{\vdash_{\text{sec}} P \triangleright \mathsf{inf}(A)}{\vdash_{\text{sec}} P \triangleright A} \qquad (\mathsf{Str})\ \frac{\vdash_{\text{sec}} P \triangleright A \quad P \equiv Q}{\vdash_{\text{sec}} Q \triangleright A}$$

where we define $\mathsf{inf}(A)$ by: $\mathsf{inf}(A) \stackrel{\text{def}}{=} A \sqcup \mathsf{tamp}(A)$.

The use of (BOut) is to have subject reduction: as discussed in the previous subsection, this does not sacrifice the expressive power. Intuitively $\mathsf{inf}(A)$ inflates the secrecy level which may contribute to the final effect up to $\mathsf{tamp}(A)$, and those which are not to some level higher than $A$. In practice the use of $\equiv$ is often avoided. As a simple example of the use of (Inf), we show the derivation of $Q$ under $B$ above.

$$(\mathsf{Inf})\ \frac{(\ldots)\ \dfrac{\cdots}{\vdash_{\text{sec}} \overline{y}(ab)(!a(c).\overline{z}(c')c'[\&_i \overline{c}\mathtt{in}_i] | b(c).\overline{c}\mathtt{in}_3) \triangleright y : (\mathbb{N}^{\circ}_H\, \overline{\mathbb{N}^{\bullet}_H})^{?_{\mathrm{L}}} \otimes z : \overline{\mathbb{N}^{\circ}_H} \otimes e : \mathbb{N}^{\bullet}_H,}}{\vdash_{\text{sec}} \overline{y}(ab)(!a(c).\overline{z}(c')c'[\&_i \overline{c}\mathtt{in}_i] | b(c).\overline{c}\mathtt{in}_3) \triangleright y : \overline{(\overline{\mathbb{N}^{\circ}_L}\, \mathbb{N}^{\bullet}_H)^{!_{\mathrm{L}}}} \otimes z : \overline{\mathbb{N}^{\circ}_H} \otimes e : \mathbb{N}^{\bullet}_H,}$$

where the omitted part is by the basic analysis.

The subject reduction for the extended analysis is proved in Appendix C.2. For noninterference, we first define $\cong_s$ in precisely the same way as Definition 3.8, except we use processes typed with inflation this time. Then we have the same noninterference result as before. For the proof, see [31].

**Proposition 3.18** (noninterference with inflation) *If* $\vdash_{\text{sec}} P_{1,2} \triangleright A$ *is well-typed with inflation such that* $\mathsf{tamp}(A) = s$ *and* $s \not\sqsubseteq s'$, *then so is* $\vdash_{\text{sec}} P_1 \cong^{\pi}_{s'} P_2 \triangleright A$.

The basic analysis (possibly combined with subtyping) would suffice in most applications: however the extended analysis gives a deep insight on the nature of the present secrecy analysis and its possible extensions.

## 4. SECRECY IN PURE FUNCTIONS

This section applies the secrecy analysis for $\pi^{\mathsf{LA}}$ presented in the previous section to the secrecy analysis for functional calculi. The first analysis is for call-by-name functions, re-justifying the secrecy of dependency core calculus by Abadi and others via $\pi^{\mathsf{LA}}$. Next we attempt a secrecy analysis for call-by-value functions, developing a secrecy typing for PCFv-like syntax. This is done by reflecting the secrecy typing in $\pi^{\mathsf{LA}}$ via the standard process encoding of call-by-functions [42; 29].

### 4.1 Dependency Core Calculus

The dependency core calculus [3] (DCC) is interesting in the present context at least in two ways. First it is one of the significant examples of a functional meta-language for type-based information flow analysis. Second it crucially relies on pointed types to combine total function types and partial ones [32; 46]. After outlining DCC in this subsection, we show a faithful embedding of DCC in $\pi^{\mathsf{LA}}$, leading to a new proof of its non-interference.

We use a slightly different, but semantically equivalent, presentation of DCC based on implicit typing. This is to allow a simpler presentation of the embedding. In particular the lifting associated with secrecy is used implicitly. We consider the system without products for simplicity: their incorporation poses no technical difficulty, and is outlined later.

The set of DCC-types are given by the following grammar. We use the same lattice $\mathcal{L}$ of secrecy levels, ranged over by $s, s', \ldots$.

$$T \quad ::= \quad \mathtt{unit}_s \;\mid\; T_1 +_s T_2 \;\mid\; T_1 \Rightarrow T_2 \;\mid\; \llcorner T \lrcorner_s \;\mid\; (T)_s$$

Unit, sums and function types should be familiar. The lifted type $\llcorner T \lrcorner_s$ is the so-called pointed type [32; 46], which indicates potential divergence. The level $s$ in $(T)_s$ indicates a secrecy level which protects a datum (in [3], this is denoted as $\mathrm{T}_s(T)$). We consider types modulo the following equations (which is justifiable via isomorphisms in the denotational universe in [3]).

$$(\mathtt{unit}_s)_{s'} = (\mathtt{unit}_{s \sqcup s'}), \quad (T_1 +_s T_2)_{s'} = T_1 +_{s \sqcup s'} T_2,$$
$$(T_1 \Rightarrow T_2)_s = T_1 \Rightarrow (T_2)_s, \quad (\llcorner T \lrcorner_s)_{s'} = \llcorner T \lrcorner_{s \sqcup s'} \quad \text{and} \quad ((T)_s)_{s'} = (T)_{s \sqcup s'}.$$

These equations essentially say that, if you protect (i.e. raise the secrecy level of) a datum with two levels $s$ and $s'$, then it is the same thing as protecting it with their join. By reading the above equations from left to right, we can rewrite types to simpler forms. It is easy to check each type rewrites to a unique normal form which erases all expressions of form $(T)_s$. This normal form has the following shape: $T_1 \Rightarrow (T_2 \Rightarrow (\ldots (T_{n-1} \Rightarrow \gamma) \ldots)$ $(n \geq 1)$ where $\gamma$ is given by the following grammar:

$$\gamma \quad ::= \quad \mathtt{unit}_s \;\mid\; T_1 +_s T_2 \;\mid\; \llcorner T \lrcorner_s$$

where $T, T_{1,2}$ is also a normal form. We write $[T_1 T_2 ... T_{n-1} \gamma]$ for a normal form $T_1 \Rightarrow (T_2 \Rightarrow (\ldots (T_{n-1} \Rightarrow \gamma) \ldots)$. Using normal forms, we introduce two key ideas in the DCC-types.

- The *protection level of* $T$, denoted $\mathsf{protect}(T)$, is given by: $\mathsf{protect}(\mathtt{unit}_s) = \mathsf{protect}(T_1 +_s T_2) = \mathsf{protect}(\llcorner T \lrcorner_s) = s$ and $\mathsf{protect}([T_1 ... T_n \gamma]) = \mathsf{protect}(\gamma)$.
- $T$ is *pointed* if $T = [T_1 .... T_n \llcorner T' \lrcorner_s]$ $(n \geq 0)$.

**Proposition 4.1**

*(1) $T$ is pointed in [3] iff there exists $T_1, .., T_n, T'$ and $s$ such that $T = [T_1 .... T_n \llcorner T' \lrcorner_s]$.*

*(2) $T$ is protected at $s$ in [3] iff $s \sqsubseteq \mathsf{protect}(T)$.*

PROOF. By mechanical structural induction. □

$$[Var]\quad \Gamma, x{:}T \vdash x : (T)_s \qquad [Unit]\quad \Gamma \vdash () : \mathtt{unit}_s$$

$$[Lam]\quad \frac{\Gamma, x{:}T \vdash M : T'}{\Gamma \vdash \lambda x.M : T \Rightarrow T'} \qquad [App]\quad \frac{\Gamma \vdash M : T \Rightarrow T' \quad \Gamma \vdash N : (T)_s}{\Gamma \vdash MN : T'}\ s \sqsubseteq \mathsf{protect}(T')$$

$$[Inl]\quad \frac{\Gamma \vdash M : T_1}{\Gamma \vdash \mathtt{inl}(M) : T_1 +_s T_2} \qquad [Case]\quad \frac{\Gamma \vdash M : T_1 +_s T_2 \quad \Gamma, x_i : T_i \vdash M_i : T'}{\Gamma \vdash \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\} : T'}\ s \sqsubseteq \mathsf{protect}(T')$$

$$[UnitM]\quad \frac{\Gamma \vdash M : T}{\Gamma \vdash M : (T)_s} \qquad [BindM]\quad \frac{\Gamma \vdash N : (T)_s \quad \Gamma, x{:}T \vdash M : T'}{\Gamma \vdash \mathtt{bind}\ x = N\ \mathtt{in}\ M : T'}\ s \sqsubseteq \mathsf{protect}(T')$$

$$[Lift]\quad \frac{\Gamma \vdash M : T}{\Gamma \vdash \mathtt{lift}(M) : \llcorner T \lrcorner_s} \qquad [Seq]\quad \frac{\Gamma \vdash N : \llcorner T \lrcorner_s \quad \Gamma, x{:}T \vdash M : T'}{\Gamma \vdash \mathtt{seq}\ x = N\ \mathtt{in}\ M : T'}\ s \sqsubseteq \mathsf{protect}(T')\ \ T' \text{ pointed}$$

$$[Rec]\quad \frac{\Gamma, x{:}T \vdash M : T}{\Gamma \vdash \mu x.M : T}\ T \text{ pointed}$$

Fig. 6. Dependency Core Calculus

The sequent of DCC has the form $\Gamma \vdash M : T$ where $M$ is an untyped $\lambda$-term with units, sums, recursion, lifting and two let-like constructs, bind and seq, while $\Gamma$ is a *base*, which is a finite map from variables to types. The reduction relation $\longrightarrow$ is generated from the following rules together with closure under all contexts.

$$\begin{array}{rcl}
(\lambda x.M)N & \longrightarrow & M\{N/x\} \\
\mathtt{case}\ \mathtt{in}_1(N)\ \mathtt{of}\ \{\mathtt{in}_i(y_i).M_i\}_{i=1,2} & \longrightarrow & M_1\{N/y_1\} \\
\mathtt{seq}\, x = \mathtt{lift}(N)\, \mathtt{in}\, M & \longrightarrow & M\{N/x\} \\
\mathtt{bind}\, x = N\, \mathtt{in}\, M & \longrightarrow & M\{N/x\} \\
M\{\mu x.M/x\} \longrightarrow N & \Rightarrow & \mu x.M \longrightarrow N
\end{array}$$

The typing rules are given in Figure 6, which are essentially the Curry (implicitly typed) version of the original rules given in [3]. Apart from the implicit typing, these rules are slightly generalised without changing semantics, so that the typability is closed under reduction. See Appendix D.1 for illustration of the typing rules, including the difference from the original presentation.

Simple examples of DCC terms follow. Below let $\mathbb{B}_s \stackrel{\text{def}}{=} () +_s ()$ and write $H$ and $L$ for $\top$ and $\bot$ for readability.

(1) $\lambda x.x : \mathbb{B}_L \Rightarrow \mathbb{B}_H$ is well-typed. This is a function which outputs a low-level datum as a high-level datum, which is surely safe.

(2) $\lambda x.\mathtt{in}_1(()) : \mathbb{B}_H \Rightarrow \mathbb{B}_L$ is well-typed. This function receives a high-level datum and returns a low-level datum: if it non-trivially uses the former it violates the secrecy, but since it does not it is safe.

(3) $y : \mathbb{B}_L \Rightarrow \mathbb{B}_H, z : \mathbb{B}_H \vdash \mathtt{bind}\ x = z\ \mathtt{in}\ yx : \mathbb{B}_H$ is well-typed. This open term shows a subtle use of bind, where a local violation of secrecy is permitted (a high-level $z$ is used as a low-level datum) while retaining safe information flow globally. Note we also have $y : \mathbb{B}_L \Rightarrow \mathbb{B}_H, z : \mathbb{B}_H \vdash (\lambda x.yx)z : \mathbb{B}_H$, indicating bind is in fact redundant in the present formulation.

We list two basic syntactic properties of DCC-terms. In (2) the generalised rules are crucial for the closure under reduction, see Remark 4.4 below.

**Proposition 4.2**

(*1*) (typability) *If* $\Gamma \vdash M : T$ *in [3] then* $\Gamma \vdash \mathsf{Erase}(M) : T$ *in the present system where* $\mathsf{Erase}(M)$ *erases all type annotations from* $M$ *including coercions* $\eta_s$.

(*2*) (subject reduction) *If* $\Gamma \vdash M : T$ *and* $M \longrightarrow M'$ *then* $\Gamma \vdash M' : T$.

PROOF. (1) is by rule induction via Prop. 4.1. (2) uses a strengthened substitution lemma, showing $\Gamma, x{:}T \vdash M : T'$ and $\Gamma \vdash N : (T)_s$ with $s \sqsubseteq \mathsf{protect}(T')$ implies $\Gamma \vdash M\{N/x\} : T'$. See Appendix D.2. □

We conclude the presentation of DCC by stipulating a Morris-like contextual congruence on DCC-terms, relativised by secrecy levels. It suffices to use the simplest pointed observable. Let $\mathbb{O}_s \stackrel{\text{def}}{=} \llcorner \mathtt{unit} \lrcorner_s$. Below $M \Downarrow$ stands for $\exists N.M \longrightarrow^* N \not\longrightarrow$.

**Definition 4.3** *Fix some* $s$. *Then* $\cong_s^{\text{DCC}}$ *is the maximum typed congruence on DCC-terms such that whenever* $\vdash M_{1,2} : \mathbb{O}_s$, *we have* $M \Downarrow$ *iff* $N \Downarrow$.

**Remarks 4.4** (subject reduction in DCC) For reference we give instances of violation of subject reduction in DCC in the original typing rules [3] (cf. [30]). We show examples in both implicitly typed and explicitly typed systems. For implicit typing, we can take $y : \mathbb{B}^L \Rightarrow \mathbb{B}^H, z : \mathbb{B}^H \vdash \mathtt{bind}\ x = z\ \mathtt{in}\ yx : \mathbb{B}^H$. Then we have $\mathtt{bind}\ x = z\ \mathtt{in}\ yx \longrightarrow yz$. However $y : \mathbb{B}^L \Rightarrow \mathbb{B}^H, z : \mathbb{B}^H \vdash yz : \mathbb{B}^H$ is *not* well-typed if we are to use the standard application rule (which does not inflate the argument type, cf. [*App*], Figure 6). For the explicitly typed system as in the original DCC, a violation comes via the coercion $\eta_l M$, which is symmetric to the issue noted above for BindM. As a simple example, we have $x{:}\mathtt{unit} \vdash \eta_\top x : (\mathtt{unit})_\top$ and $\eta_\top\ x \longrightarrow x$, but $x{:}\mathtt{unit} \not\vdash x : (\mathtt{unit})_\top$. Thus the issue arises more directly: the same remedy as we presented in Figure 6 can be used for the explicit typing.

**Remarks 4.5** (redundancy of [*UnitM*] and [*BindM*]) In the typing rules in Figure 6, [*UnitM*] and [*BindM*] are redundant in the sense that they are admissible in the system without them (regarding $\mathtt{bind}\ x = N\ \mathtt{in}\ M$ as $(\lambda x.M)N$). The permissibility of [*BindM*] is immediate. For [*UnitM*], we simultaneously establish the following two statements:

(1) If $\Gamma \vdash M : T$ then $\Gamma \vdash M : (T)_s$ for any $s$.

(2) If $\Gamma \cdot x{:}T \vdash M : T'$ with $\mathsf{protect}(T') = s'$, we have $\Gamma \cdot x{:}(T)_{s'} \vdash M : T'$.

(1) is the required statement itself: (2) is needed for establishing (1) for [*Rec*].

## 4.2 Embedding DCC

The embedding of DCC in $\pi^{\mathsf{LA}}_{\mathsf{sec}}$ is done by mapping non-pointed types to linear types and pointed ones to affine ones. The lifting $\llcorner T \lrcorner$ is replaced by a transformation from linearity to affinity. The overall scheme comes from [42; 34; 8]. First, the translation

of types is performed on their normal forms.

$$\text{(type)}\quad \mathtt{unit}_s^\bullet \stackrel{\text{def}}{=} ()_s^{\uparrow_{\mathrm{L}}} \quad (T_1 +_s T_2)^\bullet \stackrel{\text{def}}{=} [T_1^\circ \oplus T_2^\circ]_s^{\uparrow_{\mathrm{L}}} \quad \llcorner T \lrcorner_s^\bullet \stackrel{\text{def}}{=} (T^\circ)_s^{\uparrow_{\mathrm{A}}}$$

$$[T_1 ... T_{n-1}\gamma]^\circ \stackrel{\text{def}}{=} \begin{cases} (\overline{T_1^\circ} ... \overline{T_{n-1}^\circ}\gamma^\bullet)^{!_{\mathrm{L}}} & \gamma \text{ non-pointed} \\ (\overline{T_1^\circ} ... \overline{T_{n-1}^\circ}\gamma^\bullet)^{!_{\mathrm{A}}} & \gamma \text{ pointed} \end{cases}$$

$$\text{(base)}\quad \emptyset^\circ \stackrel{\text{def}}{=} \emptyset \qquad (\Gamma \cdot x{:}T)^\circ \stackrel{\text{def}}{=} \Gamma^\circ \cdot x{:}\overline{T^\circ}$$

$$\text{(action)}\quad \langle T \rangle_u^\Gamma \stackrel{\text{def}}{=} \begin{cases} (u{:}T^\circ \to A) \otimes B & T \text{ non-pointed, } \Gamma^\circ = ?_{\mathrm{L}} A \otimes ?_{\mathrm{A}} B \\ u{:}T^\circ \otimes \Gamma^\circ & T \text{ pointed} \end{cases}$$

We observe:

**Lemma 4.6** $\mathsf{tamp}(T^\circ) = \top$ *if* $T = [T_1..T_{n-1}\mathtt{unit}_s]$. $\mathsf{tamp}(T^\circ) = \mathsf{protect}(T)$ *if else. Further* $\mathsf{tamp}(\langle T \rangle_u^\Gamma) = \mathsf{tamp}(T^\circ)$ *for an arbitrary* $T$ *and* $\Gamma$.

Proof. For the first half, note $\mathsf{tamp}([T_1...T_nT']^\circ) = \mathsf{tamp}(T')$ since $\mathsf{md}(\overline{(T_i)_s^\circ}) \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}\}$ always. Using this, we calculate: $\mathsf{tamp}([T_1..T_n\mathtt{unit}_s])^\circ = \mathsf{tamp}(\mathtt{unit}_s^\bullet) = \top$. Similarly, $\mathsf{tamp}([T_1..T_n(T_1' +_s T_2')]^\circ) = \mathsf{tamp}((T_1' +_s T_2')^\bullet) = \mathsf{tamp}([T_1'^\circ \oplus T_2'^\circ]_s^{\uparrow_{\mathrm{L}}}) = s = \mathsf{protect}(T)$. The case $[T_1..T_n\llcorner T' \lrcorner_s]$ is the same. The second half is immediate by noting types in $\Gamma^\circ$ are innocuous. □

The translation of DCC-types into process types sheds a new light on DCC in a way quite different from their original denotational interpretation [3]: $[T_1...T_{n-1}\gamma]$ is now interpreted as the abstraction of interaction which may inquire at each $T_i$, to receive a datum (at specific secrecy levels) and finally emits a datum at $\gamma$ again (at a specific secrecy level). Among others we observe:

- The equation $\mathsf{protect}([T_1...T_{n-1}\gamma]) = \mathsf{protect}(\gamma)$ is now given a clear operational understanding: the translation of $T_i$ has either $?_{\mathrm{L}}$ or $?_{\mathrm{A}}$ mode, so the tamper level of $T_i$ is irrelevant.
- $s$ in $\mathtt{unit}_s$ is ignored in the translation. $[T_1..T_n\mathtt{unit}_s]^\circ$ says that, whatever the results of interactions at $T_{1..n}$, it simply signals a unique output, hence in effect there is no flow of information.

Some examples of the encoding of channel types: $\mathbb{B}_s^\bullet = [\ \oplus\ ]_s^{\uparrow_{\mathrm{L}}}$, $\mathbb{B}_s^\circ = ([\ \oplus\ ]_s^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$, $\llcorner \mathbb{B}_s \lrcorner_L^\bullet \stackrel{\text{def}}{=} (\mathbb{B}_s^\circ)_L^{\uparrow_{\mathrm{A}}}$, and $(\mathbb{B}_L \Rightarrow \mathbb{B}_H)^\circ = (\overline{\mathbb{B}_L^\circ}\mathbb{B}_H^\bullet)^{!_{\mathrm{L}}}$. As an example of the encoding of action types, we have $\langle \mathbb{B}_H \rangle_{x:\mathbb{B}_L}^u = u{:}\mathbb{B}_H^\circ \to x{:}\overline{\mathbb{B}_L^\circ}$.

The translation of DCC-terms into processes, written $[\![\Gamma \vdash M : T]\!]_u$ (often omitting $\Gamma$ for brevity), closely follows that of types, and are given in Figure 16. The translation follows [42; 35] and does not rely on secrecy annotation of DCC-types. Some examples follow, again using $H$ and $L$ for $\top$ and $\bot$. The third example shows a usage of `bind` where (Inf) (cf. § 3.3, § 3.5) is needed to justify the well-typedness of the encoding.

**Example 4.7** (DCC translations) Below we omit secrecy levels when irrelevant.

(1) $[\![x : \mathbb{B}]\!]_u \stackrel{\text{def}}{=} !u(c).\overline{x}(c')c'[.\overline{c}\mathtt{inl} \,\&\, .\overline{c}\mathtt{inr}]$. Then $\vdash_{\mathsf{sec}} [\![x : \mathbb{B}]\!]_u \triangleright u : \mathbb{B}_L^\circ \to x : \overline{\mathbb{B}_H^\circ}$ is well-typed, since $c'$ has level $L$ while $c$ has level $H$.

(2) $[\![\lambda x.x : \mathbb{B} \Rightarrow \mathbb{B}]\!]_u \stackrel{\text{def}}{=} !u(xc).\overline{x}(c')c'[.\overline{c}\mathtt{inl} \,\&\, .\overline{c}\mathtt{inr}]$. By the similar reasoning as (1), this process is secure under $u : (\mathbb{B}_L \Rightarrow \mathbb{B}_H)^\circ$ but not in $u : (\mathbb{B}_H \Rightarrow \mathbb{B}_L)^\circ$.

(3) A DCC-term $y : \mathbb{B}_L \Rightarrow \mathbb{B}_H, z : \mathbb{B}_H \vdash \mathtt{bind}\ x = z\ \mathtt{in}\ yx : \mathbb{B}_H$ (cf. §4.1) is translated into the following process (with some optimisation for simplicity).

$$!u(a).\overline{y}(bc)(!b(e).\overline{z}(f)f^H[.\overline{e}^L\mathtt{inl} \,\&\, .\overline{e}^L\mathtt{inl}] \mid c[.\overline{a}\mathtt{inl} \,\&\, .\overline{a}\mathtt{inr}]).$$

Note $y$ has type $(\overline{\mathbb{B}^\circ_L}\mathbb{B}^\bullet_H)^{?_L}$ while $z$ has type $\overline{\mathbb{B}^\circ_H}$, so that $f$ is high while $e$ is low, making the process untypable without (Inf). Using (Inf), we can regard the type of $y$ as $(\overline{\mathbb{B}^\circ_H}\mathbb{B}^\bullet_H)^{?_L}$, making $e$ high and the process as a whole typable.

Basic properties of the embedding follow. Below in (2) we define $\Omega_u^{\mathbb{O}^\circ}$ as follows: $\Omega_u^{\mathbb{O}^\circ} \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)([u \to y]^{\mathbb{O}^\circ} | [y \to u]^{\mathbb{O}^\circ})$ where $[x \to x']^\tau$ appears in Appendix B.

**Proposition 4.8**

(*1*) (typability) *If* $\Gamma \vdash M : T$ *in DCC, then* $\vdash_{\text{sec}} [\![M]\!]_u \triangleright \langle T \rangle_u^\Gamma$ *is well-typed in the secrecy typing with inflation.*

(*2*) (adequacy) *Let* $\vdash M : \mathbb{O}_s$. *Then* $M \Downarrow$ *iff* $[\![M]\!]_u \not\cong_s^\pi \Omega_u^{\mathbb{O}_s}$.

(*3*) (soundness) $[\![M_1]\!]_u \cong_s^\pi [\![M_2]\!]_u$ *implies* $M_1 \cong_s^{\text{DCC}} M_2$.

Proof. **(1)** is straightforward induction, using (Inf) for [*App*] and [*BindM*]. For **(2)**, the standard reasoning gives $M \Downarrow$ implies $[\![M]\!]_u \twoheadrightarrow \overline{u}|R$ for some $R$, that is $[\![M]\!]_u \not\cong_s^\pi \Omega_u$. For the other direction, we can obtain the stated result for the embedding without secrecy level closely following the method in [9, § J.1]. Thus in particular $[\![M]\!]_u \not\cong_\top^\pi \Omega_u$ implies $M \Downarrow$. But by $\cong_\top^\pi \subseteq \cong_s^\pi$ this means $[\![M]\!]_u \not\cong_s^\pi \Omega_u$ implies $M \Downarrow$. **(3)** is standard, using (2). □

We are now ready to establish the non-interference of DCC-terms. The result also follows from the soundness of the denotational interpretation in [3]. The present proof method has interest in that it smoothly extends to other settings such as stateful computation, cf. § 7. Below and henceforth we only consider a substitution whose codomain are closed terms. A *closing substitution* is the one whose domain covers all free variables of a given term.

**Definition 4.9** We write $E \vdash \sigma_1 \sim_s \sigma_2$ if, for well-typed substitutions $\sigma_{1,2}$, we have $\sigma_1(x) = \sigma_2(x)$ whenever $\mathsf{protect}(E(x)) \sqsubseteq s$.

**Theorem 4.10** (non-interference) *Let* $E \vdash M : \mathbb{O}_s$. *Then for any closing* $\sigma_{1,2}$ *s.t.* $E \vdash \sigma_1 \sim_s \sigma_2$, $M\sigma_1 \Downarrow$ *iff* $M\sigma_2 \Downarrow$.

Proof. Let $\vdash N_i : T$ $(i = 1, 2)$ with $\mathsf{protect}(T) \not\sqsubseteq s$. Since $\mathsf{protect}(T) \sqsubseteq \mathsf{tamp}(T^\circ)$ and $\mathsf{tamp}(\tau) = \mathsf{tamp}(A) \sqcup s$ for each $\tau$ and $s$, we can apply Proposition 3.18 to obtain $[\![N_1]\!]_x \cong_s^\pi [\![N_2]\!]_x$ under $x : T^\circ$. Assume $x : T \vdash M : \mathbb{O}_s$ (the reasoning trivially extends to multiple variables). We now reason as follows. The first implication is by non-interference. The second implication is by the standard replication theorem

[8, Proposition 7], while the third is by Proposition 4.8 (3).

$$\begin{aligned}
[\![N_1]\!]_x \cong_s^\pi [\![N_2]\!]_x \quad &\Rightarrow (\nu x)([\![M]\!]_u | [\![N_1]\!]_x) \cong_s^\pi (\nu x)([\![M]\!]_u | [\![N_2]\!]_x) \\
&\Rightarrow [\![M\{N_1/x\}]\!]_u \cong_s^\pi [\![M\{N_2/x\}]\!]_u \\
&\Rightarrow M\{N_1/x\} \cong_s^{\mathrm{DCC}} M\{N_2/x\} \\
&\Rightarrow M\{N_1/x\} \Downarrow \text{ iff } M\{N_2/x\} \Downarrow,
\end{aligned}$$

hence done. □

**Remarks 4.11** (product) We briefly discuss the encoding of product types which are omitted from our discussion of DCC so far. The encoding follows that of [35; 16] where each component of a product type (modulo associativity and other basic equations) is given a distinct name. This necessitates a mapping from a variable of type $\gamma_1 \times .. \times \gamma_n$ to a sequence of names $x_1..x_n$. For example, a variable $x$ of type $\mathtt{unit} \times \mathtt{unit}$ is translated with the map $x \mapsto x_1 x_2$ and two names $u_1$ and $u_2$ as follows:

$$!u_1(c_1).\overline{x_1}(c_1').c_1'.\overline{c_1} \mid !u_2(c_2).\overline{x_2}(c_2').c_2'.\overline{c_2}$$

whose action type becomes $\otimes_{i=1,2} u_i : \mathtt{unit}^\circ \to x_i : \overline{\mathtt{unit}^\circ}$. With this extension, all preceding results and reasoning extends to the calculus with products.

**Remarks 4.12** ([*BindM*] and (Inf)) In Example 4.7 (3), we observed the need of (Inf) for justifying the encoding of [*BindM*] (which corrects our development in [30]). However the encoding of many significant usage of [*BindM*] in the original DCC are typable in the secrecy typing without (Inf). Examples include the generalised versions of [*Case*] and [*Seq*] used in Figure 6. It remains to be seen in which practical situations [*BindM*] becomes indispensable in a way which necessitates (Inf) for its justification.

### 4.3 Call-by-Value Dependency Core Calculus

This subsection introduces a call-by-value version of DCC. Our goal is to experiment with the effectiveness of the schema mentioned in Introduction, by developing a type-based secrecy analysis for call-by-value functional calculi by reflecting the secrecy analysis in $\pi^{\mathsf{LA}}$. The resulting call-by-value calculus is useful when we consider combination with imperative features, including concurrency. It is different from the calculus called vDCC in [3] in that it is directly based on call-by-value evaluation. To distinguish the calculus from vDCC, the calculus is called DCCv. We use PCFv-like types and syntax, which are more convenient for our later applications (we can similarly use DCC-like syntax). We first give the grammar of types, which use non-standard lifting motivated from the $\pi^{\mathsf{LA}}$-encoding.

$$\begin{array}{llllll}
\text{(common)} & T & ::= & S \mid U \mid (T)_s & & \\
\text{(total)} & S & ::= & \mathbb{N}_s \mid S \Rightarrow T & \text{(partial)} \quad U & ::= \ \llcorner S \lrcorner_s
\end{array}$$

In the grammar above, $S$ is a total type while $U$ is a partial one. We call a type of form $S \Rightarrow U$ *pointed* (note pointed types are total). Notice we only allow total types to occur at the argument position of an arrow type. This does not lose generality since we can recover two kinds of partial arrow types (cf. [15]) in the following way.

$$[Var]\ \ E, x:S \vdash x : (S)_s \qquad [Num]\ \ E \vdash n : \mathbb{N}_s$$

$$[Succ]\ \ \frac{E \vdash M : \mathbb{N}_s}{E \vdash \mathtt{succ}(M) : \mathbb{N}_s} \qquad [If]\ \ \frac{E \vdash M : \mathbb{N}_s \quad E \vdash N_i : T}{E \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : T}\ \ s \sqsubseteq \mathsf{protect}(T)$$

$$[Lam]\ \ \frac{E, x:S \vdash M : T}{E \vdash \lambda x.M : S \Rightarrow T} \qquad [App]\ \ \frac{E \vdash M : S \Rightarrow T \quad E \vdash N : (S)_s}{E \vdash MN : T}\ \ s \sqsubseteq \mathsf{protect}(T)$$

$$[Lift]\ \ \frac{E \vdash M : S}{E \vdash M : \llcorner S \lrcorner_s} \qquad [Seq]\ \ \frac{E \vdash N : \llcorner S \lrcorner_s \quad E, x:S \vdash M : U}{E \vdash \mathtt{seq}\ x = N\ \mathtt{in}\ M : U}\ \ s \sqsubseteq \mathsf{protect}(U)$$

$$[Rec]\ \ \frac{E, x:S \vdash \lambda y.M : S}{E \vdash \mu x.\lambda y.M : S}\ S \text{ pointed}$$

Fig. 7. Typing Rules of Call-by-Value DCC

- $U \rightharpoonup U' \stackrel{\text{def}}{=} S \Rightarrow U'$ (with $U \stackrel{\text{def}}{=} \llcorner S \lrcorner_s$), which is a total type. Intuitively, this type represents a closure which expects a possibly nonterminating argument and, therefore, produces a possibly nonterminating datum.
- $U \rightharpoonup_s U' \stackrel{\text{def}}{=} \llcorner U \rightharpoonup U' \lrcorner_s$, which is a partial type (this notation restores the partial function space constructor in our original presentation in [31]). This is a partial version of $\rightharpoonup$, and is convenient when we have successive partial applications.

These reductions are later illustrated through their representation in $\pi^{\mathsf{LA}}$. Equations and protection levels over types are defined as before:

$$(\mathbb{N}_s)_{s'} = (\mathbb{N}_{s \sqcup s'}), \quad (\llcorner T \lrcorner_s)_{s'} = \llcorner T \lrcorner_{s \sqcup s'}, \quad \text{and} \quad (S \Rightarrow T)_s = S \Rightarrow (T)_s.$$
$$\mathsf{protect}(\mathbb{N}_s) = \mathsf{protect}(\llcorner T \lrcorner_s) = s \quad \text{and} \quad \mathsf{protect}(S \Rightarrow T) = \mathsf{protect}(T)$$

The equations above again give normal forms which have no occurrence of types of the form $(T)_s$. For terms we use the standard PCFv syntax extended with `seq`:

$$M \ ::= \ n \mid \mathtt{succ}(M) \mid \mathtt{pred}(M) \mid x \mid \lambda x.M \mid MN \mid \mu x.\lambda y.M \mid \mathtt{seq}\ x = N\ \mathtt{in}\ M$$

(we omit `bind` since it becomes redundant in our presentation, cf. Remark 4.5). The reduction $\longrightarrow$ in DCCv is call-by-value, generated from the following rules together with closure under all contexts except $\lambda$-abstraction. $V$ stands for either a variable, a natural number or a $\lambda$-abstraction.

$$\begin{array}{rcl} \mathtt{succ}(n) & \longrightarrow & n+1 \\ \mathtt{pred}(n) & \longrightarrow & n-1 \\ (\lambda x.M)V & \longrightarrow & M\{V/x\} \\ \mathtt{seq}\, x = V\, \mathtt{in}\, M & \longrightarrow & M\{V/x\} \\ ((\lambda y.M)\{\mu x.\lambda y.M/x\})V \longrightarrow N & \Rightarrow & (\mu x.\lambda y.M)V \longrightarrow N \end{array}$$

The typing rules of DCCv are given in Figure 7, using the sequent of the form $E \vdash M : T$ where we take base $E$ to be a finite map from variables to total types. We omit the rule for $\mathtt{pred}(M)$, which is typed as $\mathtt{succ}(M)$. The main difference from the typing rules for DCC lies in its distinction between total types and partial types. It is notable all typable values can be typed by total types: in this sense total types are types for values. The restriction of bases to total types does not lose generality: in fact, via the translation of partial types to total types noted above,

we can derive the following typing rules (which assume bases with partial types) as permissible rules of the present system.

$$[LamP]\ \frac{E, x{:}U' \vdash M : U}{E \vdash \lambda x.M : U' \rightharpoonup U} \qquad [AppP]\ \frac{E \vdash M : U_1 \rightharpoonup U_2 \quad E \vdash N : U_1 \quad \mathsf{protect}(U_1) \sqsubseteq \mathsf{protect}(U_2)}{E \vdash MN : U_2}$$

For justification we encode the partial application $MN$ in $[AppP]$ into $\mathtt{seq}\ x = N\ \mathtt{in}\ Mx$, which gives us, via $[Seq]$, the stated side condition. We can further derive the introduction/elimination rules of $\rightharpoonup_s$. Including these derivations, the types and typing rules of DCCv are best illustrated through their embedding in $\pi^{\mathsf{LA}}$. Here we briefly discuss two simple DCCv-terms, focussing on partiality.

**Example 4.13** (DCCv-terms) Below we write $\mathbb{N}$ for $\mathbb{N}_\perp$ for brevity.

(1) Assume $E \vdash N : \llcorner\mathbb{N}\lrcorner_M$. Then $E \cdot y : \mathbb{N} \Rightarrow \llcorner\mathbb{N}\lrcorner_H \vdash \mathtt{seq}\ x = N\ \mathtt{in}\ yx : \llcorner\mathbb{N}\lrcorner_H$ is well-typed, with $M$ being a secrecy level between $H$ and $L$. The use of possibly diverging $N$ in $\mathtt{seq}$, to be observed at level $M$, is justified by having a high-level partial type for the whole term. The term first evaluates $N$ to, say, $V$ (if it does not diverge), then evaluates $yV$.

(2) Using $[LamP]$ and $[AppP]$ given above, as well as the notation $\rightharpoonup_s$, the sequent $\vdash \lambda x.\lambda y.xy : (\llcorner\mathbb{N}\lrcorner_L \rightharpoonup_L \llcorner\mathbb{N}\lrcorner_H) \rightharpoonup_L \llcorner\mathbb{N}\lrcorner_L \rightharpoonup_M \llcorner\mathbb{N}\lrcorner_H$ is well-typed. This term denotes a higher-order partial function, receiving two potentially diverging data and applying one to the other. The type specifies a level of observation at each termination. The initial termination may be observed at $L$. Next the result of application of the first argument may be observed at level $M$. Finally the result of the second application may be observed at $H$.

A basic property of DCCv follows.

**Proposition 4.14** (subject reduction in DCCv) *If $E \vdash M : T$ and $M \longrightarrow M'$ then $E \vdash M' : T$.*

Proof. For total types we show a strengthened (call-by-value) substitution lemma as in DCC. For partial types we prove $E, x : S \vdash M : T'$ and $E \vdash V : \llcorner S\lrcorner_s$ with $s \sqsubseteq \mathsf{protect}(T')$ implies $E \vdash M\{V/x\} : T'$. See Appendix D. □

**Remarks 4.15** (DCCv-typing) DCCv-typing in Figure 7 simplifies our presentation in [30], while maintaining its essential features. In particular, $[LamP]$ and $[AppP]$ in [30] are derivable as the lifted version of $[LamP]$ and $[AppP]$ given above, while $[Rec]$ in [30] is the same as $[Rec]$ in Figure 7 via the embedding of lifted partial arrow types to pointed types (in the present sense). The simplification comes from the analysis of the $\pi^{\mathsf{LA}}$-encoding, given in the next subsection.

**Remarks 4.16** (sums and products) For reference we briefly outline a treatment of sums and products (both using eager evaluation). In both, we only combine total types, i.e. they have the form $S_1 +_s S_2$ and $S_1 \times S_2$ (note product does not mention a secrecy level). Typing rules are given accordingly. The use of total types suffices since, for example, given $U_1 = \llcorner S_1 \lrcorner_{s_1}$ and $U_2 = \llcorner S_2 \lrcorner_{s_2}$, we can encode their product as $\llcorner S_1 \times S_2 \lrcorner_{s_1 \sqcap s_2}$.

### 4.4 Embedding DCCv: Types

As we already noted, the construction of DCCv is strongly motivated by the secrecy analysis in §3 over the standard process encoding of call-by-value functions. The embedding is also used for proving the noninterference result. We first present the encoding of types.

$$
\begin{array}{lll}
\text{(type)} & S^\bullet \stackrel{\text{def}}{=} (S^\circ)^{\uparrow_{\mathrm{L}}} & U^\bullet_s \stackrel{\text{def}}{=} (U^\circ)^{\uparrow_{\mathrm{A}}}_s \quad (s = \mathsf{protect}(U)) \\
 & \mathbb{N}^\circ_s = ([\otimes_{i\in\omega}\,]^{\uparrow_{\mathrm{L}}}_s)^{!_{\mathrm{L}}} & (S \Rightarrow T)^\circ \stackrel{\text{def}}{=} \begin{cases} (\overline{S^\circ} T^\bullet)^{!_{\mathrm{L}}} & T \text{ total} \\ (\overline{S^\circ} T^\bullet)^{!_{\mathrm{A}}} & T \text{ partial} \end{cases} \\
 & \llcorner S \lrcorner^\circ_s \stackrel{\text{def}}{=} S^\circ & \\
\text{(base)} & \emptyset^\circ \stackrel{\text{def}}{=} \emptyset \qquad (E \cdot x{:}S)^\circ \stackrel{\text{def}}{=} E^\circ \cdot x{:}\overline{S^\circ} & \\
\text{(action)} & \langle T \rangle^E_u \stackrel{\text{def}}{=} u{:}T^\bullet \otimes E^\circ &
\end{array}
$$

In the standard process encoding of call-by-value computation, interaction starts from an output [42; 29; 16]. The encoding above reflects this idea, motivating the construction of DCCv-types. Among others we observe:

(1) The encoding $T^\bullet$ indicates whether this output comes from a linear channel or from an affine channel. If the channel is affine, then this emittance itself has information, hence we should specify its secrecy level. This is $s$ in $\llcorner S \lrcorner_s$.

(2) The encoding of bases (as well as types in contravariant positions) uses the dual of $(\cdot)^\circ$, and shows why it suffices to use only total types in them in the DCCv-typing. Even if we use a partial type, say, $\llcorner S \lrcorner_s$, in a base, its translation is the same as $\overline{S^\circ}$, so that it does not differ from having just $S$.

(3) $\langle T \rangle^E_u$ represents the operational structure of call-by-value which is distinct from that of call-by-name. While, as before, the process may still inquire at the environment by $?_{\mathrm{L}}/?_{\mathrm{A}}$-actions, it directly emits (if ever) information at $u$, rather than getting invoked at it.

Simple examples of encodings follow.

**Example 4.17** (encoded DCCv types)

(1) $\mathbb{N} \Rightarrow \mathbb{N}$ and its (least-level) lifting $\llcorner\mathbb{N} \Rightarrow \mathbb{N}\lrcorner_L$ are respectively translated as $((\overline{\mathbb{N}^\circ}\mathbb{N}^\bullet)^{!_{\mathrm{L}}})^{\uparrow_{\mathrm{L}}}$ and $((\overline{\mathbb{N}^\circ}\mathbb{N}^\bullet)^{!_{\mathrm{L}}})^{\uparrow_{\mathrm{A}}}_L$. Thus the lifting in DCCv simply changes $\uparrow_{\mathrm{L}}$ to $\uparrow_{\mathrm{A}}$ and adds a mandatory secrecy level (which is essentially the canonical embedding of a total type to its partial counterpart [29]).

(2) Consider $\llcorner\mathbb{N}\lrcorner \to_L \llcorner\mathbb{N}\lrcorner \to_M \llcorner\mathbb{N}\lrcorner_H$. This lifted partial arrow type is translated as $((\overline{\mathbb{N}^\circ}((\overline{\mathbb{N}^\circ}(\mathbb{N}^\circ)^{\uparrow_{\mathrm{A}}}_H)^{!_{\mathrm{A}}})^{\uparrow_{\mathrm{A}}}_M)^{!_{\mathrm{A}}})^{\uparrow_{\mathrm{A}}}_L$, indicating the behaviour which first signals at $L$, then, if the result of the first application terminates, signals at $M$, and finally if the second application terminates, signals at $H$, emitting a natural number.

(3) $\langle \llcorner\mathbb{N}\lrcorner_H \rangle^E_u$ with $E \stackrel{\text{def}}{=} x{:}\mathbb{N}$ is encoded as $u{:}(\mathbb{N}^\circ)^{\uparrow_{\mathrm{A}}}_H \otimes x{:}\overline{\mathbb{N}^\circ}$. This type represents the behaviour which may inquire at $x$ for a natural number and may emit one at $u$. The use of $x{:}\overline{\mathbb{N}^\circ}$ in the environment indicates we already assume that (the datum corresponding to) $x$ is already in a terminated form.

The protection level and the tamper level completely match via the encoding.

**Proposition 4.18** $\mathsf{protect}(T) = \mathsf{tamp}(T^\bullet)$ *for each* $T$. *Further* $\mathsf{tamp}(\langle T\rangle_u^E) = \mathsf{tamp}(T^\bullet)$ *for each* $T$, $u$ *and* $E$.

### 4.5 Embedding DCCv: Terms and Noninterference

Figure 17 in Appendix G lists the encoding of DCCv-terms. The encoding is standard [42; 29; 16]. We discuss some of the key aspects of the encoding.

(1) $\langle V\rangle_u$ always outputs at $u$ immediately, having the form $\overline{u}\langle x\rangle$ if $V = x$, and $\overline{u}(c)P$ if else.

(2) The application $\langle MN\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle M\rangle_m \mid m(a).(\boldsymbol{\nu}\, n)(\langle N\rangle_n \mid n(b).\overline{a}\langle bu\rangle)$ first waits for $\langle M\rangle_m$ to provide the name of a function $a$ via $m$; then it waits for $N$ to output an argument $b$ via $n$. Finally it sends $b$ and the name $u$ of a final value to the function via $a$. Note $m$ and $n$ are potential points at which information may flow down if they are affine (as in the derived [*AppP*]).

(3) `seq` is encoded as $\langle \mathtt{seq}\ x = N\ \mathtt{in}\ M\rangle_u \equiv (\boldsymbol{\nu}\, n)(n(x).\langle M\rangle_u | \langle N\rangle_n)$. Here $n$ is affine, hence it can receive a non-trivial information from $\langle N\rangle_n$. Another use of affinity is in recursion which relies on the fact that $x$ in $\langle \mu x.\lambda y.M\rangle_u$ should be typed with a $?_{\mathrm{A}}$-type (by the DCCv-type of $x$ being pointed).

The typing rules of DCCv are easily justifiable by the secrecy typing in $\pi^{\mathsf{LA}}$ via the encoding. Here we only show the case of [*Seq*]. In this rule we wish to infer $E \vdash \mathtt{seq}\ x = N\ \mathtt{in}\ M : U$ from $E \vdash N : \llcorner S \lrcorner_s$ and $E \cdot x : S \vdash M : U$. Assuming the encoding of these terms are well-typed, our purpose is to make the following derivation secure.

$$(\mathsf{In}^{\downarrow_{\mathrm{A}}},\ \mathsf{Par},\ \mathsf{Res})\ \frac{\begin{array}{l}\vdash_{\mathsf{sec}} \langle N\rangle_n \triangleright n : ((S^\circ)_s^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}} \otimes E^\circ \\ \vdash_{\mathsf{sec}} \langle M\rangle_u \triangleright u : U^\bullet \otimes E^\circ \otimes x : \overline{S^\circ}\end{array}}{\vdash_{\mathsf{sec}} (\boldsymbol{\nu}\, n)(\, n(x).\langle M\rangle_u \mid \langle N\rangle_n) \triangleright u : (U^\bullet)^{!_{\mathrm{A}}} \otimes E^\circ}$$

Secrecy-wise, the only non-trivial inference is for the affine input "$n(x)$", using the secure version of $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$. The secrecy condition of this rule requires (apart from the affinity of $U^\bullet$) that $s \sqsubseteq \mathsf{tamp}(U^\bullet) = \mathsf{protect}(U)$ (cf. Proposition 4.18), reaching the side condition in [*Seq*].

Typability of other encodings can similarly be verified, so that we obtain:

**Proposition 4.19** (typability) *If* $E \vdash M : T$ *in* DCCv *then* $\vdash_{\mathsf{sec}} \langle M\rangle_u \triangleright \langle T\rangle_u^E$ *is well-typed in the secrecy analysis with inflation.*

As before, the non-interference in DCCv is proved via Proposition 4.19 and computational adequacy. The argument is identical except for the use of observables at $\llcorner \mathbb{N}_s \lrcorner$. We conclude:

**Theorem 4.20** (non-interference) *Let* $E \vdash M : \llcorner \mathbb{N}_s \lrcorner$. *Then for any closing* $\sigma_{1,2}$ *s.t.* $E \vdash \sigma_1 \sim_s \sigma_2$, $M\sigma_1 \Downarrow$ *iff* $M\sigma_2 \Downarrow$.

## 5. STATE IN LINEAR/AFFINE $\pi$-CALCULUS

### 5.1 Reference Agent

The purpose of this section is to introduce a simple extension of the $\pi^{\mathsf{LA}}$-calculus to stateful computation. A basic stateful process is an encoding of an imperative variable, which we call *reference*. Using a recursive definition (which is often more convenient for representing stateful processes than replication), we can define this agent as follows.

$$\mathsf{Ref}\langle xv\rangle \stackrel{\text{def}}{=} x[(c).(\mathsf{Ref}\langle xv\rangle|\overline{c}\langle v\rangle)\&(v'e).(\mathsf{Ref}\langle xv'\rangle|\overline{e})] \tag{3}$$

In $\mathsf{Ref}\langle xv\rangle$, $x$ is its principal channel and $v$ is (a pointer to) its stored value. This process waits for invocation with two branches at $x$, with its right branch for reading and its left 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 $e$, and uses $v'$ as its new value (thus changing its state) and acknowledge the receipt via $e$. Writing down these behaviours as reduction, we can formalise the dynamics of $\mathsf{Ref}\langle xv\rangle$ as follows.

$$\mathsf{Ref}\langle xv\rangle \mid \overline{x}\mathtt{inl}\langle c\rangle \longrightarrow \mathsf{Ref}\langle xv\rangle \mid \overline{c}\langle v\rangle \tag{4}$$

$$\mathsf{Ref}\langle xv\rangle \mid \overline{x}\mathtt{inr}\langle v'c\rangle \longrightarrow \mathsf{Ref}\langle xv'\rangle \mid \overline{c} \tag{5}$$

As is well-known, combination of state and concurrency leads to a loss of Church-Rosser property via interference [37; 38; 40], as the following process shows:

$$R \stackrel{\text{def}}{=} \mathsf{Ref}\langle x1\rangle \mid \overline{x}\mathtt{inl}(2c)c.\mathbf{0} \mid \overline{x}\mathtt{inr}(c)c(y).\overline{y}(e)e[.\overline{u}\mathtt{in}_n \,\&\, .\Omega_u] \tag{6}$$

where $\mathsf{Ref}\langle xn\rangle$ stands for $(\boldsymbol{\nu}\, v)(\mathsf{Ref}\langle xv\rangle|[\![n]\!]_v)$, and $\overline{x}\mathtt{inr}(nc)P$ for $\overline{x}\mathtt{inr}(wc)([\![n]\!]_w|P)$. By the racing condition at $x$, this agent may or may not emit at $u$, i.e. either:

$$R \longrightarrow^{+} \mathsf{Ref}\langle x1\rangle \mid \overline{x}\mathtt{inr}(2c)c.\mathbf{0} \mid \overline{u}\mathtt{in}_n \longrightarrow^{+} \mathsf{Ref}\langle x2\rangle \mid \overline{u}\mathtt{in}_n \quad \text{or} \quad R \longrightarrow^{+} \mathsf{Ref}\langle x2\rangle \mid \Omega_u.$$

Hence the write action at $x$ affects termination at $u$.

Another significant property of a reference agent is that we can represent a large class of stateful and non-deterministic behaviours by combining references and replication (for a formal related result in the context of sequential computation, see [5]). As an example, a counter agent which increments a number at each time it is invoked, can be defined from a reference and replication as follows.

$$\mathsf{Counter}\langle x\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(!x(f).\overline{y}\mathtt{inl}(c)c(n).\overline{y}\mathtt{inr}(me)([\![\mathsf{succ}\langle n\rangle]\!]_m|e.\overline{f}\langle n\rangle)|\mathsf{Ref}\langle y0\rangle) \tag{7}$$

where $[\![\mathsf{succ}\langle n\rangle]\!]_m$ is a successor of $n$ defined as $[\![\mathsf{succ}\langle n\rangle]\!]_m \stackrel{\text{def}}{=} !m(c).\overline{n}(e)e[\&_i\, .\overline{c}\mathtt{in}_{i+1}]$. This agent first reads the value $n$ stored in a local reference and write $n+1$ to it, and finally returns $n$ to port $f$.

In the light of its expressiveness as well as for the sake of a clean presentation, we incorporate state into $\pi^{\mathsf{LA}}$ by introducing a reference as a constant. The syntax of processes now becomes:

$$P ::= \ldots \mid \mathsf{Ref}\langle xy\rangle$$

The constant $\mathsf{Ref}\langle xy\rangle$ has the same reduction rules as (4,5) above. Since $\pi^{\mathsf{LA}}$ is not sequential, the incorporation of references results in nondeterminism.

$$\frac{\vdash P \triangleright \vec{y}{:}\vec{\tau} \otimes \,?A^{-x}}{\vdash !x(\vec{y}).P \triangleright x{:}(\vec{\tau})^{!_{\mathrm{M}}} \otimes A}\ (\mathsf{In}^{!_{\mathrm{M}}}) \qquad \frac{-}{\vdash \mathsf{Ref}\langle x\, y\rangle \triangleright x{:}\mathsf{ref}\langle\tau\rangle \otimes y{:}\overline{\tau}}\ (\mathsf{Ref})$$

$$\frac{-}{\vdash \overline{x}\mathtt{inl}\langle c\rangle \triangleright x{:}\overline{\mathsf{ref}\langle\tau\rangle} \otimes c{:}(\tau)^{\uparrow_{\mathrm{L}}}}\ (\mathsf{Read}) \qquad \frac{-}{\vdash \overline{x}\mathtt{inr}\langle vc\rangle \triangleright x{:}\overline{\mathsf{ref}\langle\tau\rangle} \otimes v{:}\overline{\tau} \otimes c{:}()^{\uparrow_{\mathrm{L}}}}\ (\mathsf{Write})$$

Fig. 8. Typing Stateful Actions

### 5.2 Typing Stateful Agents

**Action Modes.** In the following incorporation of stateful interaction, our main goal is to maintain the behavioural constraint of (pure) linear/affine interaction for processes typed with original $\pi^{\mathrm{LA}}$-types, while seamlessly integrating them with stateful behaviour. The following action modes are additionally used.

| | | | |
|---|---|---|---|
| $!_{\mathrm{LM}}$ | Linear stateful server | $?_{\mathrm{LM}}$ | Linear stateful client |
| $!_{\mathrm{M}}$ | Affine stateful server | $?_{\mathrm{M}}$ | Affine stateful output |

These modes are stateful versions of $!_{\mathrm{L}}$, $?_{\mathrm{L}}$, $!_{\mathrm{A}}$, and $?_{\mathrm{A}}$, respectively. $!_{\mathrm{LM}}$ and $?_{\mathrm{LM}}$ are mutually dual, while $!_{\mathrm{M}}$ and $?_{\mathrm{M}}$ are mutually dual. We add $!_{\mathrm{LM}}$ and $!_{\mathrm{M}}$ (resp. $?_{\mathrm{LM}}$ and $?_{\mathrm{M}}$) to $\mathcal{M}_!$ (resp. $\mathcal{M}_?$).

**Channel Types.** Using the added action modes, the channel types are extended by the following syntax.

$$\tau_{\mathrm{I}} \ ::= \ \ldots \ \mid \ (\vec{\tau})^{!_{\mathrm{M}}} \ \mid \ \mathsf{ref}\langle\tau\rangle \qquad \tau_{\mathrm{O}} \ ::= \ \ldots \ \mid \ (\vec{\tau})^{?_{\mathrm{M}}} \ \mid \ \overline{\mathsf{ref}\langle\tau\rangle}$$

where $\mathsf{ref}\langle\tau\rangle \stackrel{\mathrm{def}}{=} [(\tau)^{\uparrow_{\mathrm{L}}} \& \overline{\tau}()^{\uparrow_{\mathrm{L}}}]^{!_{\mathrm{LM}}}$ with $\mathsf{md}(\tau) \in \mathcal{M}_!$, which is a type of reference agents whose value has type $\tau$. For simplicity we only consider reference types for stateful linear replication and unary types for stateful affine replication, which is enough for our applications (affine stateful branching/selection types follow their unary counterpart; the incorporation of general linear stateful types are outlined in [30]). A reference has the linear behaviour since it necessarily returns an answer (resp. acknowledges) whenever it is read (resp. written). This linear nature of reference is essential for the secrecy analysis in the next section.

Using extended $\mathcal{M}_!$ and $\mathcal{M}_?$, the well-formedness of channel types are defined in the same way as before. For example, the well-formedness of $(\vec{\tau})^{!_{\mathrm{M}}}$ and $(\vec{\tau})^{?_{\mathrm{M}}}$ follows **(C3)** in Definition 2.2, so the former should carry a unique $\uparrow_{\mathrm{A}}$-channel and zero or more $?$-channels, dually for the latter. Note also $\mathsf{ref}\langle\tau\rangle$ is indeed well-formed whenever $\mathsf{md}(\tau) \in \mathcal{M}_!$.

**Typing.** The typing rules are given by Figure 8 combined with the rules in Figure 3. $?A$ now indicate $A$ may include $?_{\mathrm{LM}}/?_{\mathrm{M}}$-types. $(\mathsf{In}^{!_{\mathrm{M}}})$ is defined as in $(\mathsf{In}^{!_{\mathrm{A}}})$. In (Ref), we note $\mathsf{md}(\tau) \in \mathcal{M}_!$ by the well-formedness. (Read) is the left selection which first inquires at a reference, then receives a value; (Write) is the right selection which

writes to a reference then gets an acknowledgement. These rules can be understood in the light of the reduction rules given in (4) and (5) in § 5.1. A few examples of stateful processes follow.

**Example 5.1** (reference)

(1) (reference as a product) It is well-known that we can encode an imperative variable as a pair of functions. For example, a variable $x$ can be represented as $\langle \lambda y.!x,\ \lambda v.x := v \rangle$. We can represent this behaviour as the following process: $ProdVar_{rx} \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, w)(!r(c).\overline{x}\mathtt{inl}(c')c'(v).\overline{c}\langle v\rangle \mid !w(vc).\overline{x}\mathtt{inr}\langle vc\rangle)$. We can then type this agent as $\vdash ProdVar_{rx} \triangleright r\colon ((\tau)^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{M}}} \otimes x\colon \overline{\mathsf{ref}\langle\tau\rangle}$.

(2) (newref) In ML, Ref $M$ creates a new reference and stores $M$ in it (after evaluating $M$). To represent Ref as a process, we first decompose it into finer operations: $\lambda m.\mathtt{new}\, y \mapsto m\ \mathtt{in}\ y$ (where $\mathtt{new}\, y \mapsto V\ \mathtt{in}\ N$ creates a new reference $y$ with value $V$ in $N$). This is then encoded as $\overline{u}(c)!c(mz).\overline{z}(y)\mathsf{Ref}\langle ym\rangle$, which is well-typed under $u\colon ((\overline{\tau}(\mathsf{ref}\langle\tau\rangle)^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}})^{\uparrow_{\mathrm{L}}}$. The process first signals itself, then receives a value $v$ and a return channel $z$, and finally sends, via $z$, a pointer to a new reference with value $v$.

(3) (read) In ML, $!M$ indicates the result of evaluating $M$ into a reference label and reading from it. To represent the operation ! as a process, we again first decompose it into $\lambda m.(\mathtt{let}\ x = !m\ \mathtt{in}\ x)$, using finer operations. We can then represent this expression as $\overline{u}(c)!c(mz).\overline{m}\mathtt{inl}(c)c(x).\overline{z}\langle x\rangle$. This process has the typing $u\colon (\overline{(\mathsf{ref}\langle\tau\rangle}(\tau)^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}})^{\uparrow_{\mathrm{L}}}$, assuming the type of $x$ is $\tau$.

The stateful extension of $\pi^{\mathsf{LA}}$, written $\pi^{\mathsf{LAM}}$, satisfies the subject reduction (which is established for its secure version in the next section) and allows faithful embeddings of languages with imperative features. Using $\pi^{\mathsf{LAM}}$, the next section develops a theory of secrecy analysis for imperative, concurrent computation.

## 6. SECRECY WITH STATE

### 6.1 Secrecy Annotation on Channel Types.

We start from annotating the extended set of channel types as follows.

$$\tau_{\mathrm{I}} ::= \ldots \mid (\vec{\tau})^{!_{\mathrm{M}}}_{s} \mid \mathsf{ref}_s\langle\tau\rangle \qquad \tau_{\mathrm{O}} ::= \ldots \mid (\vec{\tau})^{?_{\mathrm{M}}}_{s} \mid \overline{\mathsf{ref}_s\langle\tau\rangle}$$

where $\mathsf{ref}_s\langle\tau\rangle$ is a reference type with secrecy level $s$: $\mathsf{ref}_s\langle\tau\rangle \stackrel{\text{def}}{=} [(\tau)^{\uparrow_{\mathrm{L}}} \& \overline{\tau}()^{\uparrow_{\mathrm{L}}}]^{!_{\mathrm{LM}}}_{s}$ (with $\mathsf{md}(\tau) \in \mathcal{M}_!$ by well-fomedness). Note a reference type can carry another reference type, noting $\mathcal{M}_!$ now includes $!_{\mathrm{LM}}$ and $!_{\mathrm{M}}$.

### 6.2 Refined Typing for Stateful Agents

**Motivation.** We use a refined typing for reference types based on a distinction between read and write modes, rather than by building the secrecy analysis directly on $\pi^{\mathsf{LAM}}$. This is needed for a precise secrecy analysis for stateful processes. We illustrate the idea with a concrete example. First consider the following imperative command (we assume $x$ is an imperative variable of a boolean type; $!x$ reads the content of $x$).

$$C \stackrel{\text{def}}{=} \mathsf{if}\ !x\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2.$$

The encoding of $[\![C]\!]_f$ would be given as follows (with $\mathbb{B}^\circ_s = ([\,\oplus\,]^{\uparrow_{\mathrm{L}}}_s)^{!_{\mathrm{L}}}$):

$$\vdash \overline{x}\mathtt{inl}(c)c(b).\overline{b}(g)g[.[\![C_1]\!]_f \& .[\![C_2]\!]_f] \triangleright x : \overline{\mathsf{ref}_s\langle \mathbb{B}^\circ_s\rangle} \otimes A \tag{8}$$

We leave the unknown part of the action type as $A$. Note the first action of this process *reads* from $x$, which does not change the state of $x$, even though $x$ has a mutable type. So the tampering level of this process at $x$ should not be recorded. For example, even if $s = \bot$, if $C_{1,2}$ tamper only at the high level, the command as a whole should be regarded as a high-level command.

Now suppose $C_1$ is $y := 1$ and $C_2$ is $y := 2$. Each command writes a natural number at a variable $y$, whose encoding is given by:

$$[\![y := n]\!]_f \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, c)(\overline{y}\mathtt{inr}\langle cf\rangle \mid [\![n]\!]_c) \tag{9}$$

where $f$ is a channel for acknowledgement. The typing of the process in (8) can now be elaborated as follows.

$$x : \overline{\mathsf{ref}_s\langle \mathbb{B}^\circ_s\rangle} \otimes y : \overline{\mathsf{ref}_{s'}\langle \mathbb{N}^\circ_{s'}\rangle} \otimes f : ()^{\uparrow_{\mathrm{L}}} \tag{10}$$

($f$ has a truly linear output since command "$y := n$" always terminates). The actions of this process at $y$ changes the state of the reference $y$ in the environment, hence the tamper level of an output at $y$ should be recorded, unlike $x$. Thus, if the level of the boolean type, $s$, is high, then for $g[.[\![C_1]\!]_f \& .[\![C_2]\!]_f]$ in (10) to be typable, $s'$ should also be high (cf. $(\mathsf{Bra}^{\downarrow_{\mathrm{L}}})$ in Figure 5), conforming to the treatment of conditionals in [58; 62].

This example suggests the distinction between reading and writing at a mutable type is fundamental for assigning an appropriate tamper level to stateful processes. The typing in (10) however does not distinguish the reading at $x$ from the writing at $y$: the write mode annotation, which we introduce next, is used to supply this very information.

**Annotated Typing.** In the refined typing, we annotate a channel typed with a $?_{\mathrm{LM}}/?_{\mathrm{M}}$-type by the write mode $\mathtt{w}$ if the channel can be used for mutating interaction, using the notation $x^{\mathtt{w}} : \tau$. Otherwise the name is not annotated. Thus $x^{\mathtt{w}} : \overline{\mathsf{ref}_s\langle\tau\rangle}$ indicates there may be a write action to $x$, while $x : \overline{\mathsf{ref}_s\langle\tau\rangle}$ indicates there can only be a read action at $x$. For uniformity we assume mutating channels are annotated with $\varepsilon$ when it has no write-mode annotation, writing $x^{\varepsilon} : \tau$, and let $\delta$ range over $\{\mathtt{w}, \varepsilon\}$. In spite of this, we often omit $\varepsilon$ from $x^{\varepsilon} : \tau$. The type assignment for $?_{\mathrm{LM}}/?_{\mathrm{M}}$-types obeys the following algebra. Let $\mathsf{md}(\tau) \in \{?_{\mathrm{LM}}, ?_{\mathrm{M}}\}$ below.

$$x^{\mathtt{w}} : \tau \odot x^{\delta} : \tau = x^{\delta} : \tau \odot x^{\mathtt{w}} : \tau = x^{\mathtt{w}} : \tau \qquad x : \tau \odot x : \tau = x : \tau$$

The composition with $\overline{\tau}$ is as before, i.e. $x^{\delta} : \tau \odot x : \overline{\tau} = x : \overline{\tau}$. The tampering level for annotated typing is now given as follows.

**Definition 6.1** (tampering) $\mathsf{tamp}(\tau)$ is defined by the same clauses as Definition 3.2, except we stipulate $\tau$ such that $\mathsf{md}(\tau) \in \{?_{\mathrm{LM}}, ?_{\mathrm{M}}\}$ is immediately tampering. $\mathsf{tamp}(A)$ is then given by:

$$\mathsf{tamp}(A) \stackrel{\text{def}}{=} \sqcap\{\,\mathsf{tamp}(\tau) \mid (x:\tau \in A \text{ and } \mathsf{md}(\tau) \notin \{?_{\mathrm{LM}}, ?_{\mathrm{M}}\}) \text{ or } x^{\mathtt{w}} : \tau \in A\}$$

**Example 6.2** (tampering level for references) $(\vec{\tau})^{?_{\mathrm{M}}}_{s}$ is immediately tampering since the opponent $!x(\vec{y}).P$ in the environment may expose free write actions from $P$ which affects the environment. Similarly $\overline{\mathsf{ref}_s\langle\tau\rangle}$ is immediately tampering. However in the definition of $\mathsf{tamp}(A)$, we do not record $s$ of $\overline{\mathsf{ref}_s\langle\tau\rangle}$ if it is not annotated with $\mathtt{w}$. Thus, for example, $\mathsf{tamp}(x^{\mathtt{w}}{:}\overline{\mathsf{ref}_s\langle\tau\rangle}) = s$, but $\mathsf{tamp}(x{:}\overline{\mathsf{ref}_s\langle\tau\rangle}) = \top$.

### 6.3 Structural Security

A key element in secrecy typing for stateful agents is an additional well-formedness condition for channel types. It reflects a different way in which information leaks in stateful computing. We first state the condition, then illustrate the idea.

**Definition 6.3** (structural security) $\tau$ is *structurally secure* if for each occurrence $\tau'$ in $\tau$ (1) $\mathsf{sec}(\tau') \sqsubseteq \mathsf{tamp}(\tau')$ when $\mathsf{md}(\tau') \in \{!_{\mathrm{LM}}, !_{\mathrm{M}}\}$, and (2) $\mathsf{sec}(\tau') \sqsubseteq \mathsf{tamp}(\overline{\tau'})$ when $\mathsf{md}(\tau') \in \{?_{\mathrm{M}}, ?_{\mathrm{LM}}\}$.

As a simple example, $\mathsf{ref}_L\langle\mathbb{B}_H\rangle$ is structurally secure while $\mathsf{ref}_H\langle\mathbb{B}_L\rangle$ is not, with $H$ and $L$ standing for $\top$ and $\bot$. The definition says a mutable type should have higher tampering levels in carried types than enclosing types. This is to prevent leakage of information, as the following example shows (the process already appeared in (6) in § 5.1: the notations $\mathsf{Ref}\langle x1\rangle$ and $\overline{x}\mathtt{in}_2(2c)P$ are also illustrated there).

$$R \stackrel{\mathrm{def}}{=} \mathsf{Ref}\langle x1\rangle \mid \overline{x}\mathtt{inl}(2c)c.\mathbf{0} \mid \overline{x}\mathtt{inr}(c)c(y).\overline{y}(e)e[.\overline{u}\mathtt{in}_n \,\&\, .\Omega_u]$$

As we already observed in § 5.1, the write action at the channel $x$ by $\overline{x}\mathtt{in}_2(2c)P$ may affect termination at a channel $u$. Now let $\mathbb{N}^{\diamond}_{s}$ stand for $([\oplus_{i\in\omega}\,]^{\uparrow_{\mathrm{A}}}_{s})^{!_{\mathrm{A}}}$. Assume $x$ has reference type $\mathsf{ref}_H\langle\mathbb{N}^{\diamond}_{L}\rangle$ violating structural security in Definition 6.3 (here $u$ should have type $\mathbb{N}^{\diamond}_{L}$ due to non-termination). Then we can observe that the high-level channel $x$ affects an action at a low-level channel $u$.

The anomaly takes place because stateful agents can transmit information using time-difference, storing what has happened in its state to transmit it later [42]. Structurally secure types prevent this leakage by requiring that a stateful replication to transmit information at the same, or higher, level than it receives. *Hereafter we assume all channel types are structurally secure.*

### 6.4 Secrecy Typing with State

The secrecy typing for stateful processes adds the additional rules in Figure 9 as well as refining the existing rules as given in Figure 10. Other rules remain the same as before (the summary of all secrecy typing rules for $\pi^{\mathsf{LAM}}$ is given in Figure 19 in Appendix, the last page). A brief illustration of these rules follow.

- $(\mathsf{In}^{!_{\mathrm{M}}})$ requires the secrecy level $s$ to be lower than the tampering level of body $P$, since it directly receives information. Note we allow the prefixing of free mutating outputs with write modes (which we never allow for stateless $!_{\mathrm{LM}}/!_{\mathrm{M}}$-replications).
- $(\mathsf{Out}^{?_{\mathrm{M}}})$ not only annotate its subject with $\mathtt{w}$, but also its object $y_i$ if it has a mutable type. Intuitively this is because a communicated name may as well be used, in the receiving process, for both reading and writing.
- $(\mathsf{Ref})$ is the same as in Figure 8 except we annotate $y$ (which points to the content of the reference) with the $\mathtt{w}$-mode when $\tau$ is mutable. This is because when $y$ is

$$(\mathsf{In}^{!_{\mathrm{M}}}) \quad \frac{s \sqsubseteq \mathsf{tamp}(A) \qquad \vdash_{\mathrm{sec}} P \triangleright \vec{y}{:}\vec{\tau} \otimes ?A^{-x}}{\vdash_{\mathrm{sec}} !x(\vec{y}).P \triangleright x{:}(\vec{\tau})^{!_{\mathrm{M}}}_{s} \otimes A}$$

$$(\mathsf{Out}^{?_{\mathrm{M}}}) \quad \frac{\text{if } \mathsf{md}(\tau_i) \in \{!_{\mathrm{LM}}, !_{\mathrm{M}}\} \text{ then } \delta_i = \mathtt{w} \text{ else } \delta_i = \varepsilon}{\vdash \overline{x}\langle\vec{y}\rangle \triangleright x^{\mathtt{w}}{:}(\vec{\tau})^{?_{\mathrm{M}}} \otimes (\odot_i\, y_i^{\delta_i}{:}\overline{\tau_i})}$$

$$(\mathsf{Ref}) \quad \frac{\text{if } \mathsf{md}(\tau) \in \{!_{\mathrm{LM}}, !_{\mathrm{M}}\} \text{ then } \delta = \mathtt{w} \text{ else } \delta = \varepsilon}{\vdash_{\mathrm{sec}} \mathsf{Ref}\langle xy\rangle \triangleright x{:}\mathsf{ref}_s\langle\tau\rangle \otimes y^{\delta}{:}\overline{\tau}}$$

$$(\mathsf{Read}) \quad \frac{\text{if } \mathsf{md}(\tau) \in \{!_{\mathrm{LM}}, !_{\mathrm{M}}\} \text{ then } \delta = \mathtt{w} \text{ else } \delta = \varepsilon}{\vdash_{\mathrm{sec}} \overline{x}\mathtt{inl}\langle c\rangle \triangleright x^{\delta}{:}\overline{\mathsf{ref}_s\langle\tau\rangle} \otimes c{:}(\tau)^{\uparrow_{\mathrm{L}}}}$$

$$(\mathsf{Write}) \quad \frac{\text{if } \mathsf{md}(\tau) \in \{!_{\mathrm{LM}}, !_{\mathrm{M}}\} \text{ then } \delta = \mathtt{w} \text{ else } \delta = \varepsilon}{\vdash_{\mathrm{sec}} \overline{x}\mathtt{inr}\langle vc\rangle \triangleright x^{\mathtt{w}}{:}\overline{\mathsf{ref}_s\langle\tau\rangle} \otimes v^{\delta}{:}\overline{\tau} \otimes c{:}()^{\uparrow_{\mathrm{L}}}}$$

$$(\mathsf{Weak}\text{-}\mathtt{w}) \quad \frac{\vdash_{\mathrm{sec}} P \triangleright A \otimes x{:}\tau \quad \mathsf{md}(\tau) \in \{?_{\mathrm{LM}}, ?_{\mathrm{M}}\}}{\vdash_{\mathrm{sec}} P \triangleright A \otimes x^{\mathtt{w}}{:}\tau}$$

Fig. 9. Secrecy Typing for Stateful Actions (added rules)

later sent to the environment, $y$ may as well be used for writing. $s$ should be lower than $\mathsf{tamp}(\tau)$ by structural security.

- In (Read), if $\mathsf{md}(\tau) \in \{!_{\mathrm{LM}}, !_{\mathrm{M}}\}$, then $x$ is given the write mode, even if $x$ is in fact "read", at least directly in this action, which is dual to (Ref) above. Note also $\mathsf{tamp}(\tau) \sqsubseteq s$ by structural security. The annotation of $v$ in (Write) is understood as (Out) above.
- (Weak-w) adds the write mode later (which is needed for subject reduction, cf. Appendix C). Note the original (Weak) rule can be used to add $?_{\mathrm{LM}}/?_{\mathrm{M}}$-types *without* the write mode, which makes sense since weakened type has no effect.
- For the refined rules, possible annotations on objects in (Out) and (Sel) are understood as in $(\mathsf{Out}^{?_{\mathrm{M}}})$. $(\mathsf{In}^{!_{\mathrm{A}}})$ and $(\mathsf{Bra}^{!_{\mathrm{A}}})$ refine the typing for stateless affine replicated inputs, exploiting the distinction between read/write modes. $?^{\varepsilon}_{\mu}A$ indicates $\mathsf{md}(A) \subset \{?_{\mathrm{LM}}, ?_{\mathrm{M}}\}$ such that no write action occurs in $A$. These rules allow

$$(\mathsf{Out}) \quad (p_{\mathrm{O}} \neq ?_{\mathrm{M}}) \qquad \frac{\text{if } \mathsf{md}(\tau_i) \in \{!_{\mathrm{LM}}, !_{\mathrm{M}}\} \text{ then } \delta_i = \mathtt{w} \text{ else } \delta_i = \varepsilon}{\vdash \overline{x}\langle\vec{y}\rangle \triangleright x{:}(\vec{\tau})^{p_{\mathrm{O}}} \otimes (\odot_i\, y_i^{\delta_i}{:}\overline{\tau_i})}$$

$$(\mathsf{Sel}) \quad (p_{\mathrm{O}} \neq ?_{\mathrm{LM}}) \qquad \frac{\text{if } \mathsf{md}(\tau_{ij}) \in \{!_{\mathrm{LM}}, !_{\mathrm{M}}\} \text{ then } \delta_j = \mathtt{w} \text{ else } \delta_j = \varepsilon}{\vdash \overline{x}\mathtt{in}_i\langle\vec{y}\rangle \triangleright x{:}[\oplus_i \vec{\tau}_i]^{p_{\mathrm{O}}} \otimes (\odot_j y_j^{\delta_j}{:}\overline{\tau_{ij}})}$$

$$(\mathsf{In}^{!_{\mathrm{A}}}) \quad \frac{\vdash_{\mathrm{sec}} P \triangleright \vec{y}{:}\vec{\tau} \otimes ?_{\mathrm{L}}?_{\mathrm{A}}?^{\varepsilon}_{\mu}A^{-x}}{\vdash_{\mathrm{sec}}! x(\vec{y}).P \triangleright x{:}(\vec{\tau})^{!_{\mathrm{A}}} \otimes A}$$

$$(\mathsf{Bra}^{!_{\mathrm{A}}}) \quad \frac{\vdash P_i \triangleright \vec{y}_i{:}\vec{\tau}_i \otimes ?_{\mathrm{L}}?_{\mathrm{A}}?^{\varepsilon}_{\mu}A^{-x}}{\vdash! x[\&_i(\vec{y}_i).P_i] \triangleright x{:}[\&_i\vec{\tau}_i]^{!_{\mathrm{A}}} \otimes A}$$

Fig. 10. Secrecy Typing for Stateful Actions (Refined Rules)

"pure" affine replicated outputs to cause a read action at a mutable channel. We note the same refinement cannot be done for $(\mathsf{In}^{!_{\mathrm{L}}})$ and $(\mathsf{Bra}^{!_{\mathrm{L}}})$ because it can lead to the violation of linearity.

**Example 6.4** (secrecy typing for state)

(1) $\vdash_{\mathsf{sec}} [\![y := n]\!]_f \triangleright y^{\mathtt{w}} : \overline{\mathsf{ref}_s\langle \mathbb{N}^{\circ}_{s'} \rangle} \otimes f : ()^{\uparrow_{\mathrm{L}}}$ if $s \sqsubseteq s'$ (cf. (9) in § 6.2).

(2) $\vdash_{\mathsf{sec}} [\![\mathtt{if}\ !x\ \mathtt{then}\ y := 1\ \mathtt{else}\ y := 2]\!]_f \triangleright x : \overline{\mathsf{ref}_s\langle \mathbb{B}^{\circ}_{s_b} \rangle} \otimes y^{\mathtt{w}} : \overline{\mathsf{ref}_{s'}\langle \mathbb{N}^{\circ}_{s_n} \rangle} \otimes f : ()^{\uparrow_{\mathrm{L}}}$ if $s \sqsubseteq s_b \sqsubseteq s' \sqsubseteq s_n$ (cf. (8) in § 6.2).

(3) Let $[\![\mathtt{let\ new}\ y \mapsto x\ \mathtt{in}\ y]\!]_m \stackrel{\text{def}}{=} \overline{m}(y)\mathsf{Ref}\langle yx \rangle$ (cf. Example 5.1 (2)). Then we have $\vdash_{\mathsf{sec}} [\![\mathtt{let\ new}\ y \mapsto x\ \mathtt{in}\ y]\!]_m \triangleright x : \overline{\tau} \otimes m : (\mathsf{ref}_s\langle \tau \rangle)^{\uparrow_{\mathrm{L}}}$ with $s \sqsubseteq \mathsf{tamp}(\tau)$. Then the process representing $\lambda x.\mathtt{let\ new}\ y \mapsto x\ \mathtt{in}\ y$ (cf. Example 5.1 (2)) is typable with type $u : ((\overline{\tau}(\mathsf{ref}_s\langle \tau \rangle)^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}})^{\uparrow_{\mathrm{L}}}$.

(4) $\vdash_{\mathsf{sec}} [\![\mathtt{let}\ x = !y\ \mathtt{in}\ x]\!]_u \triangleright u : \tau \otimes y^{\delta} : \overline{\mathsf{ref}_s\langle \tau \rangle}$ if $s \sqsubseteq \mathsf{tamp}(\tau)$ (cf. Example 5.1 (3)) where $\delta = \varepsilon$ if $\mathsf{md}(\tau) \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}\}$; else $\delta = \mathtt{w}$.

(5) $\vdash_{\mathsf{sec}} \mathsf{Counter}\langle x \rangle \triangleright x : ((\mathbb{N}^{\circ}_{s_n})^{\uparrow_{\mathrm{A}}}_{s})^{!_{\mathrm{M}}}_{s'}$ if $s' \sqsubseteq s \sqsubseteq s_n$ (cf. (7) in § 5.1).

### 6.5 Basic Properties of Secrecy Typing in $\pi^{\mathsf{LAM}}$

We start from the subject reduction theorem.

**Proposition 6.5** (subject reduction) *If* $\vdash_{\mathsf{sec}} P \triangleright A$ *and* $P \twoheadrightarrow Q$ *then* $\vdash_{\mathsf{sec}} Q \triangleright A$.

Proof. As in the proof of Proposition 3.6, we first prove the substitution lemma. Since $\mathtt{w}$ may be attached to each name with mutable type $?_{\mathrm{LM}}/?_{\mathrm{M}}$, we need to prove the following additional substitution lemma.

> Suppose $\vdash_{\mathsf{sec}} P \triangleright x^{\delta} : \tau \otimes y^{\delta'} : \tau \otimes A$ with either $\delta = \delta'$ or $\delta = \varepsilon$ and $\delta' = \mathtt{w}$. Then (a) $\vdash_{\mathsf{sec}} P\{y/x\} \triangleright y^{\delta'} : \tau \otimes A$ and (b) $\mathsf{tamp}(x^{\delta} : \tau \otimes y^{\delta'} : \tau \otimes A) = \mathsf{tamp}(y^{\delta'} : \tau \otimes A)$.

The remaining interesting case is references, which is similarly proved in [65, Proposition 3]. See Appendix C.3 for the full proof. □

There are several ways for defining a secrecy-sensitive contextual congruence for $\pi^{\mathsf{LAM}}$. If we use the same clause as given in Definition 3.8, Section 3, we obtain a version of May-equivalence. In the presence of nondeterminism, however, it is often more convenient to use those equivalences which capture branching structure, such as failure/testing equivalences and bisimulations. Here we consider the equality over stateful processes based on reduction-closure [28]. Below we say $A$ is *closed* when $\mathsf{md}(A) \subset \mathcal{M}_! \cup \{\updownarrow\}$.

**Definition 6.6** A typed congruence $\cong$ on secure processes in $\pi^{\mathsf{LAM}}$ is *reduction-closed* when $\vdash_{\mathsf{sec}} P \cong Q \triangleright A$ with $A$ closed and $P \twoheadrightarrow P'$ implies $Q \twoheadrightarrow Q'$ such that $\vdash_{\mathsf{sec}} P' \cong Q' \triangleright A$. It is *$s$-sound* if it is reduction-closed and, moreover, whenever $\vdash_{\mathsf{sec}} P \cong Q \triangleright x : ()^{\uparrow_{\mathrm{A}}}_{s}$, we have $P \Downarrow_x$ iff $Q \Downarrow_x$. The maximum $s$-sound congruence exists for each $s$, which we denote by $\cong^{\pi}_{s}$.

The restriction of this congruence to secure $\pi^{\mathsf{LA}}$-processes coincides with $\cong_s$ in Section 3, so our overloaded notation is consistent. As before, a basic tool for reasoning about $\cong_s$ in $\pi^{\mathsf{LAM}}$ is a context lemma, whose form is more complicated than before due to nondeterminism.

**Lemma 6.7** (context) *Let* $\vdash_{\mathsf{sec}} P_{1,2} \triangleright A$ *in* $\pi^{\mathsf{LAM}}$. *Then* $P_1 \cong_s P_2$ *if and only if* $P_1 \mathcal{R} P_2$ *for some* $\mathcal{R}$ *satisfying the following condition: whenever* $\vdash_{\mathsf{sec}} Q_1 \mathcal{R} Q_2 \triangleright B$,

(1) *For each* $\vdash_{\mathsf{sec}} R \triangleright C$ *s.t.* $B \asymp C$, *we have* $\vdash_{\mathsf{sec}} Q_1 | R \mathcal{R} Q_2 | R \triangleright B \odot C$
(2) *if* $Q_1 \longrightarrow Q_1'$ *with* $B$ *closed, then* $Q_2 \twoheadrightarrow Q_2'$ *s.t.* $Q_1' \mathcal{R} Q_2'$, *and the symmetric case, and*
(3) *if* $B = C \otimes x\colon()_s^{\uparrow_{\mathrm{A}}}$ *with* $C$ *closed, then* $Q_1 \Downarrow_x$ *iff* $Q_2 \Downarrow_x$.

Proof. The "only if" direction is immediate. The "if" direction is shown by closing the relation $\mathcal{R}$ under all contexts, and by showing it is sound, which is standard [36; 28]. □

As in $\pi^{\mathsf{LA}}$, an alternative characterisation of $\cong_s$ can be given by a secrecy-sensitive bisimulation, see [67]. The context lemma above can also be sharpened by restricting the form of composed type $C$, in a form close to Lemma 3.10, though the above form suffices for its use in the present paper.

Finally we extend the noninterference result to stateful processes.

**Proposition 6.8** (non-interference) *Let* $\vdash_{\mathsf{sec}} P_{1,2} \triangleright A$ *such that* $\mathsf{tamp}(A) = s$. *Then* $s \not\sqsubseteq s'$ *implies* $\vdash_{\mathsf{sec}} P_1 \cong_{s'}^{\pi} P_2 \triangleright A$.

The statement is literally the same as the noninterference theorem for stateless processes (Proposition 3.11). However the involved equivalence is different which now uses reduction-closure. The proof of Proposition 6.8 is given in [31], which uses, following the proof of noninterference of secure $\pi^{\mathsf{LA}}$-processes, an inductive causality analysis of secure stateful processes, finally appealing to the context lemma above.

### 6.6 Refinement by Subtyping and Inflation

We can consistently extend the secrecy subtyping for $\pi^{\mathsf{LA}}$ discussed in §3.4 to stateful processes, by adding the following subtyping rules (retaining all preceding subtyping rules and the subsumption rule).

$$\frac{\tau_i \leq \tau_i' \quad s \sqsubseteq s'}{(\vec{\tau})_s^{?_{\mathrm{M}}} \leq (\vec{\tau}')_{s'}^{?_{\mathrm{M}}}} \qquad \frac{s \sqsubseteq s'}{\overline{\mathsf{ref}_s\langle\tau\rangle} \leq \overline{\mathsf{ref}_{s'}\langle\tau\rangle}}$$

In the last rule, we cannot vary $\tau$ in $\mathsf{ref}_s\langle\tau\rangle \stackrel{\mathrm{def}}{=} [(\tau)^{\uparrow_{\mathrm{L}}} \& \overline{\tau}()^{\uparrow_{\mathrm{L}}}]_s^{!_{\mathrm{LM}}}$ since it occurs both as itself (covariant position) and as its dual (contravariant position), cf. [51]. As in §3.4, we can translate away the subsumption into copy-cats (see Appendix B for details). This translation reduces the subtyping to the basic analysis in $\pi^{\mathsf{LAM}}$, leading to the noninterference for the subtyped analysis.

The refinement based on inflation (cf. § 3.5) is also consistently extendible to stateful processes with bound output, using the same rule and reaching the same noninterference result. The proof of subject reduction in Appendix C.2 extends without change: the proof of noninterference is given in [31].

## 7. CONCURRENCY, REFERENCE AND PROCEDURE

In this section we use the secrecy analysis in $\pi^{\mathsf{LAM}}$ for the development of a secrecy typing for concurrent programs with general -references and procedures. The language is based on Smith-Volpano's secure multi-threaded imperative calculus. The typing rules are directly suggested from the secrecy typing in $\pi^{\mathsf{LAM}}$. Among others we establish (1) the conservativity result for Smith's recent secrecy discipline [57] when restricted to first-order references; and (2) the noninterference for the language via its reduction to the secrecy analysis in $\pi^{\mathsf{LAM}}$.

### 7.1 A Volpano-Smith Language

We briefly review the syntax and operational semantics of an imperative language we consider. Below $x, y, \ldots$ range over a countable set of *names*, used both for (function) variables and labels for reference.

$$
\begin{array}{lll}
\text{(expression)} & e & ::= 1,2,\ldots \mid x \mid \mathtt{succ}(e) \mid \mathtt{pred}(e) \\
& & \quad \mid \lambda x.e \mid (e_1)e_2 \mid c\ \mathtt{return}\ e \mid \mathtt{seq}\ x = e\ \mathtt{in}\ e' \\
\text{(value)} & v & ::= 1,2,\ldots \mid x \mid \lambda x.e \\
\text{(command)} & c & ::= \mathtt{skip} \mid x := v \mid c_1; c_2 \\
& & \quad \mid \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \\
& & \quad \mid \mathtt{while}\ !y\ \mathtt{do}\ c \\
& & \quad \mid \mathtt{let}\ x = e\ \mathtt{in}\ c \\
& & \quad \mid \mathtt{let}\ x = !y\ \mathtt{in}\ c \\
& & \quad \mid \mathtt{new}\ x \mapsto v\ \mathtt{in}\ c \\
\text{(threads)} & o & ::= \textstyle\prod_i c_i
\end{array}
$$

The syntax of commands is from [58], extended with general references, local variable declaration and higher-order procedures. We use two let commands for simpler presentation of typing rules, though we shall be sometimes informal about them, writing e.g. $x := !y$ instead of $\mathtt{let}\ z = !y\ \mathtt{in}\ x := z$. For brevity of presentation we do not include `lets` and `new` in expressions (which can be treated similarly in both dynamics and types). `seq` is used in expressions following DCC and DCCv, since it gives a simpler presentation of typing rules.

The reduction rules of commands are given in Figure 11, which uses the *configuration* of the form $(c, \sigma)_X \longrightarrow (c', \sigma')_{X'}$ where $\sigma, \sigma', \ldots$ denote *environments*, i.e. finite maps from names to values such that $X \subset \mathtt{dom}(\sigma)$ and $X' \subset \mathtt{dom}(\sigma')$. $X$ in $(c, \sigma)_X$ indicates hidden (local) references. Names in $X$ are bound occurrences in $(c, \sigma)_X$, and assume the standard $\alpha$-equality $\equiv_\alpha$ on configurations combined with other standard bindings in commands and expressions. We stipulate:

$$
\begin{array}{rcll}
(c,\ \sigma)_Y & \equiv & (c',\ \sigma')_{Y'} & ((c,\sigma)_Y \equiv_\alpha (c',\sigma')_{Y'}) \\
((\mathtt{new}\ x \mapsto v\ \mathtt{in}\ c_1); c_2,\ \sigma)_Y & \equiv & (\mathtt{new}\ x \mapsto v\ \mathtt{in}\ (c_1; c_2),\ \sigma)_Y & (x \notin \mathsf{fv}(c_2)) \\
(\mathtt{new}\ x \mapsto v\ \mathtt{in}\ c,\ \sigma)_Y & \equiv & (c,\ \sigma \cdot x \mapsto v)_{Y \cup \{x\}} & (x \notin \mathsf{fv}(\sigma))
\end{array}
$$

For expressions we assume the single-step call-by-value reduction following DCCv, which again takes the form $(e, \sigma)_X \longrightarrow (e, \sigma)_{X'}$ due to the new name creation inside

$$(x := v,\ \sigma)_X \longrightarrow (\mathtt{skip},\ \sigma[x \mapsto v])_X$$

$$(\mathtt{skip}; c,\ \sigma)_X \longrightarrow (c,\ \sigma)_X$$

$$\frac{(c_1, \sigma)_X \longrightarrow (c_1', \sigma')_{X'}}{(c_1; c_2, \sigma)_X \longrightarrow (c_1'; c_2, \sigma')_{X'}}$$

$$(\mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma)_X \longrightarrow (c_1, \sigma)_X \qquad (\sigma(v) = 0)$$

$$(\mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma)_X \longrightarrow (c_2, \sigma)_X \qquad (\sigma(v) \neq 0)$$

$$(\mathtt{while}\ !y\ \mathtt{do}\ c, \sigma)_X \longrightarrow (c; \mathtt{while}\ !y\ \mathtt{do}\ c, \sigma)_X \qquad (\sigma(y) = 0)$$

$$(\mathtt{while}\ !y\ \mathtt{do}\ c, \sigma)_X \longrightarrow (\mathtt{skip}, \sigma)_X \qquad (\sigma(y) \neq 0)$$

$$(\mathtt{let}\ x = e\ \mathtt{in}\ c, \sigma)_X \longrightarrow (\mathtt{let}\ x = e'\ \mathtt{in}\ c, \sigma')_{X'} \qquad ((e, \sigma)_X \longrightarrow (e', \sigma')_{X'})$$

$$(\mathtt{let}\ x = v\ \mathtt{in}\ c, \sigma)_X \longrightarrow (c\{v/x\},\ \sigma)_X$$

$$(\mathtt{let}\ x = !y\ \mathtt{in}\ c, \sigma)_X \longrightarrow (c\{v/y\},\ \sigma)_X \qquad (\sigma(y) = v)$$

$$\frac{(c,\ \sigma)_X \equiv (c_0,\ \sigma_0)_{X_0} \longrightarrow (c_0',\ \sigma_0')_{X_0'} \equiv (c',\ \sigma')_{X'}}{(c,\ \sigma)_X \longrightarrow (c',\ \sigma')_{X'}}$$

Fig. 11. Reduction of VS-Calculus with Reference and Procedure

$e$. We omit concrete rules except for $c\ \mathtt{return}\ e$, which are given as:

$$\frac{-}{(\mathtt{skip}\ \mathtt{return}\ e, \sigma)_X \longrightarrow (e, \sigma)_X} \qquad \frac{(c, \sigma)_X \longrightarrow (c', \sigma')_{X'}}{(c\ \mathtt{return}\ e, \sigma)_X \longrightarrow (c'\ \mathtt{return}\ e, \sigma')_{X'}}$$

The rules for other forms of expressions are just as in DCCv. Finally the reduction for threads is given as follows, assuming $X_i' \cap X_j' = \emptyset$ whenever $i \neq j$.

$$\frac{\exists i.\ (c_i,\ \sigma)_{X_i} \longrightarrow (c_i',\ \sigma')_{X_i'}}{(\prod_i c_i,\ \sigma)_{\bigcup_i X_i} \longrightarrow (\prod_i c_i',\ \sigma')_{\bigcup_i X_i'}}$$

Note if $X_i'$ and $X_j'$ are disjoint then so are $X_i$ and $X_j$ since $X_i \subset X_i'$ for each $i$ always. $(o, \sigma)_X$ is also called *configuration*. $(o, \sigma)$ stands for $(o, \sigma)_\emptyset$.

**Remarks 7.1**

(1) (choice of syntax) The presented syntax is based on distinction between commands and expressions, which would make the comparison with, and heritage from, languages by Smith and Volpano clearer. Another possible choice of syntax is to consider commands as part of expressions (as in ML), which is discussed in Section 8.

(2) (local names in configuration) As is customarily done (and as we did in the conference version), we could have omitted the use of local labels in operational

semantics by choosing fresh names appropriately. We choose the present formulation since it allows us to treat transition more rigorously. The local labels are also in direct correspondence with the process encoding of configurations.

### 7.2 Secrecy with Reference and Procedure

We first illustrate the subtlety in secrecy with local references and procedure by examples. *For brevity we assume $u, v, w$ are low-level variables while $x, y, z$ are high-level variables in the following examples.* In § 7.7, we shall see how secure or insecure property of each of the following commands is exactly analysed via its encoding into the $\pi$-calculus.

**Local references.** Local references give abstraction, while aliasing may break this abstraction. As an example, let $u$ be a low-level reference to a natural number and consider the following command.

$$c_1 \stackrel{\text{def}}{=} \texttt{new}\ u \mapsto 0\ \texttt{in}\ u := !v; x := !u;$$

Here the locality raises abstraction, hiding the low-level writing at $u$: only the writing at $x$ is visible. Thus in effect $c_1$ only writes at the high-level. Now consider the following:

$$c_2 \stackrel{\text{def}}{=} \texttt{new}\ w \mapsto v\ \texttt{in}\ (w := u; \texttt{let}\ w' = !w\ \texttt{in}\ w' := 3).$$

The command writes at $w$ and $w'$, which are both local; however in fact it is writing at $u$, which is free. Thus $c_2$ tampers at the low-level.

**Imperative procedures.** DCC and DCCv capture non-trivial features of secrecy in pure higher-order functions. With imperative features, procedures add different kinds of subtlety.

- (Divergence) Let $e_1 \stackrel{\text{def}}{=} \lambda y.(!x)y$ and $e_2 \stackrel{\text{def}}{=} \lambda y.y$. Consider

  $$c_3 \stackrel{\text{def}}{=} u{:=}1; (\texttt{if}\ z\ \texttt{then}\ x{:=}e_1\ \texttt{else}\ x{:=}e_2); z' := (!x)0; u{:=}0$$

  Then $c_3$ reveals $z$ at $u$ by diverging when $z = \texttt{true}$.
- (Side effects) Take $e_3 \stackrel{\text{def}}{=} \lambda x.\ u := x\ \texttt{return}\ 0$. Then

  $$c_4 \stackrel{\text{def}}{=} \texttt{if}\ z\ \texttt{then let}\ y = (e_3)0\ \texttt{in skip}.$$

  leaks information at $u$, though $e_3$ is secure as a function. If we use $e_4 \stackrel{\text{def}}{=} \lambda x.\ \texttt{skip return}\ !u$ instead of $e_3$ then $c_4$ becomes secure, since it only *reads* from $u$.
- (Aliasing) Given $e_5 \stackrel{\text{def}}{=} \lambda u.\ !!u := 1\ \texttt{return}\ 0$, Then

  $$c_5 \stackrel{\text{def}}{=} \texttt{if}\, z\, \texttt{then new}\, v \mapsto w\, \texttt{in let}\, x = (e_5)v\, \texttt{in skip}$$

  is not secure since $w$ can be aliased. However if we further hide $w$, the command becomes secure.

The aim of the proposed typing system is to detect any possible danger involving aliasing and side-effects, while type-checking pure functions generously.

### 7.3 Types

The syntax of types for commands and expressions follows. This is an extension of call-by-value DCC (§ 4.3) adding a general reference type and mutable arrow types. A base is a finite function from annotated variables to total value types, where an annotation is either $\mathtt{w}$ or $\varepsilon$. The symbol $\varepsilon$ in $x^{\varepsilon}:T$ is usually omitted. As will be stipulated later, the $\mathtt{w}$-annotation is used only when a name is assigned to a reference type or a mutable arrow type, and indicates the possibility of writing to a reference, directly or indirectly. Following DCCv (cf. § 4.3), we use $S$ for total types and $U$ for partial types.

$$
\begin{array}{lll}
\text{(value)} & T & ::= S \mid U \\
 & S & ::= \mathbb{N}_s \mid \mathtt{ref}_s(S) \mid S \Rightarrow T \mid S \stackrel{s}{\Rightarrow} U \\
 & U & ::= \llcorner S \lrcorner_s \\
\text{(base)} & E & ::= \emptyset \mid E \cdot x{:}S \mid E \cdot x^{\mathtt{w}}{:}S \\
\text{(command)} & \rho & ::= \mathrm{cmd}\,\tau_s \qquad (\tau \in \{\Downarrow, \Uparrow\})
\end{array}
$$

$\mathtt{ref}_s(S)$ is a reference type of value typed by $S$, which may receive a writing as side effects at the level $s$. We have the following kinds of arrow types:

- $S \Rightarrow S'$ is a pure function space from a total type to a total type.
- $S \Rightarrow U$ is a pure function space from a total type to a partial type.
- $S \stackrel{s}{\Rightarrow} U$ is an impure function space from a total type to a partial type with side effect at the level $s$ (as $s$ in $\mathtt{ref}_s(S)$).

The restriction of argument types to total types is as in DCCv, and does not lead to a loss of expressiveness: following §4.3, we have a pure partial function space $U \rightharpoonup U'$ and an impure partial function space $U \stackrel{s}{\rightharpoonup} U$ as derived constructions (which are interpreted in terms of $\Rightarrow$ and $\stackrel{s}{\Rightarrow}$ as before). In an impure arrow type $S \stackrel{s}{\Rightarrow} U$, we do not allow the resulting type to be total (i.e. we only allow pointed types to be impure). This is for the sake of simplicity. While non-pointed impure arrow types can be added, their typing rules become complicated due to the necessity to preserve totality (see [30]).

The following distinction between pure and impure types is important in the type discipline given later.

**Definition 7.2** *$T$ is* mutable *if it is of form either $\mathtt{ref}_s(S)$ or $S \stackrel{s}{\Rightarrow} U$.*

Using this definition, we assume $S$ in $E \cdot x^{\mathtt{w}}{:}S$ in the grammar of bases, is always mutable.

In $\mathrm{cmd}\,\tau_s$, $\tau = \Downarrow$ (resp. $\tau = \Uparrow$) indicates convergence (resp. potential divergence), while $s$ is a lower bound at which the termination may be observed (or, as Smith [57] puts it, at which level of variables a termination depends upon).

We use the subtyping on value and command types, which come from [58; 27; 57], as well as the secrecy subtyping for the $\pi^{\mathsf{LAM}}$-calculus discussed in §6.6. First for value types, we set:

$$
\frac{s \sqsubseteq s'}{\mathbb{N}_s \leq \mathbb{N}_{s'}} \qquad \frac{S' \leq S \quad T \leq T'}{S \Rightarrow T \leq S' \Rightarrow T'} \qquad \frac{s' \sqsubseteq s}{\mathtt{ref}_s(S) \leq \mathtt{ref}_{s'}(S)}
$$

$$\frac{S' \leq S \quad U \leq U' \quad s' \sqsubseteq s}{S \overset{s}{\Rightarrow} U \leq S' \overset{s'}{\Rightarrow} U'} \qquad \frac{S \leq S' \quad s \sqsubseteq s'}{\llcorner S \lrcorner_s \leq \llcorner S' \lrcorner_{s'}}$$

Note $T$ does not vary in $\mathtt{ref}_s(T)$ (see illustration in Proposition 7.7).

$$\frac{-}{\mathrm{cmd} \Downarrow_s \leq \mathrm{cmd} \Downarrow_{s'}} \qquad \frac{-}{\mathrm{cmd} \Downarrow_s \leq \mathrm{cmd} \Uparrow_s} \qquad \frac{s \sqsubseteq s'}{\mathrm{cmd} \Uparrow_s \leq \mathrm{cmd} \Uparrow_{s'}}$$

Note secrecy levels are irrelevant in converging commands. For this reason we sometimes omit $s$ from $\Downarrow_s$ without loss of precision. The subtyping on expression types and command types will be later illustrated in their relationship to the secrecy subtyping in $\pi^{\mathsf{LAM}}$.

### 7.4 Information Level and Safety

As in DCC and DCCv, we define the protection level of each value type $T$, denoted by $\mathsf{protect}(T)$, which indicates the level of information which $T$ embodies. One difference from DCC and DCCv is that protection levels should be assigned not only to value types but also to their duals, which we call *environment types*. To motivate their introduction, we start from stateless interactions in a DCCv-term $E \vdash M : T$. In its $\pi$-calculus translation, this becomes $\vdash_{\mathsf{sec}} \langle M \rangle_u \triangleright u : T^\bullet \otimes E^\circ$. The tampering level of this behaviour is calculated from the action type $T^\bullet \otimes E^\circ$. Since all channel type in $E^\circ$ has mode $?_{\mathrm{L}}$ or $?_{\mathrm{A}}$, we can completely neglect $E^\circ$ from the calculation, and consider only $T^\bullet$.

The situation is however quite different in the present imperative setting. In both $E \vdash c : \rho$ and $E \vdash e : T$, the command/expression viewed as a process interacts at $E$ at mutable channels so that the levels of these actions should also be taken into account. Incorporating these dual levels are also essential when formalising what corresponds to structural security in the present context. The definition follows.

(1)
- $\mathsf{protect}(\mathbb{N}_s) = \mathsf{protect}(\llcorner S \lrcorner_s) = s$
- $\mathsf{protect}(S \Rightarrow T) = \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(T)$
- $\mathsf{protect}(S \overset{s}{\Rightarrow} U) = \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(U)$
- $\mathsf{protect}(\mathtt{ref}_s(S)) = \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(S)$.

(2)
- $\mathsf{protect}^{\mathcal{E}}(\mathbb{N}_s) = \mathsf{protect}^{\mathcal{E}}(S \Rightarrow T) = \top$
- $\mathsf{protect}^{\mathcal{E}}(S \overset{s}{\Rightarrow} U) = \mathsf{protect}^{\mathcal{E}}(\mathtt{ref}_s(T)) = s$

The illustration of the definition is best given after we present its embedding into secure process types, where, among others, we show the above definition precisely corresponds to the tamper level in $\pi^{\mathsf{LAM}}$ (cf. Proposition 7.8). Using protection levels, we can now introduce a basic condition on value types, which plays a key rôle for harnessing aliases.

**Definition 7.3** (safety) The set of *safe types* are generated by:

- $\mathbb{N}_s$ is safe for any $s$.
- If $S$ and $T$ are safe then $S \Rightarrow T$ is safe.
- If $S$ is safe and $s \sqsubseteq \mathsf{protect}(\mathtt{ref}_s(S))$ then $\mathtt{ref}_s(S)$ is safe.
- If $S$ and $U$ are safe and $s \sqsubseteq \mathsf{protect}(S \overset{s}{\Rightarrow} U)$ then $S \overset{s}{\Rightarrow} U$ is safe.
- If $S$ is safe then $\llcorner S \lrcorner_s$ is safe for any $s$.

The condition is directly suggested by structually security of the $\pi^{\mathsf{LAM}}$-calculus (Definition 6.3), see Proposition 7.9 later. In essence, it says that, as a command unfolds a sequence of references, the secrecy level either remains the same or gets higher. As an example, take the following program:

$$\texttt{let}\ z = !x\ \texttt{in}\ \texttt{let}\ w = !z\ \texttt{in}\ \texttt{if}\ w\ \texttt{then}\ u := 0\ \texttt{else}\ u := 1$$

The safety condition statically ensures that $w$ is higher than $x$ and $z$. We observe:

- For writing, a program should be prohibited from writing at a low-level as the result of reading a high-level information, so the constraint makes sense.
- For reading, making the secrecy level of a local resource lower than its container would not be meaningful since the reader has already cleared a higher level of security (in particular if a low-level datum is pointed from a high-level reference by a destructive update, one may safely upgrade the level of this datum to the high-level).

From these observations we claim that the constraint is reasonable in practice, at least for basic programming. *We hereafter assume all types are safe.*

### 7.5 Secrecy Typing

We use the following three kinds of sequents, which are derived from the associated typing rules.

| | | |
|---|---|---|
| (expression) | $E \vdash e : T$ | (rules given in Figure 12) |
| (command) | $E \vdash c : \mathrm{cmd}\,\tau_s$ | (rules given in Figure 13) |
| (thread) | $E \vdash o : \mathrm{cmd}\,\tau_s$ | (the rule given in Figure 13) |

Further Figure 14 gives weakening and subsumption rules common to expressions and commands (letting $t$ range over their union, and $\alpha$ over the union of value/command types). In the rules we use the following notations (each definition coincides with the corresponding notion in the $\pi^{\mathsf{LAM}}$-calculus via the encoding presented later, cf. Proposition 7.8).

(1) $E^{-x}$ indicates $x \notin \mathtt{dom}(E)$.

(2) $E_1 \asymp E_2$ indicates that, if $x^{\sigma} : \tau \in E_1$ and $x^{\sigma'} : \tau' \in E_2$, then $\tau = \tau'$. Then $E_1 \odot E_2$ is defined by taking their union so that $x^{\mathtt{w}} : \tau$ and $x^{\varepsilon} : \tau$ are composed to become $x^{\mathtt{w}} : \tau$.

(3) $\mathsf{tamp}(E)$ (cf. Definition 3.2) is defined as:

$$\mathsf{tamp}(E) \stackrel{\text{def}}{=} \sqcap\{\mathsf{protect}^{\mathcal{E}}(S) \mid x^{\mathtt{w}} : S \in E \text{ or } (x : S \in E \text{ and } S \text{ immutable})\}$$

As we shall show later, the system is a conservative extension of [57] (neglecting protect [12; 27]). Below we illustrate the typing rules, concentrating on those points which are new in the present system. One of the key aspects is the use of write modes at mutable types for capturing the level of writing, which is crucial for controlling aliasing effects. In the following we illustrate typing rules one by one.

$$[Var]\ \frac{S \text{ immutable}}{E \cdot x{:}S \vdash x : S} \qquad [VarM]\ \frac{S \text{ mutable}}{E \cdot x^{\Psi}{:}S \vdash x : S} \qquad [Num]\ E \vdash n : \mathbb{N}_s$$

$$[Lam]\ \frac{E \cdot x^{\delta}{:}S \vdash e : T}{E \vdash \lambda x.e : S \Rightarrow T}\ (*) \qquad [App]\ \frac{E \vdash e : S \Rightarrow T \quad E \vdash e' : S}{E \vdash ee' : T}$$

$$[LamM]\ \frac{E \cdot x^{\delta}{:}S \vdash e : U}{E \vdash \lambda x.e : S \stackrel{s}{\Rightarrow} U}\ (*) \qquad [AppM]\ \frac{E \vdash e : S \stackrel{s}{\Rightarrow} U \quad E \vdash e' : S}{E \vdash ee' : U}$$

$$[Lift]\ \frac{E \vdash e : S}{E \vdash e : \llcorner S \lrcorner_s} \qquad [Seq]\ \frac{E'^{-x} \vdash e' : \llcorner S \lrcorner_s \quad E \cdot x{:}S \vdash e : U \quad E' \asymp E}{E' \odot E \vdash \mathtt{seq}\ x = e'\ \mathtt{in}\ e : U}\ (\dagger)$$

$$[Ret]\ \frac{E \vdash c : \mathrm{cmd} \Downarrow_s \quad E \vdash e : T}{E \vdash c\ \mathtt{return}\ e : T} \qquad [RetP]\ \frac{E' \vdash c : \mathrm{cmd} \Uparrow_s \quad E \vdash e : U \quad E' \asymp E}{E' \odot E \vdash c\ \mathtt{return}\ e : U}\ (\star)$$

$(*)$ For each $x^{\delta}{:}S \in E$, (1) $S$ is immutable if $T$ is total; and (2) $\delta = \varepsilon$ if $T$ is partial.
$(\star)$ $s \sqsubseteq \mathsf{tamp}(E)$.
$(\dagger)$ $s \sqsubseteq \mathsf{protect}(U) \sqcap \mathsf{tamp}(E)$.

Fig. 12. Typing Rules for the Extended VS-Calculus: expressions

**(Expressions)**

- *Var, VarM, Num.* *[VarM]* records the mutable variable in $E$. This is because such $x$ can be used both for reading and writing. Constants such as `succ` and `pred` are typed just as in DCCv.
- *Lam, LamM.* The condition $(*)$ in *[Lam]* prohibits, for ensuring pure functional behaviour: (1) accessing free variables of mutable types for ensuring totality, and (2) doing mutable actions. In *[LamM]*, this constraint does not exist; on the other hand, we require the constraint on secrecy levels, which says that invoking a mutable abstraction should not affect write actions in its body lower than the receiving level. This has a clear process-based interpretation, see Proposition 7.10.
- *App, AppM.* Neither *[App]* nor *[AppM]* mention secrecy levels since they assume the arguments always terminate. As in DCCv, we can derive the following partial version of *[AppM]* from the rules in Figure 12.

$$[AppPM]\ \frac{E \vdash e : U_1 \rightharpoonup U_2 \quad E \vdash e' : U_1}{E \vdash ee' : U_2} \quad \mathsf{protect}(U_1) \sqsubseteq \mathsf{tamp}(E) \sqcap \mathsf{protect}(U_2)$$

  We omit the corresponding partial version of *[LamM]*, which is given just as *[LamM]*. *[AppPM]* is easily justifiable by regarding $ee'$ as $\mathtt{seq}\ x = e'\ \mathtt{in}\ ex$ and using *[Seq]*, noting $U_1$ can always be written as $\llcorner S \lrcorner_s$ for some $S$ and $s$.
- *Lift, Seq.* These rules are as in DCCv except *[Seq]* now respects the level of writing at the base, which should be the same as, or higher than, the level of termination of $N$, which affects the actions at the base (note having distinct bases allow us to type more terms since $\mathsf{tamp}(E_1 \odot E_2) \sqsubseteq \mathsf{tamp}(E_2)$ in general).
- *Ret, RetP.* The total higher-order procedure *[Ret]* does not need considering secrecy levels. On the other hand, if the command $c$ is partial, the termination level affects to the level of $e$; hence we need the side condition on secrecy.

$$[Skip] \qquad E \vdash \mathtt{skip} : \mathrm{cmd}\ \Downarrow$$

$$[Ass] \qquad \frac{E \cdot x^{\delta} : \mathtt{ref}_s(S) \vdash v : S}{E \cdot x^{\mathtt{w}} : \mathtt{ref}_s(S) \vdash x := v \ : \ \mathrm{cmd}\Downarrow}$$

$$[Com] \qquad \frac{E_i \vdash c_i : \mathrm{cmd}\,\tau_{s_i}\ (i = 1,2) \quad E_1 \asymp E_2}{E_1 \odot E_2 \vdash c_1; c_2 : \mathrm{cmd}\,\tau_{s_2}} \qquad \begin{array}{l}\text{if } \tau = \Uparrow \text{ then} \\ s_1 \sqsubseteq s_2 \sqcap \mathsf{tamp}(E_2)\end{array}$$

$$[If] \qquad \frac{E \vdash v : \mathbb{N}_s \quad E \vdash c_i : \mathrm{cmd}\,\tau_{s'}}{E \vdash \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \mathrm{cmd}\,\tau_{s'}} \qquad \begin{array}{l}s \sqsubseteq \mathsf{tamp}(E) \\ \text{if } \tau = \Uparrow \text{ then } s \sqsubseteq s'\end{array}$$

$$[While] \qquad \frac{E(x) = \mathtt{ref}_{s'}(\mathbb{N}_s) \quad E \vdash c : \mathrm{cmd}\ \Uparrow_{s_0}}{E \vdash \mathtt{while}\ !x\ \mathtt{then}\ c : \mathrm{cmd}\ \Uparrow_{s_0}} \qquad s \sqsubseteq s_0 \sqsubseteq \mathsf{tamp}(E)$$

$$[Let] \qquad \frac{E \vdash e : S \quad E \cdot x : S \vdash c : \mathrm{cmd}\,\tau_{s'}}{E \vdash \mathtt{let}\ x = e\ \mathtt{in}\ c : \mathrm{cmd}\,\tau_{s'}}$$

$$[LetP] \qquad \frac{E \vdash e : \llcorner S \lrcorner_s \quad E \cdot x : S \vdash c : \mathrm{cmd}\ \Uparrow_{s'}}{E \vdash \mathtt{let}\ x = e\ \mathtt{in}\ c : \mathrm{cmd}\ \Uparrow_{s'}} \qquad s \sqsubseteq s' \sqcap \mathsf{tamp}(E)$$

$$[Deref] \qquad \frac{E \cdot z^{\delta} : \mathtt{ref}_s(S) \cdot x : S \vdash c : \mathrm{cmd}\,\tau_{s_0}}{E \cdot z^{\delta_0} : \mathtt{ref}_s(S) \vdash \mathtt{let}\ x = !z\ \mathtt{in}\ c : \mathrm{cmd}\,\tau_{s_0}} \qquad \begin{array}{l}\text{if } S \text{ mutable then } \delta_0 = \mathtt{w} \\ \text{else } \delta_0 = \delta\end{array}$$

$$[New] \qquad \frac{E \vdash v : S \quad E \cdot x^{\delta} : \mathtt{ref}_s(S) \vdash c : \mathrm{cmd}\,\tau_{s_0}}{E \vdash \mathtt{new}\ x \mapsto v\ \mathtt{in}\ c : \mathrm{cmd}\,\tau_{s_0}}$$

$$[Par] \qquad \frac{E \vdash c_i : \mathrm{cmd}\,\tau_s}{E \vdash \prod_i c_i : \mathrm{cmd}\,\tau_s}$$

Fig. 13. Typing Rules for the Extended VS-Calculus: command and thread

**(Command and Thread)**

- *Skip.* `skip` terminates immediately, so it has the $\Downarrow$-type.
- *Assignment.* The rule crucially relies on the safety condition (Definition 7.3). For example, $u :=!x$ with $u$ and $x$ typed as $\mathtt{ref}_L(\mathbb{N}_L)$ and (unsafe) $\mathtt{ref}_H(\mathbb{N}_L)$, respectively, becomes typable without the safety condition, which is clearly insecure. The rule records the write mode of "$x$" in $E$ by *[VarM]*.
- *Com* and *If*. *[Com]*'s side condition is equivalent to [57], which enhances [27; 58]. If the preceding command may not terminate, the information of termination (at $s_1$) should not flow down to $c_2$'s termination ($s_2$) and tampering ($\mathsf{tamp}(E_2)$). Note allowing distinct bases in these two commands adds typability since, if we weaken $E_{1,2}$ so that they coincide, the resulting tampering level is in general lower than $\mathsf{tamp}(E_2)$. The condition does not specify the case when the preceding command does terminate, since if so no information flows down to the subsequent command. *[If]* is standard, requiring the conditional variable cannot influence later behaviour at lower levels.

$$[Sub]\ \frac{E \vdash t : \alpha \quad \alpha \leq \alpha'}{E \vdash t : \alpha'} \qquad [Weak]\ \frac{E \vdash t : \alpha}{E \cdot y{:}S \vdash t : \alpha} \qquad [Weak\text{-}\mathtt{w}]\ \frac{E \cdot x{:}S \vdash t : \alpha}{E \cdot x^{\mathtt{w}}{:}S \vdash t : \alpha}\ S \text{ mutable}$$

Fig. 14. Weakening and Subsumption

- *While.* The side condition is due to Smith [57] who enlarges the typability of the while command on the basis of [58; 27] (the condition is somewhat simplified, but is equivalent to the one given in [57], see Proposition 7.6 (2) later). We offer intuitive illustration following [57]. The information at $s$ influences whether this command terminates at $s_0$, hence $s \sqsubseteq s_0$. The condition $s_0 \sqsubseteq \mathsf{tamp}(E)$ is more subtle. Assume $!x$ is initially true. Then we unfold the while loop into $c$; $\mathtt{while}\ !x\ \mathtt{do}\ c$, which shows the termination of $c$ influences later actions of $c$ at $E$, leading to the side condition. Later we shall see these conditions are precisely what are derivable from the embedding into $\pi^{\mathsf{LAM}}$. Note that $x$ is allowed to have the $\varepsilon$-mode in $E$ because $x$ only reads a natural number (which is immutable).
- *Let, LetP.* In *[Let]*, $e$ is total, so both tampering and secrecy levels are ignored. *[LetP]* corresponds to *[Seq]*, considering the tampering of write actions of $c$ in addition.
- *Deref.* Note $z$ has $\mathtt{w}$-mode in $E$ if $z$ is mutable, even though $z$ is read in this command. To see its necessity, we consider:

$$\mathtt{let}\ x = !z\ \mathtt{in}\ x := 3. \tag{11}$$

  Here $x$ looks local, but may be aliased to a free name referred by $z$. By changing mode of $z$ (which is lower than $x$ by the safety condition), we effectively accumulate the tampering level of $z$ in the environment.
- *New.* Similar to *[Deref]*, if $v$ has a mutable type, then it is recorded with $\mathtt{w}$-mode by *[VarM]*. To understand its necessity, consider:

$$\mathtt{new}\ z \mapsto y\ \mathtt{let\ in}\ \ x = !z\ \mathtt{in}\ x := 3. \tag{12}$$

  Note $y$ should have a reference type. Hence when $z \mapsto y$ is inferred, $y$ has $\mathtt{w}$-mode in $E$, which subsumes the writing at $x$ since $x$ is higher than $y$ by safety.

**(Weakening/Subsumption)**

- *Sub, Weak, Weak-*$\mathtt{w}$. *[Sub,Weak]* are standard (in particular, when mutable types are weakened, real actions at them are non-existent, hence we annotate them with $\varepsilon$). *[Weak-*$\mathtt{w}$*]* adds the write mode to mutable types, and is necessary for subject reduction. These rules correspond to (Sub), (Weak) and (Weak-$\mathtt{w}$) in $\pi^{\mathsf{LAM}}$, respectively.

We list a few typing examples, using the expressions and commands discussed in § 7.2. Below we freely use the partial versions of abstraction and application rules.

**Example 7.4** (typing examples in the extended Smith-Volpano calculus) Commands/expressions are from § 7.2. We assume $x, y, z$ are high while $u, v, w$ are low unless otherwise stated.

(1) Let $E = u^{\mathtt{w}} : \mathtt{ref}_L(\mathbb{B}_H), x^{\mathtt{w}} : \mathtt{ref}_H(\mathbb{B}_H), y : \mathtt{ref}_L(\mathbb{B}_H)$. By *[Ass]* and *[Deref]*:

$$E \vdash x := !u \ : \ \mathrm{cmd} \Downarrow_s \quad \text{and} \quad E \vdash u := !y \ : \ \mathrm{cmd} \Downarrow_s$$

(note $x := !u$ in fact stands for $\mathtt{let}\ x' = !u\ \mathtt{in}\ x := x'$). By *[Seq]*, we have:

$$E \vdash u := !y; x := !u \ : \ \mathrm{cmd} \Downarrow_s$$

We also have $E/x \vdash 0 : \mathbb{B}_H$. Finally by *[New]*,

$$E/u \vdash \mathtt{new}\ u \mapsto 0\ \mathtt{in}\ u := !y; x := !u \ : \ \mathrm{cmd} \Downarrow_s$$

for arbitrary $s$ (we omit such $s$ from now on). The level of writing of this command is $\mathsf{protect}^{\mathcal{E}}(E(x)) = H$.

(2) Let $E = w^{\mathtt{w}} : \mathtt{ref}_L(\mathtt{ref}_L(\mathbb{B}_H)), u^{\mathtt{w}} : \mathtt{ref}_L(\mathbb{B}_H), v^{\mathtt{w}} : \mathtt{ref}_L(\mathbb{B}_H)$. Then we have

$$E \cdot w_0^{\mathtt{w}} : \mathtt{ref}_L(\mathbb{B}_H) \vdash w_0 := 3 \ : \ \mathrm{cmd} \Downarrow$$

Since $w_0$ has a mutable type, by *[Deref]*, we obtain:

$$E \vdash \mathtt{let}\ w_0 = !w\ \mathtt{in}\ w_0 := 3 \ : \ \mathrm{cmd} \Downarrow \tag{13}$$

Note $w$ should have $\mathtt{w}$-mode in $E$ to infer the above sequent. We also have, by *[VarM]*, $E \vdash u : \mathtt{ref}_L(\mathbb{B}_H)$, which implies $u$ should have $\mathtt{w}$-mode in $E$. Hence by *[Ass]* we obtain:

$$E \vdash w := u \ : \ \mathrm{cmd} \Downarrow \tag{14}$$

We now apply *[Com]* to (13) and (14). Also by *[VarM]* we have $E/w \vdash v : \mathtt{ref}_L(\mathbb{B}_H)$ (we note $v$ should have $\mathtt{w}$-mode in $E$). Finally we obtain, by applying *[New]*:

$$E \vdash \mathtt{new}\ w \mapsto v\ \mathtt{in}\ w := u; \mathtt{let}\ w_0 = !w\ \mathtt{in}\ w_0 := 3 \ : \ \mathrm{cmd} \Downarrow$$

The tampering level is $\mathsf{protect}^{\mathcal{E}}(E(u)) \sqcup \mathsf{protect}^{\mathcal{E}}(E(v)) = L$.

(3) Recall $c_3 \stackrel{\text{def}}{=} u := 1; (\mathtt{if}\ z\ \mathtt{then}\ x := e_1\ \mathtt{else}\ x := e_2); z' := (!x)0; u := 0$, with $e_1 \stackrel{\text{def}}{=} \lambda y.(!x)y$ and $e_2 \stackrel{\text{def}}{=} \lambda y.y$ (note $z' := (!x)0$ stands for $\mathtt{let}\ w = !x\ \mathtt{in}\ \mathtt{let}\ k = w0\ \mathtt{in}\ z' := k$). To analyse this command, we first note that since $e_1$ contains a mutable variable $x$, it is only typable by *[LamMP]* (which is equivalent to, via decomposition into $\mathtt{seq}\ x = e\ \mathtt{in}\ e'$, the combination of *[Seq]* and *[LamM]*). Secondly, to type $\mathtt{if}$-command, $x$ should have type $\mathtt{ref}_H(T)$. By the safety condition, this means $T$ has type $T = \llcorner S \lrcorner_H$. Hence by the side condition of *[LetP]*, $\mathtt{let}\ k = w0\ \mathtt{in}\ z' := k$ has type cmd $\Uparrow_H$, so that $z' := (!x)0$ has type cmd $\Uparrow_H$. To compose $z' := (!x)0$ and $u := 0$, we need to satisfy the side condition of *[Seq]*, that is $H \sqsubseteq \mathsf{tamp}(E)$, which is impossible because we have $\mathsf{tamp}(E) = \mathsf{protect}^{\mathcal{E}}(E(u)) = L$. Hence $c_3$ is untypable.

(4) Let $e_3 \stackrel{\text{def}}{=} \lambda x.\ u := x\ \mathtt{return}\ 0$ and $E = z : \mathbb{B}_H, u : \mathtt{ref}_L(\mathbb{N}_H)$. We can check $e_3$ is typable as $E \vdash e_3 \ : \ \llcorner \mathbb{N}_L \stackrel{H}{\Rightarrow} \mathbb{N}_{H \lrcorner H}$. We now analyse the command $c_4 \stackrel{\text{def}}{=} \mathtt{if}\ z\ \mathtt{then}\ \mathtt{let}\ y = (e_3)0\ \mathtt{in}\ \mathtt{skip}$. We observe that the tampering level of the internal $\mathtt{let}$-command is $\mathsf{tamp}(E) = L$. Thus the side condition for the $\mathtt{if}$-command, $H \sqsubseteq \mathsf{tamp}(E)$, is not satisfied, so $c_4$ as a whole is untypable. Next

let $c_4' \stackrel{\text{def}}{=} \texttt{if}\ z\ \texttt{then let}\ y = (e_4)0\ \texttt{in skip}$, with $e_4 \stackrel{\text{def}}{=} \lambda x.\ \texttt{skip return}\ !u$. Then we have:

$$E \vdash \texttt{let}\ u' = !u\ \texttt{in let}\ y = (e_4)0\ \texttt{in skip} : \mathrm{cmd} \Downarrow$$

(note $u$ is not recorded by *[Deref]* because $\mathbb{N}_H$ immutable). Hence $c_4'$ is typable because $H \sqsubseteq \mathsf{tamp}(E) = H$.

(5) Similarly we can check $c_5$ of § 7.2 is untypable. This is because the write mode of $w$ is recorded by *[New]* and its tampering level is $L$, which violates the condition for `if`-command. However if we change $c_5$ into:

$$\texttt{if}\, z\, \texttt{then new}\, w \mapsto 1\, \texttt{in new}\, v \mapsto w\, \texttt{in let}\, x = (e_5)v\, \texttt{in skip}$$

Then the command is typable since the body of the `if`-command is a high-level (by the lack of free variable).

Basic syntactic properties of the secrecy typing follow, after a definition. In Proposition 7.6 (2) below, the *possibilistic Smith-calculus* is the calculus by Smith [57] which neglects (Protect) (i.e. we ignore the part of the system in [57] which "counts execution steps"). Since the calculus by Smith does not use annotation on bases, we write $|E|$ for the base which erases the write-mode annotation from $E$.

**Definition 7.5** *Let $E \vdash c : \rho$. Then we say $c$ is* first-order under $E$ *iff:* (1) *each type in $\mathsf{cod}(E)$ has shape either $\mathsf{ref}_s\langle \mathbb{N}_s \rangle$ or $\mathbb{N}_s$; and* (2) *$c$ is typed under $E$ using none of* (i) *[New] in Figure 13 and* (ii) *the rules in Figure 12 except for [Var], [VarM] and [Num].*

**Proposition 7.6**

*(1)* (subject reduction) *Let $E \vdash c : \rho$ and $E \vdash \sigma$. Then $(c, \sigma)_X \longrightarrow (c', \sigma')_{X'}$ implies:* (i) *if $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma)$ then $E \vdash c' : \rho$ and $E \vdash \rho'$ ; and* (ii) *if $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) \uplus \{x'\}$ then $E \cdot x' : T' \vdash c : \rho$ and $E \cdot x' : T' \vdash \sigma'$ for some $T'$.*

*(2)* (conservativity) *Assume $\mathcal{L} \stackrel{def}{=} \{\bot, \top\}$ and $c$ is first-order under $E$ in the above sense. Then* (i) *if $E \vdash c : \mathtt{cmd} \Uparrow_{s'}$ in the present calculus, then $|E| \vdash c : s\ \mathtt{cmd} \Uparrow_{s'}$ with $s = \mathsf{tamp}(E)$ in the possibilistic Smith calculus; and* (ii) *if $E \vdash c : s\ \mathtt{cmd} \Uparrow_{s'}$ in the possibilistic Smith calculus then $E' \vdash c : \mathtt{cmd} \Uparrow_s$ in the present calculus such that $E = |E'|$ and $s \sqsubseteq \mathsf{tamp}(E')$.*

Proof. For (1) we use the both-way correspondence in typability between the explicit versions of secure $\pi^{\mathsf{LAM}}$ and the extended Smith-Volpano, as well as with their implicit versions. For (2), we interpret the sequent $E \vdash c : s\ \mathtt{cmd}\, s'$ in [57] as $E' \vdash c : \mathtt{cmd} \Uparrow_{s'}$, with $E'$ an annotated version of $E$ so that $\mathsf{tamp}(E')$ essentially has the level $s$ or higher, and $E \vdash c : s\, \mathtt{cmd}\, m$ (with $m$ a natural number) as $E' \vdash c : \mathtt{cmd} \Downarrow$. The only non-trivial points are the correspondence between the tampering level in [57] and $\mathsf{tamp}(E)$, as well as the side conditions in `while`-rules. See Appendix E. □

### 7.6 Embedding

We first show the embedding of types. Except for the mutable types, the embedding of value types is the same as DCCv.

$$
\begin{array}{lll}
\text{(value)} & S^\bullet \stackrel{\text{def}}{=} (S^\circ)^{\uparrow_{\mathrm{L}}} & U^\bullet \stackrel{\text{def}}{=} (U^\circ)^{\uparrow_{\mathrm{A}}}_s \quad (\mathsf{protect}(U) = s) \\
& \mathbb{N}^\circ_s = ([\otimes_{i\in\omega}\,]^{\uparrow_{\mathrm{L}}}_s)^{!_{\mathrm{L}}} & (S \Rightarrow T)^\circ \stackrel{\text{def}}{=} (\overline{S^\circ} T^\bullet)^{!_{\mathrm{L}}} \\
& \llcorner S \lrcorner^\circ_s \stackrel{\text{def}}{=} S^\circ & \\
& \mathtt{ref}_s(S)^\circ \stackrel{\text{def}}{=} \mathsf{ref}_s\langle S^\circ\rangle & (S \stackrel{s}{\Rightarrow} U)^\circ \stackrel{\text{def}}{=} (\overline{S^\circ} U^\bullet)^{!_{\mathrm{M}}}_s \\
\text{(base)} & (\emptyset)^\circ = \emptyset \qquad (E \cdot x^\delta : S)^\circ = E^\circ \cdot x^\delta : \overline{S^\circ} & \\
\text{(action)} & \Downarrow^\bullet_s \stackrel{\text{def}}{=} ()^{\uparrow_{\mathrm{L}}} \qquad \Downarrow^\bullet_s \stackrel{\text{def}}{=} ()^{\uparrow_{\mathrm{A}}}_s & \\
& \langle \mathrm{cmd}\,\tau_s\rangle^E_u \stackrel{\text{def}}{=} u : \tau^\bullet_s \otimes E^\circ \quad \langle S\rangle^E_u \stackrel{\text{def}}{=} u : S^\bullet \otimes E^\circ &
\end{array}
$$

The subtyping in command types for converging commands, cf. § 7.3, is now given a clear account: the termination channel has a unary $\uparrow_{\mathrm{L}}$-type, so its level is insignificant. Similarly, the invariance in subtyping of reference types is elucidated by observing the content type now occurs both covariantly and contravariantly [5]. In fact the subtyping on value types in § 7.3 precisely corresponds to the secrecy subtyping in $\pi^{\mathsf{LAM}}$-types.

**Proposition 7.7** (subtyping) $T_1 \leq T_1$ *iff* $T_1^\circ \leq T_2^\circ$ *iff* $T_1^\bullet \leq T_2^\bullet$.

Proof. See Appendix F.1. □

The protection levels in the VS-calculus and the tampering levels in the $\pi^{\mathsf{LAM}}$-calculus coincide via encoding. For the proof see Appendix F.2.

**Proposition 7.8** (protection levels and tampering levels)

(*1*) $\mathsf{tamp}(S^\bullet) = \mathsf{tamp}(S^\circ) = \mathsf{tamp}(\llcorner S \lrcorner^\circ_s)$.
(*2*) $\mathsf{protect}(T) = \mathsf{tamp}(T^\bullet)$ *and* $\mathsf{protect}^{\mathcal{E}}(S) = \mathsf{tamp}(\overline{S^\circ})$
(*3*) $\mathsf{tamp}(E) = \mathsf{tamp}(E^\circ)$

Using (1) above, we can prove the coincidence between safety and structurally security on mutable types.

**Proposition 7.9** (safety and structurally security) *$T$ is safe iff $T^\circ$ is structurally secure iff $T^\bullet$ is structurally secure.*

Proof. By induction on the size of types. We only prove if $T$ is safe then $T^\circ$ is safe. The only interesting case is either $T = \mathtt{ref}_s(S)$ or $T = S \stackrel{s}{\Rightarrow} U$. Suppose $\mathtt{ref}_s(S)$ is safe. Then by definition and Proposition 7.8 (1), we have $\mathsf{sec}(\mathtt{ref}_s(S)^\circ) = s \sqsubseteq \mathsf{protect}(\mathtt{ref}_s(S)) = \mathsf{tamp}(\mathtt{ref}_s(S)^\circ)$, as required. The case $T = S \stackrel{s}{\Rightarrow} U$ is just similar. □

The encoding of commands and expressions is given in Figure 18 in Appendix. Expressions use call-by-value encoding [42; 29]. `while` is translated using tail recursion. By Propositions 7.7, 7.8 and 7.9, we can easily verify:

**Proposition 7.10** (syntactic soundness)

(1) (expression) $E \vdash e : T$ *implies* $\vdash_{\mathsf{sec}} [\![e]\!]_u^E \triangleright \langle T \rangle_u^E$
(2) (command) $E \vdash c : \rho$ *implies* $\vdash_{\mathsf{sec}} [\![c]\!]_u^E \triangleright \langle \rho \rangle_u^E$
(3) (thread) $E \vdash o : \rho$ *implies* $\vdash_{\mathsf{sec}} [\![o]\!]_u^E \triangleright \langle \rho \rangle_u^E$

PROOF. See Appendix F.3 (the while command is treated in §7.7). □

### 7.7 Analysis of Imperative Secrecy via Embedding

The embedding offers an in-depth analysis of imperative secrecy via the fine-grained representation as name passing processes, both in types and operations. In the following we explore this point using the examples from § 7.2 and § 7.5, as well as presenting a derivation of the conditions for while loop due to Smith [57].

(1) (`if`) `if` $v$ `then` $c_1$ `else` $c_2$ is encoded as, assuming $v$ is a boolean for simplicity:

$$(\boldsymbol{\nu}\, x)(\langle v \rangle_x \mid x(c).\overline{c}(z)z[.[\![c_1]\!]_e \& .[\![c_2]\!]_e]).$$

Hence the secrecy level of $z$ (i.e. $s$ of $\mathbb{B}_s$) should be lower than the tampering level of $[\![c_1]\!]_e$ and $[\![c_2]\!]_e$ by the side condition of $(\mathsf{Bra}^{\downarrow_{\mathrm{L}}})$. If either $c_1$ or $c_2$ may not terminate, then $e$ should have affine type $()_{s'}^{\uparrow_{\mathrm{A}}}$. Hence the tampering level of $[\![c_i]\!]_e$ is the tampering level of mutable names which appeared in $[\![c_i]\!]_e$, and if $c_i$ non-terminated, the level $s'$. This essentially corresponds to the side condition of *[Seq]* of the VS-calculus given in Figure 13.

(2) (`deref`) First we show the encoding of command `let` $w_0 = !w$ `in` $w_0 := 3$ from (11) in Example 7.4, § 7.5 (cf. Example 6.4 (1)).

$$\vdash_{\mathsf{sec}} \overline{w}\mathtt{inl}(e)e(w_0).\overline{w_0}\mathtt{inr}\langle 3 f\rangle \triangleright w^{\mathrm{w}} : \overline{\mathsf{ref}_L\langle \mathsf{ref}_L\langle \mathbb{B}_H\rangle\rangle} \otimes f : ()^{\uparrow_{\mathrm{L}}}$$

Since $f$ (the linear unary output) has no tampering level, the tampering level of the above command is that of $w$, i.e. $\mathsf{protect}^{\mathcal{E}}(\mathsf{ref}_L\langle \mathsf{ref}_L\langle \mathbb{B}_H\rangle\rangle) = L$.

(3) (`new`) A process representation of `new` $w \mapsto v$ `let in` $w_0 = !w$ `in` $w_0 := 3$ appeared in (12) in § 7.5 is typed as follows.

$$\vdash_{\mathsf{sec}} (\boldsymbol{\nu}\, w)(\mathsf{Ref}\langle wv\rangle \mid [\![\mathtt{let}\ w_0 = !w\ \mathtt{in}\ w_0 := 3]\!]_f) \triangleright f : ()^{\uparrow_{\mathrm{L}}} \otimes v^{\mathrm{w}} : \overline{\mathsf{ref}_L\langle \mathbb{B}_H\rangle}$$

In the above encoding, we can observe that $w_0$ is assigned $v$; hence the tampering level of the above command is that of $v$, i.e. $L$. Note also if we set the type of $v$ to be $\overline{\mathsf{ref}_H\langle \mathbb{B}_L\rangle}$, then this command is untypable (and in fact unsafe), due to a violation of structurally security. Similarly we can analyse the correspondence between tampering level of the commands $c_1$ and $c_2$ which appeared in § 7.2 and that of the encoding.

(4) (`let` and the sequential composition) We analyse the untypability of $c_3$ in § 7.2 by its encoding. The middle command $z' := (!x)0$ ($\stackrel{\text{def}}{=}$ `let` $w = !x$ `in let` $k = w0$ `in` $z' := k$) is encoded as follows.

$$(\boldsymbol{\nu}\, c)(\overline{x}\mathtt{inl}(e)e(w).\overline{w}\langle 0c\rangle \mid c(k).\overline{z}\mathtt{inr}(kc')c'.\overline{f})$$

First because of `if`-statement, $x$ should have the high reference type. Hence $w$ and $k$ should have $H$ by structurally security. Also the reply at $c$ may not terminate, hence for $c(k).\overline{z}\mathtt{inr}(kc')c'.\overline{f}$ to be typable, by $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$, $z$'s level should

be $H$ and moreover $\overline{f}$ has a type $()^{\uparrow_{\mathsf{A}}}_{H}$. Now, the sequential composition is given as:

$$(\boldsymbol{\nu}\, f)([\![z := (!x)0]\!]_f \mid f.[\![u := 0]\!]_{f'})$$

To type $f.[\![u := 0]\!]_{f'}$, by $(\mathsf{In}^{\downarrow_{\mathsf{A}}})$, it is necessary for $H$ to be lower than the tampering level of $[\![u := 0]\!]'_f$, but we have $\mathsf{tamp}(u^{\mathsf{w}} : \overline{\mathsf{ref}_L\langle \mathbb{B}_s\rangle}) = L$, hence the command is untypable. Similarly we can observe the untypability of $c_4$ and the typability of $c'_4$ via the tampering level of their encodings.

(5) (`while`) The side condition for the typability of while commands is due to Smith [57]. We show this condition is precisely what we derive from the security of the encoding. The encoding of `while` $!x$ `do` $c$ reads:

$$(\boldsymbol{\nu}\, e)(\overline{e}\langle u\rangle \mid !f(k).\overline{e}\langle k\rangle \mid !e(k).\overline{x}\mathtt{inl}(c)c(y).\overline{y}(z)z[(\boldsymbol{\nu}\, l)([\![c]\!]_l \mid l.\overline{f}\langle k\rangle)\&\overline{k}]).$$

Note $e$ and $f$ have mode $!_{\mathsf{M}}$ because $[\![c]\!]_l$ may as well have (free) mutable outputs. Since $e$ and $f$ suppresses each other, they should have the same level, say $s'$. Assuming that $x$ stores a natural number of level $s$, that $l$ has level $s_0$ and that $c$ tampers under $E$, we can annotate the above process with secrecy levels as follows. Below $s'_0$ is the level of $u$, i.e. the termination level of the command.

$$(\boldsymbol{\nu}\, ef)(\overline{e}\langle u\rangle \mid !f^{s'}(k).\overline{e^{s'}}\langle k^{s'_0}\rangle \mid !e^{s'}(k).\overline{x}\mathtt{inl}(c)c(y).\overline{y}(z)z^{s}[(\boldsymbol{\nu}\, l)([\![c]\!]_l \mid l^{s_0}.\overline{f}^{s'}\langle k\rangle)\&\overline{k}^{s'_0}])$$

From this we can immediately derive the following conditions:

(a) By $(\mathsf{In}^{!_{\mathsf{M}}})$ at the $e$-replications, we require $s' \sqsubseteq \mathsf{tamp}(E)$. Further by structural security (for the types of $f$ and $e$) we require $s' \sqsubseteq s'_0$.
(b) By $(\mathsf{Bra}^{!_{\mathsf{M}}})$ at the $z$-input, we require $s \sqsubseteq \mathsf{tamp}(E)$ and $s \sqsubseteq s'_0$.
(c) By $(\mathsf{In}^{\downarrow_{\mathsf{A}}})$ at the $l$-input, we require $s_0 \sqsubseteq s'$ and $s_0 \sqsubseteq s'_0$.

Thus we reach the following conditions : (i) $s_0 \sqcup s \sqsubseteq s'_0$, (ii) $s \sqsubseteq \mathsf{tamp}(E)$, and (iii) $s_0 \sqsubseteq \mathsf{tamp}(E)$ ($s'$, which is a level for hidden names $f$ and $e$, is any such that $s_0 \sqsubseteq s' \sqsubseteq \mathsf{tamp}(E)$). Since $s_0$ can be raised by subsumption we may set $s_0 = s'_0$ without loss of precision, reaching the stated condition $s \sqsubseteq s_0 \sqsubseteq \mathsf{tamp}(E)$, which is equivalent to the condition by Smith [57] (in fact the condition in [57], which is $s'_0 = s_0 \sqcup s$ together with (ii) and (iii) above, are more directly derivable from these conditions). Note the structural security plays a fundamental rôle in this derivation.

### 7.8 Noninterference

An *environment* is a finite map from variables to values. $\sigma, \sigma', \ldots$ range over environments. We write $E \vdash \sigma$ when $\sigma$ is well-typed with respect to $E$ in the obvious sense, and often implicitly assume well-typedness of an environment under a given base. We define:

- $E \vdash \sigma_1 \sim_s \sigma_2$ denotes $\sigma_1(x) = \sigma_2(x)$ such that $\mathsf{protect}^{\mathcal{E}}(E(x)) \sqsubseteq s$ for each $x \in \mathtt{dom}(E)$.
- $[\![\sigma]\!] \stackrel{\text{def}}{=} \Pi_i \mathsf{Ref}\langle x_i c_i\rangle^{\circ}$ where $\mathsf{Ref}\langle x_i c_i\rangle^{\circ}$ is given as, with $\sigma(x_i) = v_i$, (1) $(\boldsymbol{\nu}\, c_i)(\mathsf{Ref}\langle x_i c_i\rangle | P)$ if $\langle v_i\rangle_u \equiv \overline{u}(c)P$ and (2) $\mathsf{Ref}\langle x_i y_i\rangle$ if $\langle v_i\rangle_u \equiv \overline{u}\langle y\rangle$.

Using the noninterference result for $\pi^{\mathsf{LAM}}$ (cf. Proposition 6.8), we obtain the following equational correspondence between environments and their embedding.

**Proposition 7.11** *Let $E$, $\sigma_{1,2}$ and $A$ be such that $E(x_i) = \mathtt{ref}_s(S_i)$, $\mathtt{dom}(\sigma_1) = \mathtt{dom}(\sigma_2) = \{x_1, ..., x_n\}$ and $A = x_1 : \mathsf{ref}_{s_1}\langle S_1^\circ\rangle \otimes v_1 : \overline{S_1^\circ} \otimes \cdots \otimes x_n : \mathsf{ref}_{s_n}\langle S_n^\circ\rangle \otimes v_n : \overline{S_n^\circ}$.*

(1) *If $E \vdash \sigma_1 \sim_s \sigma_2$ then $\vdash_{\mathtt{sec}} [\![\sigma_1]\!] \cong_s^\pi [\![\sigma_2]\!] \triangleright A$.*
(2) *Further let $T_i = \mathbb{N}_{s_i}$ for each $i$. Then $E \vdash \sigma_1 \sim_s \sigma_2$ iff $\vdash_{\mathtt{sec}} [\![\sigma_1]\!] \cong_s^\pi [\![\sigma_2]\!] \triangleright A$.*

PROOF. Let $\sigma_1(x_i) = v_i$ and $\sigma_2(x_i) = v_i'$ for each $x_i$ in their domain. For (1) if $s_i \not\sqsubseteq s$ then since $\mathsf{tamp}(\mathsf{ref}_{s_i}\langle T_i\rangle^\circ) \sqsupseteq s_i$ we have $\mathsf{tamp}(\mathsf{ref}_{s_i}\langle T_i\rangle^\circ) \not\sqsubseteq s$. Hence $\vdash_{\mathtt{sec}} (\boldsymbol{\nu}\, c_i)(\mathsf{Ref}\langle x_i c_i\rangle | \langle v_i\rangle_{c_i}^\circ) \cong_s^\pi (\boldsymbol{\nu}\, c_i)(\mathsf{Ref}\langle x_i c_i\rangle | \langle v_i'\rangle_{c_i}^\circ) \triangleright x_i : \mathsf{ref}_{s_i}\langle T_i\rangle^\circ$ for each $i$ by Proposition 6.8, hence done. For (2) we already know "only if" by (1). For the "if" direction we use a contextual reasoning. Suppose $\vdash_{\mathtt{sec}} [\![\sigma_1]\!] \cong_s^\pi [\![\sigma_2]\!] \triangleright A$ and $s_i \sqsubseteq s$. Then we have

$$\langle v_i\rangle_u \cong_s^\pi (\boldsymbol{\nu}\, \vec{x})([\![\sigma_1]\!] | \overline{x_i}\mathtt{inl}(c)c(y).\overline{u}\langle y\rangle) \cong_s^\pi (\boldsymbol{\nu}\, \vec{x})([\![\sigma_2]\!] | \overline{x_i}\mathtt{inl}(c)c(y).\overline{u}\langle y\rangle) \cong_s^\pi \langle v_i'\rangle_u.$$

By taking a suitable context we know $\langle v_i\rangle_u \cong_s^\pi \langle v_i'\rangle_u$ implies $v_i = v_i'$, as required. □

Write $(o, \sigma)_X \Downarrow \sigma'_{X'}$ if $(o,\, \sigma)_X \longrightarrow^* (\Pi_i \mathtt{skip},\, \sigma')_{X'}$. $(o, \sigma)_X \Downarrow$ stands for $(o, \sigma)_X \Downarrow \sigma'_{X'}$ for some $\sigma'$ and $X'$. We can now establish the operational correspondence. Below $P \cong^\pi Q$ stands for $P \cong_\dagger^\pi Q$.

**Proposition 7.12** (operational correspondence)

(1) (a) *Suppose $(o, \sigma)_X \longrightarrow (o', \sigma')_{X'}$. Then there exists $P'$ such that $(\boldsymbol{\nu}\, X)([\![o]\!]_{\vec{u}} | [\![\sigma]\!]) \longrightarrow^+ P'$ and $P' \cong^\pi (\boldsymbol{\nu}\, X')([\![o']\!]_{\vec{u}} | [\![\sigma']\!])$.*
(b) *Suppose $(o, \sigma)_X \twoheadrightarrow (o', \sigma')_{X'}$. Then there exists $P'$ such that $(\boldsymbol{\nu}\, X)([\![o]\!]_{\vec{u}} | [\![\sigma]\!]) \twoheadrightarrow P'$ and $P' \cong^\pi (\boldsymbol{\nu}\, X')([\![o']\!]_{\vec{u}} | [\![\sigma']\!])$.*

(2) *Suppose $(\boldsymbol{\nu}\, X)([\![o]\!]_{\vec{u}} | [\![\sigma]\!]) \longrightarrow P$ with $X \subset \mathtt{dom}(\sigma)$. Then $P \twoheadrightarrow \cong^\pi (\boldsymbol{\nu}\, X')([\![o']\!]_{\vec{u}} | [\![\sigma']\!])$ such that $(o, \sigma)_X \longrightarrow (o', \sigma')_{X'}$.*

(3) *If $(o, \sigma)_X \Downarrow \sigma'_{X'}$ then $(\boldsymbol{\nu}\, X)([\![o]\!]_{\vec{u}} | [\![\sigma]\!]) \twoheadrightarrow \vec{u} \mid R$ such that $R \cong (\boldsymbol{\nu}\, X')[\![\sigma']\!]$. Conversely, if $(\boldsymbol{\nu}\, X)([\![o]\!]_{\vec{u}} | [\![\sigma]\!]) \twoheadrightarrow \vec{u} \mid R$ with $X \subset \mathtt{dom}(\sigma)$ then $(o, \sigma)_X \Downarrow \sigma'_{X'}$ such that $R \cong^\pi (\boldsymbol{\nu}\, X')[\![\sigma']\!]$.*

PROOF. For (1-a), we first show $(o, \sigma)_X \longrightarrow (o', \sigma')_{X'}$ implies $(\boldsymbol{\nu}\, X)([\![o]\!]_{\vec{u}} | [\![\sigma]\!]) \longrightarrow \mapsto^+ (\boldsymbol{\nu}\, X')([\![o']\!]_{\vec{u}} | [\![\sigma']\!])$ where $\mapsto$ is the extended reduction as given in [9; 66; 67]. For reference we present these rules for unary cases (we assume the binding condition):

$$\begin{array}{rcl} x(\vec{y}).P | C[\overline{x}\langle\vec{z}\rangle] & \mapsto & C[P\{\vec{z}/\vec{y}\}] \\ !x(\vec{y}).P | C[\overline{x}\langle\vec{z}\rangle] & \mapsto & !x(\vec{y}).P | C[P\{\vec{z}/\vec{y}\}] \\ (\boldsymbol{\nu}\, x)!x(\vec{y}).P & \mapsto & \mathbf{0} \end{array}$$

where the second rule is applied only when $x$ is typed with a $\mathbf{!}_{\mathrm{L}}/\mathbf{!}_{\mathrm{A}}$-type (hence $\mapsto$ satisfies CR [8; 66; 67]). Since $\mapsto$ stays within $\cong^\pi$, this immediately gives us the stated property. (1-b) is a corollary of (1-a). (2) is similar, showing $[\![(o, \sigma)]\!]_{\vec{u}} \longrightarrow P$ implies $P \longrightarrow^* \mapsto^+ [\![(o', \sigma')]\!]_{\vec{u}}$ for each rule. (3) is from (1-a) and (2). □

We are now ready to establish the noninterference results.

**Theorem 7.13** (non-interference) *If $E \vdash o : cmd\, \tau_s$ and $E \vdash \sigma_1 \sim_s \sigma_2$ then $(o, \sigma_1) \Downarrow$ iff $(o, \sigma_2) \Downarrow$. If, moreover, $E(x_i) = \mathbb{N}_{s_i}$ for each $x_i \in \mathtt{dom}(\sigma)$, then $(o, \sigma_1) \Downarrow \sigma'_{1\,X_1}$ implies $(o, \sigma_2) \Downarrow \sigma'_{2\,X_2}$ such that $E \vdash \sigma_1' \restriction \mathtt{dom}(E) \sim_s \sigma_2' \restriction \mathtt{dom}(E)$.*

PROOF. For both statements, we use the following claim:

**Claim.** Let $E \vdash o : \mathrm{cmd}\,\tau_s$ and $E \vdash \sigma$. Then $(\boldsymbol{\nu}\, X)(\llbracket o \rrbracket_{\vec{u}} | \llbracket \sigma \rrbracket) \longrightarrow^* P \cong_s^\pi \Pi_i \overline{u_i}|R$ implies $P \equiv \Pi_i \overline{u_i}|R'$ such that $R \cong_s^\pi R'$.

For the proof of the claim, we use the context $C[\ \cdot\ ] \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, \vec{u})(S\langle \vec{u}, z\rangle | [\ \cdot\ ])$ where $S\langle \vec{u}, z\rangle$ is a synchroniser which signals, via an affine channel $z$, the arrival of all of $\vec{u}$. For example, the following process suffices (using booleans and their conjunctions w.l.o.g.).

$$S\langle \vec{u}, z\rangle \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, \vec{x}y)(\Pi_i u_i.\overline{x_i}\mathtt{in}_2(Fc)c.\mathbf{0} \mid \Pi_i \mathsf{Ref}\langle x_i T\rangle \mid \langle y := \wedge_i x_i \ ;\ \mathtt{while}\ !y\ \mathtt{do}\ \mathtt{skip}\rangle_z)$$

This agent stores information for each $u_i$ as it arrives in separate references, while polling these stores by busy-waiting. With the context $C[\ ]$ using this $S$, we can show (assuming appropriate typing in $\pi^{\mathsf{LAM}}$) that $C[P] \Downarrow_z$ iff $P \twoheadrightarrow \Pi_i \overline{u_i}|R$. We are now ready to show the first half. Assume the stated conditions. From $(o, \sigma_1) \Downarrow$ and Proposition 7.12 (3) we know $\llbracket (o, \sigma_1) \rrbracket_{\vec{u}} \longrightarrow^* \Pi_i \overline{u_i}|R$ for some $R$. We now infer:

$$\begin{array}{lcll} E \vdash \sigma_1 \sim_s \sigma_2 & \Rightarrow & \llbracket \sigma_1 \rrbracket \cong_s^\pi \llbracket \sigma_2 \rrbracket & \text{(Prop. 7.11 (1))} \\ & \Rightarrow & \llbracket o \rrbracket_{\vec{u}} \mid \llbracket \sigma_1 \rrbracket \cong_s^\pi \llbracket o \rrbracket_{\vec{u}} \mid \llbracket \sigma_2 \rrbracket & \text{(congruency of } \cong^\pi) \\ & \Rightarrow & \llbracket o \rrbracket_{\vec{u}} | \llbracket \sigma_2 \rrbracket \longrightarrow^* \Pi_i \overline{u_i}|R & \text{(Claim above)} \\ & \Rightarrow & (o, \sigma_2) \Downarrow & \text{(Prop.7.12 (3))} \end{array}$$

as required. For the second half, from $(o, \sigma_1)_X \longrightarrow^* {\sigma'_1}_{X_1}$ and Proposition 7.12 (3) we know $\llbracket o \rrbracket_{\vec{u}} | \llbracket \sigma_1 \rrbracket \longrightarrow^* P_1 \stackrel{\mathrm{def}}{=} \Pi_i \overline{u_i}|R_1$ such that $R_1 \cong_s^\pi (\boldsymbol{\nu}\, X_1)\llbracket \sigma'_1 \rrbracket$. Using this we infer:

$E \vdash \sigma_1 \sim_s \sigma_2$

$$\begin{array}{lcll} & \Rightarrow & \llbracket \sigma_1 \rrbracket \cong_s^\pi \llbracket \sigma_2 \rrbracket & \text{(Prop. 7.11 (1))} \\ & \Rightarrow & (\llbracket o \rrbracket_{\vec{u}} | \llbracket \sigma_1 \rrbracket) \cong_s^\pi (\llbracket o \rrbracket_{\vec{u}} | \llbracket \sigma_2 \rrbracket) & \text{(congruency of } \cong^\pi) \\ & \Rightarrow & (\llbracket o \rrbracket_{\vec{u}} | \llbracket \sigma_2 \rrbracket) \longrightarrow^+ P_2 \cong_s^\pi P_1 & \\ & \Rightarrow & P_2 \cong \prod_i \overline{u_i}|R_2 \text{ s.t. } R_2 \cong_s^\pi (\boldsymbol{\nu}\, \vec{z}_2)\llbracket \sigma'_2 \rrbracket & \\ & & \qquad \text{and } (o, \sigma_2) \twoheadrightarrow \sigma'_2. & \text{(Claim, Prop. 7.12 (3))} \\ & \Rightarrow & (\boldsymbol{\nu}\, X_1)\llbracket \sigma'_1 \rrbracket \cong^\pi (\boldsymbol{\nu}\, \vec{u})(\vec{u} \mid (\boldsymbol{\nu}\, X_1)\llbracket \sigma'_1 \rrbracket \mid \textstyle\prod u_i.\mathbf{0}) & \\ & & \qquad \cong_s^\pi (\boldsymbol{\nu}\, \vec{u})(\vec{u} \mid (\boldsymbol{\nu}\, X_2)\llbracket \sigma'_2 \rrbracket \mid \textstyle\prod u_i.\mathbf{0}) & \\ & & \qquad \cong^\pi (\boldsymbol{\nu}\, X_2)\llbracket \sigma'_2 \rrbracket & \text{(By } \mapsto \subset \cong^\pi) \\ & \Rightarrow & (\boldsymbol{\nu}\, X_1)\llbracket \sigma'_1 \rrbracket \cong_s^\pi (\boldsymbol{\nu}\, X_2)\llbracket \sigma'_2 \rrbracket & \text{(By } \cong^\pi \subset \cong_s^\pi) \\ & \Rightarrow & E \vdash \sigma'_1/X_1 \sim_s \sigma'_2/X_2 & \text{(Prop. 7.11 (2))} \end{array}$$

where, in the final step, we can take off $X_{1,2}$ because, due to the type of $\sigma_{1,2}$, $\llbracket \sigma'_{1,2} \rrbracket$ is the disjoint union of $\llbracket \sigma''_{1,2} \rrbracket$ and $S_{1,2}$ where $\mathtt{dom}(\sigma''_i) = \mathtt{dom}(\sigma_i)$ and $S_i$ is the parallel composition of replicated processes with subjects in $X_i$. □

We can generalise (2) to arbitrary types (using a contextual congruence for the extended VS-calculus) if we used a generalised version of Proposition 7.11 (2). Further we can prove a stronger non-interference property in which intermediate states are also $s$-equated (cf. [12]), by establishing a close operational correspondence between the original program phrases and their embedding. See Section 8 for more discussion.

$$[VarM]\ \frac{S \text{ mutable}}{E \cdot x^{\mathtt{w}}{:}S \vdash x : (S)_s} \qquad [Skip]\ \frac{-}{E \vdash \mathtt{skip} : \mathtt{COM}}$$

$$[LamM]\ \frac{E \cdot x^{\delta}{:}S \vdash M : U}{E \vdash \lambda x.M : S \stackrel{s}{\Rightarrow} U}\ (*) \qquad [AppM]\ \frac{E \vdash M : S \stackrel{s}{\Rightarrow} U \quad E \vdash N : (S)_{s'}}{E \vdash MN : U}\ (\star)$$

$$[Ass]\ \frac{E \cdot x^{\delta}{:}\mathtt{ref}_s(S) \vdash V : S}{E \cdot x^{\mathtt{w}}{:}\mathtt{ref}_s(S) \vdash x := V : \mathtt{COM}} \qquad [New]\ \frac{E \vdash V : S \quad E \cdot x^{\delta}{:}\mathtt{ref}_s(S) \vdash M : T}{E \vdash \mathtt{new}\ x \mapsto V\ \mathtt{in}\ M : T}$$

$$[Deref]\ \frac{E \cdot z^{\delta}{:}\mathtt{ref}_s(S) \cdot x{:}S \vdash M : T}{E \cdot z^{\delta_0}{:}\mathtt{ref}_s(S) \vdash \mathtt{let}\ x = !z\ \mathtt{in}\ M : T}\ \text{if } T \text{ mutable then } \delta_0 = \mathtt{w} \text{ else } \delta_0 = \delta$$

$(*)$ $s \sqsubseteq \mathsf{tamp}(E)$.
$(\star)$ $s' \sqsubseteq \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(U)$

Fig. 15. Additional Typing Rules for Imperative DCCv

## 8. DISCUSSIONS

### 8.1 Further Study on Imperative Secrecy

In Section 7 we proposed a new secure imperative and higher-order programming, extending Volpano-Smith multi-threaded imperative language with higher-order procedures and general references. In this subsection, we outline how we can directly apply the $\pi^{\mathsf{LAM}}$-calculus to propose another sequential secure imperative programming, extending DCCv (cf. 4.3) with imperative constructs. The syntax is extended as:

$$M \ ::= \ \ldots \mid x := V \mid \mathtt{let}\ x = !y\ \mathtt{in}\ M \mid \mathtt{new}\ x \mapsto V\ \mathtt{in}\ M \mid \mathtt{skip}$$

where $V$ denotes a value (the set of values are given by the union of variables, natural numbers, `skip` and abstractions). For types we only extend total types:

$$S \ ::= \ \ldots \mid \mathtt{ref}_s(S) \mid S \stackrel{s}{\Rightarrow} U \mid \mathtt{COM}$$

Unlike the language in Section 7, we use `COM` (which is equivalent to the unit type) instead of having a separate category for command types. For typing rules, we set:

- From Figure 7, the following rules are used without change: $[Num]$, $[Succ]$, $[Lift]$ and $[Rec]$.
- For $[App]$ in Figure 7, the side condition $s \sqsubseteq \mathsf{protect}(T)$ is replaced by $s \sqsubseteq \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(T)$. For the remaining rules ($[Var]$, $[If]$, $[Lam]$ and $[Seq]$), we give additional secrecy conditions following Figures 13 and 12.
- For imperative extensions, we add the rules in Figure 15.

The encoding follows Figure 18, based on which the noninterference is proved just as in Section 7. This language can soundly encode the sequential part of the extended Smith-Volpano language in Section 7. First, command types are recovered by regarding cmd $\Downarrow_s$ as `COM` (note $s$ does not matter for converging commands) and cmd $\Uparrow_s$ as $\llcorner\mathtt{COM}\lrcorner_s$. The language constructs are translated in the standard manner:

for example, `while` $!x$ `do` $c$ is encoded as follows (assuming $c$ is translated into $M$ and using shorthands for ; and $!x$):

$$(\mu z.\lambda y.\texttt{if}\ !x\ \texttt{then}\ M; z\ \texttt{else}\ y)\texttt{skip}$$

We can verify that the encoding yields precisely the same side condition as given in Figure 13. The syntax presented above would be useful when we consider secrecy analyses of functional languages with imperative constructs, such as ML.

### 8.2 Related Work and Further Issues

First, there are a few prominent examples of integrated function-based type disciplines, which often use monads. Basic examples include pointed types [32; 46] and the incorporation of imperative constructs in Haskell [33]. They offer not only combination of type structures but also preservation of individual type structures in the integrated types. The present work explores the same kind of integration in the context of the $\pi$-calculus, and shows its significance in secrecy analysis.

Secrecy and other security issues in processes are widely studied recently, cf. [1; 54; 17; 20]. [1] includes insightful discussions on secrecy. These studies mainly focus on modelling security concerns in cryptography protocols or distributed systems, and do not directly pursue integrated secrecy typing for language constructs. One of the challenging topics is the integration of the technologies as experimented in the present paper with secrecy/security concerns in distributed computing.

For secrecy analysis proper, we observe that the $\pi$-calculus-based secrecy analyses in the preceding work including our own, cf. [20; 27], have been based on the explicit recording of a secrecy level, whose sequent may be written

$$\vdash_s P \triangleright A$$

which means $P$ tampers the environment at a secrecy level at most $s$. Here we explicitly record the tampering level of $P$ independently from $A$ and handle its level via subtyping (i.e. we infer $\vdash_s P \triangleright A$ if $\vdash_{s'} P \triangleright A$ with $s \sqsubseteq s'$). In contrast, the approach in the present paper, which we may call *implicit approach*, does not use such $s$ but derive this level from $A$. In principle, $\pi^{\mathsf{LA}}$ and $\pi^{\mathsf{LAM}}$ can adopt both kinds of approaches, of which the present paper explored the implicit approach. At this point it looks the implicit approach tends to offer more typable terms while the explicit one has a merit in the imperative setting in that it does not need structural security, though details are to be seen. It is a valuable enterprise to study the explicit secrecy analysis to $\pi^{\mathsf{LA}}$ and $\pi^{\mathsf{LAM}}$, which would lead to precise comparisons of the merits and demerits of these approaches. It also opens a possibility to analyse and compare typability of the known implicit [3]/explicit [50; 58; 68] secure programming languages using the secure $\pi$-calculi as meta-calculi via translations.

The secrecy analysis proposed in the present paper owes much to the preceding work on type-based secrecy analyses for traditional functional and imperative languages. Among secure functional calculi (cf. [48; 49; 3; 19]), the dependency core calculus [3] is a powerful functional metalanguage for secrecy, using pointed types [32; 46]. The semantics is given by a denotational universe based on logical relations. The calculus is effective for analysing diverse sequential notions of dependency and secrecy. At the same time, the formalism is difficult to apply to the realm outside of sequential higher-order functions. The present work offers an

alternative tool which can easily incorporate impure features such as concurrency and state. Another significant aspect of the $\pi$-calculus would be its fine-grained nature as a meta-language, due to which developing analyses would in general become more elementary (this is especially true when there are subtle interplays between language constructs, as in languages we have studied in Sections 7 and §8.1 above).

Smith and Volpano (cf.[57; 58; 62]) studied various aspects of secrecy in imperative languages. Sequential procedures are studied in [62]. Multi-threading is studied in [58], whose typability was enlarged by our work with Vasconcelos [27] using the $\pi$-calculus, based on which a further enhancement was done in [57]. As we have seen in Proposition 7.6, our calculus is a conservative extension of the possibilistic part of the calculus in [57], integrating it with higher-order procedures and general references. One of the interesting aspects is the correspondence between the two kinds of command types in Smith-Volpano languages on the one hand and linearity/affinity in $\pi^{\mathsf{LA}}$ on the other. The incorporation of execution steps into secrecy typing [57] into the present framework is one of the remaining topics.

Boudol and Castellani [12] also studied a language similar to [57]. One of the significant features is the use of bisimulation for formulating and proving noninterference properties, leading to a stronger property. In this context, the use of secrecy bisimulation in $\pi^{\mathsf{LA}}$ [67] for proving similar results would be worth exploring. Another topic is the incorporation of *scheduler* into the $\pi^{\mathsf{LAM}}$-calculus to enrich language constructs following [12].

There are two prominent recent work which presented secrecy analysis for the combination of higher-order procedures and imperative features, one by Myers [44] (using Java) and another by Pottier and Simonet [50] (using ML). Both are based on explicit secrecy analysis (in the terminology we introduced above). For this reason, the treatment of a function with imperative effect (such as $\lambda x^{L}.x := 1; \mathtt{return}\ y^{H}$) is done using a closure type (and in effect would have a low tampering level) in [50], while it is treated just as other program phrases with a high-tampering level in our approach. We are currently working on an exact comparison in typability between our approach (based on implicit secrecy) and [50], using the $\pi$-calculus as an intermediate language (we believe this will fairly easily extends to the comparison with [44]). One aspect of secrecy in programming languages whose study has just started is secrecy for low-level programming primitives. In this respect, one interesting work [68] presents a typed control calculus with references, intended as a meta-language for possibly low-level languages via CPS translation. Its type discipline is adapted to this end, in particular in its use of linear continuations. As secrecy typing for imperative languages, [68] does not treat multi-threading, and is not (intended as) an extension of the language in [58; 57]. We believe that the incorporation of the dynamics and types in [68] into the $\pi$-calculus is a topic for the further study [24], as well as polymorphic extension of the $\pi^{\mathsf{LA}}$-calculus [10].

Finally we briefly discuss type inference in the presented secrecy analysis in $\pi^{\mathsf{LAM}}$. It is easy to show that, given $P$ and $A$, it is linearly decidable whether $\vdash P \triangleright A$ or not. Another possible question is to what extent a feasible type inference is possible when we leave out secrecy levels of some of the types as unknown. For example, one may wish to leave secrecy levels for implicit affine channels unknown, and deduce permissible secrecy levels as a whole. A study in this direction would offer a useful basis for considering possible forms of type inference for secrecy analyses in complex

programming languages.

## REFERENCES

[1] Abadi, M., Secrecy in programming-language semantics, *MFPS XV*, ENTCS, 20 (April 1999).

[2] Abadi, M., Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[3] Abadi, M., Banerjee, A., Heintze, N. and Riecke, J., A core calculus of dependency, *POPL'99*, ACM, 1999.

[4] Abramsky, S., Jagadeesan, R. and Malacaria, P., Full Abstraction for PCF, 1994. *Info. & Comp.* 163 (2000), 409-470.

[5] Abramsky, S., Honda, K. and McCusker, G., Fully Abstract Game Semantics for General References, *LICS'98*, 334–344, IEEE, 1998.

[6] Agat, J. Transforming Out Timing Leaks, *POPL'00*, 2000, ACM Press.

[7] Amtoft T., Nielson F., and Nielson H.R., Type and Effect Systems: Behaviours for Concurrency, IC Press, 1999.

[8] Berger, M., Honda, K. and Yoshida, N., Sequentiality and the $\pi$-Calculus, *TLCA01*, LNCS 2044, 29–45, Springer, 2001.

[9] The full version of [8]. Available at www.dcs.qmw.ac.uk/~kohei.

[10] Berger, M., Honda, K. and Yoshida, N., Genericity and the $\pi$-Calculus, a draft, January 2002. To appear as Queen Mary technical report.

[11] Boudol, G., Asynchrony and the pi-calculus, INRIA Research Report 1702, 1992.

[12] Boudol, G. and Castellani, I., Noninterference for Concurrent Programs, *ICALP01*, LNCS 2076, 382–395, Springer, 2001.

[13] Damas, L., *Type Assignemnt in Programming Languages*, PhD thesis, University of Edinburgh, 1985.

[14] Denning, D. and Denning, P., Certification of programs for secure information flow. *Communication of ACM*, ACM, 20:504–513, 1997.

[15] Fiore, M. Axiomatic domain theory in cagtegory of partial maps, Ph.D. Thesis, University of Edinburgh, 1994.

[16] Fiore, M. and Honda, K. Recursive Types in Games: Axiomatics and Process Representation. *LICS'98*, 1998.

[17] Focardi, R., Gorrieri, R. and Martinelli, F., Non-interference for the analysis of cryptographic protocols. *ICALP00*, LNCS 1853, Springer, 2000.

[18] Girard, J.-Y., Linear Logic, *TCS*, Vol. 50, 1–102, 1987.

[19] Heintze, N. and Riecke, J., The SLam calculus: programming with secrecy and integrity, *POPL'98*, 365–377, ACM, 1998.

[20] Hennessy, M. and Riely, J., Information flow vs resource access in the asynchronous pi-calculus, *ICALP00*, LNCS 1853, 415-427, Springer, 2000.

[21] Honda, K., Types for Dyadic Interaction. *CONCUR'93*, LNCS 715, 509-523, 1993.

[22] Honda, K., Composing Processes, *POPL'96*, 344-357, ACM, 1996.

[23] Honda, K., Notes on Linear Typing for Free Outputs, May, 2001.

[24] Honda, K., Notes on the linear $\pi$-calculus and LLP, June, 2001.

[25] Honda, K., Kubo, M. and Vasconcelos, V., Language Primitives and Type Discipline for Structured Communication-Based Programming. *ESOP'98*, LNCS 1381, 122–138. Springer-Verlag, 1998.

[26] Honda, K. and Tokoro, M. An object calculus for asynchronous communication. ECOOP'91, LNCS 512, 133–147, 1991.

[27] Honda, K., Vasconcelos, V. and Yoshida, N., Secure Information Flow as Typed Process Behaviour, *ESOP'00*, LNCS 1782, 180–199, 2000. Full version: MCS technical report 2000-01, University of Leicester, http://www.mcs.ac.uk/~nyoshida.

[28] Honda, K. and Yoshida, N. On Reduction-Based Process Semantics. *TCS*. 151, 437-486, 1995.

[29] Honda, K. and Yoshida, N. Game-theoretic analysis of call-by-value computation. *TCS*, 221 (1999), 393–456.

[30] Honda, K. and Yoshida, N., A Uniform Type Structure for Secure Information Flow, *POPL'02*, 81–92, ACM Press, 2002.

[31] Honda, K. and Yoshida, N., Noninterference proofs from Flow Analysis. Department of Computing Technical Report, Imperial College, 12pp. July, 2002. Available at `http://www.doc.ic.ac.uk/~noyo`.

[32] Howard, B. T., Inductive, coinductive, and pointed types, *ICFP'96*, 102–109, ACM, 1996.

[33] The Haskell home page, http://haskell.org.

[34] Hyland, M. and Ong, L., "On Full Abstraction for PCF": I, II and III. *Info. & Comp.* 163 (2000), 285-408.

[35] Hyland, M. and Ong, L., Pi-calculus, dialogue games and PCF, *FPCA'93*, ACM, 1995.

[36] Kobayashi, N., Pierce, B., and Turner, D., Linear types and the $\pi$-calculus, *TOPLAS*, 21(5):914–947, 1999.

[37] Jones, C., Tentative steps toward a development method for interfering programs, *ACM TOPLAS*, 5(4):596–619, 1983.

[38] Jones, C., Specification and design of (parallel) programs, *Information Processing*, 321–332, North-Holland, 1983.

[39] Leroy, X. and Weis, P., Polymorphic type inference and assignment, *POPL'91*, 291–302, ACM Press, 1991.

[40] Milner, R., A Calculus of Communicating Systems, LNCS 92, Springer, Berlin, 1980.

[41] Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989.

[42] Milner, R., Functions as Processes, *MSCS.* 2(2):119–141, 1992,

[43] Milner, R., Parrow, J. and Walker, D., A Calculus of Mobile Processes, *Info. & Comp.* 100(1):1–77, 1992.

[44] Myers, A. JFlow: Practical Mostly-Static Information Flow Control. *POPL'99*, pages 228-241, ACM Press, January 1999.

[45] Myers, A. and Liskov, B., A decentralized model for information flow control. *SOSP'97*, 129–142, ACM, 1997.

[46] Mitchell, J., *Foundations for Programming Languages* MIT Press, 1996.

[47] Palsberg, J., Type-based analysis and applications, 2001, available at: http://www.cs.purdue.edu/homes/palsberg.

[48] Palsberg, J. and Ørbaek , J., Trust in the $\lambda$-Calculus. *JFP*, 7(6):557–591, 1997.

[49] Pottier, F. and Conchon, S, Information flow inference for free, *ICFP00*, 46–57, ACM, 2000.

[50] Pottier, F. and Simonet, V., Information Flow Inference for ML, *POPL'02*, 319–330, ACM Press, 2002.

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

[52] Morris, J-H., Lambda calculus models of programming languages, PhD-Thesis, M.I.T, 1968.

[53] Nielson, F., Nielson H.R., and Hankin C., Principles of Program Analysis, Springer-Verlag, 1999.

[54] Ryan, P. and Schneider, S. Process Algebra and Non-interference. *CSFW'99*, IEEE, 1999.

[55] Sangiorgi, D. $\pi$-calculus, internal mobility, and agent-passing calculi. *TCS*, 167(2):235–271, 1996.

[56] Sabelfield, A. and Sand, D. A per model of secure information flow in sequential programs. *ESOP'99*, LNCS 1576, Springer, 1999.

[57] Smith, G., A New Type System for Secure Information Flow, *CSFW'01*, IEEE, 2001.

[58] Smith, G. and Volpano, D., Secure information flow in a multi-threaded imperative language, 355–364, *POPL'98*, ACM, 1998.

[59] Talpin, J.-P. and Jouvelot, P., The type and effect discipline, *Info. & Comp.*, Vol 111(2):245–296, Academic Press, 1992.

[60] Tofte, M., Type inference for polymorphic references, *Info. & Comp.*, 89:1–34, 1990.

[61] Vasconcelos, V., Typed concurrent objects. *ECOOP'94*, LNCS 821, 100–117. Springer, 1994.

[62] Volpano, D., Smith, G. and Irvine, C., *A Sound type system for secure flow analysis.* J. Computer Security, 4(2,3):167–187, 1996.

[63] Wright, A., Typing references by effect inference, *ESOP'92*, LNCS 582, Springer, 1992.

[64] Yoshida, N. Graph Types for Mobile Processes. *FST/TCS'16*, LNCS 1180, 371–386, Springer, 1996. The full version as LFCS Technical Report, ECS-LFCS-96-350, 1996.

[65] Yoshida, N., Type-Based Liveness Guarantee in the Presence of Nontermination and Nondeterminism, April 2002. MCS Technical Report, 2002-20, University of Leicester. Available at www.mcs.le.ac.uk/~yoshida.

[66] Yoshida, N., Berger, M. and Honda, K., Strong Normalisation in the $\pi$-Calculus, *LICS'01*, 311–322, IEEE, 2001. The full version as MCS technical report, 2001-09, University of Leicester, 2001. Available at www.mcs.le.ac.uk/~yoshida.

[67] Yoshida, N., Honda, K. and Berger, M. Linearity and Bisimulation, Proc. of 5th International Conference, Foundations of Software Science and Computer Structures (FoSSaCs 2002), LNCS 2303, pp.417–433, Springer, 2002. A full version as a MCS technical report, 2001-48, University of Leicester, 2001. Available at www.mcs.le.ac.uk/~yoshida.

[68] Zdancewic, S. and Myers, A., Secure Information Flow and CPS, *ESOP01*, LNCS 2028, 46-62, Springer, 2001.

## A. ACTION TYPES

We first reiterate the definition of action types. 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

—edges are of the form $x:\tau \to y:\tau'$.

We write $x \to y$ if $x:\tau \to y:\tau'$ for some $\tau$ and $\tau'$, in a given action type. If $x$ occurs in $A$ and for no $y$ we have $y \to x$ then we say $x$ is *active in* $A$. $|A|$ (resp. $\mathsf{fn}(A)$, $\mathsf{sbj}(A)$, $\mathsf{md}(A)$) denotes the set of nodes (resp. names, active names, modes) in $A$. We often write $x:\tau \in A$ instead of $x:\tau \in |A|$, and write $A(x)$ for the channel type assigned to $x$ in $A$. $A/\vec{x}$ is the result of taking off nodes with names in $\vec{x}$ from $A$. $A \otimes B$ is the graph union of $A$ and $B$, with the condition that $\mathsf{fn}(A) \cap \mathsf{fn}(B) = \emptyset$.

The symmetric partial operator $\odot$ on types is already given in Section 2. We then write $A \asymp B$ iff:

—whenever $x:\tau \in A$ and $x:\tau' \in B$, $\tau \odot \tau'$ is defined; and

—whenever $x_1 \to x_2$, $x_2 \to x_3$, $\ldots$, $x_{n-1} \to x_n$ alternately in $A$ and $B$ ($n \geq 2$), 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) $x \in (\mathsf{fn}(A) \setminus \mathsf{fn}(B)) \cup (\mathsf{fn}(B) \setminus \mathsf{fn}(A))$ and $x:\tau$ occurs in $A$ or $B$; or (2) $x:\tau' \in A$ and $x:\tau'' \in B$ and $\tau = \tau' \odot \tau''$.

—$x \to y$ in $A \odot B$ iff $x = z_1 \to z_2$, $z_2 \to z_3, \ldots, z_{n-1} \to z_n = y$ alternately in $A$ and $B$ ($n \geq 2$) and, moreover, for no $w$ we have $w \to x$ and for no $w'$ we have $y \to w'$ in $A$ or $B$.

We can easily check that $\odot$ is a symmetric and associative partial operation on action types with unit $\emptyset$.

## B. COPYCATS

Let $\mathsf{md}(\tau)$ be an input type in $\pi^{\mathrm{LA}}$. Then $[x \to x']^{\tau}$ is given by the following induction.

$$\begin{array}{rcll}
[x \to x']^{(\vec{\tau})^p} & \stackrel{\text{def}}{=} & x(\vec{y}).\overline{x'}(\vec{y'})\Pi_i[y'_i \to y_i]^{\overline{\tau_i}} & (p \in \mathcal{M}_{\downarrow}) \\
[x \to x']^{(\vec{\tau})^p} & \stackrel{\text{def}}{=} & !x(\vec{y}).\overline{x'}(\vec{y'})\Pi_i[y'_i \to y_i]^{\overline{\tau_i}} & (p \in \mathcal{M}_{!}) \\
[x \to x']^{[\&_i \vec{\tau}_i]^p} & \stackrel{\text{def}}{=} & x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i(\vec{y'}_i)\Pi_{ij}[y'_{ij} \to y_{ij}]^{\overline{\tau_{ij}}}] & (p \in \mathcal{M}_{\downarrow}) \\
[x \to x']^{[\&_i \vec{\tau}_i]^p} & \stackrel{\text{def}}{=} & !x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i(\vec{y'}_i)\Pi_{ij}[y'_{ij} \to y_{ij}]^{\overline{\tau_{ij}}}] & (p \in \mathcal{M}_{!})
\end{array}$$

Note $[x \to y]$ in Example 2.1 (4) is precisely $[x \to y]^{\mathbb{N}^\circ}$. Using general copy-cats, we can build divergent agents with arbitrary types as: $\Omega_u^{\tau} \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, x)([u \to x]|[x \to u])$ where $\mathsf{md}(\tau) = \mathbf{!}_{\mathrm{A}}$. Then $\vdash_{\mathsf{sec}} \Omega_u^{\tau} \triangleright x : \tau$ always. A copy-cat with a reference type is introduced as a constant which is inferred by the following axiom (since we do not include $(\mathsf{Bra}^{?_{\mathrm{LM}}})$ in $\pi^{\mathrm{LAM}}$). We show the secrecy-sensitive version below.

$$\frac{\text{(CRef)} \qquad s \sqsubseteq s' \quad \text{if } \mathsf{md}(\tau) \in \{\mathbf{!}_{\mathrm{LM}}, \mathbf{!}_{\mathrm{M}}\} \text{ then } \delta = \mathtt{w} \text{ else } \delta = \varepsilon}{\vdash_{\mathsf{sec}} [x \to y]^{\mathsf{ref}_s\langle\tau\rangle} \triangleright x : \mathsf{ref}_s\langle\tau\rangle \to y^{\delta} : \overline{\mathsf{ref}_{s'}\langle\tau\rangle}}$$

Note the causality is recorded in this rule, reflecting the linear character of replication in references. Opereationally the copycat $[x \to y]^{\mathsf{ref}_s\langle\tau\rangle}$ is representable as $!x[\ (c).\overline{y}\mathtt{inl}(c')[c' \to c]^{(\overline{\tau})^{\downarrow_{\mathrm{L}}}}\ \&\ (vc).\overline{y}\mathtt{inr}(v'c')([v' \to v]^{\tau}|c'.\overline{c})]$. $\pi^{\mathrm{LAM}}$ extended with this constant does not alter the essential operational properties of $\pi^{\mathrm{LAM}}$ such as liveness at linear channels [66; 65; 67]. Proposition 3.13 (1) extends to reference types. For secrecy analysis, adding this constant does not change the nature of the analysis, leading to the non-interference via precisely the same method. Proposition 3.13 (2) is intact due to structural security.

Using copy-cats, we can translate arbitrary (typed) free outputs by bound name outputs. The encoded free outputs mention types and are defined as follows.

$$\overline{x}\langle y_1..y_n\rangle^{\tau_1..\tau_n} \stackrel{\text{def}}{=} \overline{x}(y'_1..y'_n)\Pi_i[y'_i \to y_i]^{\overline{\tau_i}}$$

assuming $\vdash \overline{x}\langle y_1..y_n\rangle \triangleright x : (\tau_1..\tau_n)^{p_{\mathrm{O}}} \otimes y_1..y_n : \tau_1..\tau_n$.

## C. SUBJECT REDUCTION

### C.1 $\pi^{\mathrm{LA}}$ and $\pi^{\mathrm{LAM}}$

In this section, we prove the following result for the secrecy typing for $\pi^{\mathrm{LAM}}$ (subsuming the same property for $\pi^{\mathrm{LA}}$).

**(subject reduction)** *Let $\vdash_{\mathsf{sec}} P \triangleright A$. Then $P \twoheadrightarrow P'$ implies $\vdash_{\mathsf{sec}} P' \triangleright A$.*

Note that if the subject reduction is satisfied as above, then the subject reduction of the $\pi^{\mathrm{LA}}$, $\pi^{\mathrm{LAM}}$ and $\pi^{\mathrm{LA}}$'s secrecy extension are automatically proved. For the proof, we follow the same routine as given in Appendix A.1 in the long version of [66]. We also follow the same routine as given in the proof of Proposition 3 in [65] for references. What corresponds to Lemma A.1 of [66] (well-definedness of operators) is easy. A basic lemma follows, which corresponds to Lemma A.2 in [66].

**Lemma C.1** *Let $A_1$, $A_2$, $A_3$, $A$ and $B$ be action types. Then we have:*

(i) (commutativity) *Assume $A_1 \asymp A_2$. Then we have $A_2 \asymp A_1$ and $A_1 \odot A_2 = A_2 \odot A_1$.*
(ii) (associativity) *Assume $A_1 \asymp A_2$ and $(A_1 \odot A_2) \asymp A_3$. Then we have:* (1) $A_1 \asymp A_3$ *and* $A_2 \asymp A_3$, (2) $A_1 \asymp (A_2 \odot A_3)$ *and* (3) $(A_1 \odot A_2) \odot A_3 = A_1 \odot (A_2 \odot A_3)$.
(iii) *If $x : \tau \in |A|$ and $\mathsf{md}(\tau) \in \mathcal{M}_{!,\downarrow}$ then there is no $y : \tau' \in |A|$ such that $y : \tau' \to x : \tau$.*
(iv) *If $A \asymp B$ with $A/\vec{x} = A_0$, $x_i : \tau_i \in |A|$, $\mathsf{md}(\tau_i) \in \mathcal{M}_{!,\downarrow\updownarrow}$, and $\mathsf{fn}(B) \cap \{\vec{x}\} = \emptyset$, then $A_0 \asymp B$ and $(A \odot B)/\vec{x} = A_0 \odot B$.*
(v) *If $A \asymp B$ with $A/\vec{x} = A_0$, $x_i : \tau_i \in |A|$, $\mathsf{md}(\tau_i) \in \mathcal{M}_{!,\downarrow}$, and $B/\vec{x} = B_0$, then $A_0 \asymp B$, $A \asymp B_0$, $A_0 \asymp B_0$, and $(A \odot B)/\vec{x} = A_0 \odot B_0$.*
(vi) *$?B \asymp ?B$ and $B \odot B = B$.*
(vii) *Suppose $A/\vec{x} = A_0$, $B/\vec{x} = B_0$ and $A_0 \asymp B_0$. Assume also $x_i : \tau_i \in |A|$, $x_i : \tau_i' \in |B|$ with $\tau_i \asymp \tau_i'$, and $\mathsf{md}(\tau_i) \in \mathcal{M}_{!,\downarrow}$. Then $A \asymp B$ and $(A \odot B)/\vec{x} = A_0 \odot B_0$.*

PROOF. By regarding $!_{\text{LM}}/?_{\text{LM}}$ as $!_{\text{A}}/?_{\text{A}}$, we can essentially follow the reasoning given in [9; 66]. (i,ii) are immediate from the definitions. (iii) is obvious since there is no edge to input/$\updownarrow$ nodes. (iv) is because we can write $A = \otimes_i (x_i : \tau_i \to A_i) \otimes A'$ by the side condition $\mathcal{M}_{!,\downarrow\updownarrow}$ (note $A_i$ may be $\emptyset$). Then by $A \asymp B$, obviously $A_0 = (A_i \otimes A') \asymp B$. Hence we have $(A \odot B)/VECx = (A/\vec{x} \odot B/\vec{x}) = A_0 \odot B$. Similarly for (v). (vi) is by $\tau \odot \tau = \tau$ with $\mathsf{md}(\tau) \in \mathcal{M}_?$. (vii) uses (v) and (vi). □

**Lemma C.2** (substitution lemma) *In the following, $\delta$ is either $\varepsilon$ or $\mathtt{w}$.*
*If $\vdash_{\text{sec}} P \triangleright x : \tau \otimes A$, $\mathsf{md}(\tau) \in \mathcal{M}_\uparrow$ and $y \notin \mathsf{fn}(A)$, then*
(a) $\vdash_{\text{sec}} P\{y/x\} \triangleright y : \tau \otimes A$ *and* (b) $\mathsf{tamp}(x : \tau \otimes A) = \mathsf{tamp}(y : \tau \otimes A)$.

*(1)* (innocuous client type) *If $\vdash_{\text{sec}} P \triangleright x : \tau \otimes A$, $\mathsf{md}(\tau) \in \mathcal{M}_?$ and $A(y) = \tau$, then*
(a) $\vdash_{\text{sec}} P\{y/x\} \triangleright A$ *and* (b) $\mathsf{tamp}(x : \tau \otimes A) = \mathsf{tamp}(A)$.

*(2)* (mutable client type) *Suppose $\vdash_{\text{sec}} P \triangleright x^\delta : \tau \otimes y^{\delta'} : \tau \otimes A$ with either $\delta = \delta'$ or $\delta = \varepsilon$ and $\delta' = \mathtt{w}$. Then we have* (a) $\vdash_{\text{sec}} P\{y/x\} \triangleright y^{\delta'} : \tau \otimes A$ *and*
(b) $\mathsf{tamp}(x^\delta : \tau \otimes y^{\delta'} : \tau \otimes A) = \mathsf{tamp}(y^{\delta'} : \tau \otimes A)$.

**Remark:** For the subject reduction theorem, we do not have to prove the case such that $\delta' = \varepsilon$ and $\delta = \mathtt{w}$.

PROOF. (b) of (1,2) is by definition. (a) is proved by the rule induction of $\vdash_{\text{sec}} P \triangleright A$. The only interesting case is $x \in \mathsf{fn}(P)$ and the last rule is either (Par), $(\mathsf{In}^{\downarrow_{\text{A}}})$, $(\mathsf{Bra}^{\downarrow_{\text{L}}})$, $(\mathsf{In}^{!_{\text{LM}}})$ or $(\mathsf{Bra}^{\downarrow_{\text{A}}})$. We prove (1) in the case of (Par) and (3) in the case of $(\mathsf{In}^{!_{\text{LM}}})$. Others are similar.

Suppose $\vdash_{\text{sec}} P_1 \mid P_2 \triangleright x : \tau \otimes A$ and this is obtained by (Par) as follows.

$$\frac{\vdash_{\text{sec}} P_1 \triangleright A_1 \otimes x : \tau \qquad \vdash_{\text{sec}} P_2 \triangleright A_2}{\vdash_{\text{sec}} P_1 \mid P_2 \triangleright A_1 \odot A_2 \otimes x : \tau}$$

In the above we assume $x \in \mathsf{fn}(P_1)$ (hence $x \notin \mathsf{fn}(P_2)$) and $A_1 \odot A_2 = A$. We prove

$$\vdash_{\text{sec}} (P_1 \mid P_2)\{y/x\} \triangleright A_1 \odot A_2 \otimes y : \tau \tag{15}$$

By inductive hypothesis, we have $\vdash_{\mathsf{sec}} P_1\{y/x\} \triangleright A_1 \otimes y\!:\!\tau$. Since $x, y \notin \mathsf{fn}(A_2)$, we have $(A_1 \otimes y\!:\!\tau) \asymp A_2$ and $(A_1 \otimes y\!:\!\tau) \odot A_2 = A \otimes y\!:\!\tau$. Hence by $(P_1 \mid P_2)\{y/x\} = (P_1\{y/x\} \mid P_2)$, we have (15) as required.

Next suppose $\vdash_{\mathsf{sec}} !a(\vec{z}).Q \triangleright A \otimes x\!:\!\tau \otimes y^{\mathtt{w}}\!:\!\tau$. Assume the last applied rule is $(\mathsf{In}^{!_{\mathrm{M}}})$ as follows.

$$\frac{\begin{array}{l} s \sqsubseteq \mathsf{tamp}(A \otimes x\!:\!\tau \otimes y^{\mathtt{w}}\!:\!\tau) \\ \vdash_{\mathsf{sec}} Q \triangleright \vec{z}\!:\!\vec{\tau} \otimes \mathtt{?}A^{\text{-}a} \otimes x\!:\!\tau \otimes y^{\mathtt{w}}\!:\!\tau \end{array}}{\vdash_{\mathsf{sec}} !a(\vec{z}).Q \triangleright a\!:\!(\vec{\tau})^{!_{\mathrm{M}}}_{s} \otimes A \otimes x\!:\!\tau \otimes y^{\mathtt{w}}\!:\!\tau}$$

Then we prove

$$\vdash_{\mathsf{sec}} !a(\vec{z}).Q\{y/x\} \triangleright a\!:\!(\vec{\tau})^{!_{\mathrm{M}}}_{s} \otimes A \otimes y^{\mathtt{w}}\!:\!\tau \tag{16}$$

By inductive hypothesis, we have

$$\vdash_{\mathsf{sec}} Q\{y/x\} \triangleright \vec{z}\!:\!\vec{\tau} \otimes \mathtt{?}A \otimes y^{\mathtt{w}}\!:\!\tau \tag{17}$$

By (b), we know $s \sqsubseteq \mathsf{tamp}(A \otimes x^{\varepsilon}\!:\!\tau \otimes y^{\mathtt{w}}\!:\!\tau) = \mathsf{tamp}(A \otimes y^{\mathtt{w}}\!:\!\tau)$. Hence by applying $(\mathsf{In}^{!_{\mathrm{M}}})$ to the above again, we conclude (16) as desired. □

**Lemma C.3**

(i) $\vdash_{\mathsf{sec}} P \triangleright A$ *and* $P \equiv Q$ *then* $\vdash_{\mathsf{sec}} Q \triangleright A$.
(ii) $\vdash_{\mathsf{sec}} x(\vec{y}).P \mid \overline{x}\langle\vec{v}\rangle \triangleright A$ *implies* $\vdash_{\mathsf{sec}} P\{\vec{v}/\vec{y}\} \triangleright A$. *Similarly for selection.*
(iii) $\vdash_{\mathsf{sec}} !x(\vec{y}).P \mid \overline{x}\langle\vec{v}\rangle \triangleright A$ *implies* $\vdash_{\mathsf{sec}} P\{\vec{v}/\vec{y}\} \mid !x(\vec{y}).P \triangleright A$. *Similarly for selection.*
(iv) $\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{x}\mathtt{inl}\langle c\rangle \triangleright A$ *implies* $\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{c}\langle y\rangle \triangleright A$
(v) $\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{x}\mathtt{inr}\langle wc\rangle \triangleright A$ *implies* $\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xw\rangle \mid \overline{c} \triangleright A$.

PROOF: For (i), we can use the same reasoning as the proof of (i) in Lemma A.3 in [66]; For example, in the structural rule

$$(\boldsymbol{\nu}\,\vec{y})P \mid Q \;\equiv\; (\boldsymbol{\nu}\,\vec{y})(P \mid Q) \quad \text{with } y_i \notin \mathsf{fn}(Q)$$

$y_i$'s mode should be $\mathcal{M}_{!,\updownarrow}$ because of the definition of (Res). Hence we can use (iv) and (v) of Lemma C.1 as in [66]. For (ii) and (iii), we first note that if $v_i$ in $\overline{x}\langle\vec{v}\rangle$ has mode $\mathtt{?}_{\mathrm{LM}}$ or $\mathtt{?}_{\mathrm{M}}$, then $v_i$ has write mode, so we can use the condition of (3) in Substitution Lemma. Then we can directly use (v,vi,vii) of Lemma C.1 together with Substitution Lemma to prove (ii) and (iii) as in [66]. Hence we only have to prove (iv) and (v).

**(iv, read):** We prove this statement by rule induction. If the last rule is (Weak), then it is trivial. So assume

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{x}\mathtt{inl}\langle c\rangle \triangleright x\!:\!\mathsf{ref}_s\langle\tau\rangle \otimes y^{\delta}\!:\!\overline{\tau} \otimes c\!:\!(\tau)^{\uparrow_{\mathrm{L}}} \tag{18}$$

where $\delta = \mathtt{w}$ if $\mathsf{md}(\tau) \in \{\mathtt{?}_{\mathrm{M}}, \mathtt{?}_{\mathrm{LM}}\}$. Then we prove:

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{c}\langle y\rangle \triangleright x\!:\!\mathsf{ref}_s\langle\tau\rangle \otimes y^{\delta}\!:\!\overline{\tau} \otimes c\!:\!(\tau)^{\uparrow_{\mathrm{L}}} \tag{19}$$

We know (18) is inferred by (Par). Hence we have:

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \triangleright x\!:\!\mathsf{ref}_s\langle\tau\rangle \otimes y^{\delta}\!:\!\overline{\tau} \quad \text{and} \quad \vdash_{\mathsf{sec}} \overline{x}\mathtt{inl}\langle c\rangle \triangleright c\!:\!(\tau)^{\uparrow_{\mathrm{L}}} \otimes x^{\delta'}\!:\!\overline{\mathsf{ref}_s\langle\tau\rangle}$$

By this, we can apply (Out) to obtain:

$$\vdash_{\mathsf{sec}} \overline{c}\langle y\rangle \ \triangleright\ c\colon(\tau)^{\uparrow_{\mathrm{L}}} \otimes y^{\delta}\colon\overline{\tau} \tag{20}$$

By (iv) of Lemma C.1, we know $y^{\delta}\colon\overline{\tau} \odot y^{\delta}\colon\overline{\tau} = y^{\delta}\colon\overline{\tau}$. By applying (Par) to $\mathsf{Ref}\langle xy\rangle$ and (20) again, we now have (19) as required.

**(v, write):** Assume

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{x}\mathtt{inr}\langle wc\rangle \ \triangleright\ x\colon\mathsf{ref}_s\langle\tau\rangle \otimes w^{\delta}\colon\overline{\tau} \otimes c\colon()^{\uparrow_{\mathrm{L}}} \otimes y^{\delta_y}\colon\overline{\tau} \tag{21}$$

where $\delta, \delta_y = \mathtt{w}$ if $\mathsf{md}(\tau) \in \{?_{\mathrm{LM}}, ?_{\mathrm{M}}\}$ (else $\delta$ and $\delta_y$ are arbitrary due to (Weak-w)). We shall prove:

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xw\rangle \mid \overline{c} \ \triangleright\ x\colon\mathsf{ref}_s\langle\tau\rangle \otimes w^{\delta}\colon\overline{\tau} \otimes c\colon()^{\uparrow_{\mathrm{L}}} \otimes y^{\delta_y}\colon\overline{\tau} \tag{22}$$

We know (21) is inferred by (Par), so we have:

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \ \triangleright\ x\colon\mathsf{ref}_s\langle\tau\rangle \otimes y^{\delta_y}\colon\overline{\tau} \quad \text{and} \quad \vdash_{\mathsf{sec}} \overline{x}\mathtt{inr}\langle wc\rangle \ \triangleright\ x^{\mathtt{w}}\colon\overline{\mathsf{ref}_s\langle\tau\rangle} \otimes w^{\delta}\colon\overline{\tau} \otimes c\colon()^{\uparrow_{\mathrm{L}}}$$

Then, we have

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xw\rangle \ \triangleright\ x\colon\mathsf{ref}_s\langle\tau\rangle \otimes w^{\delta}\colon\overline{\tau} \quad \text{and} \quad \vdash_{\mathsf{sec}} \overline{c} \ \triangleright\ c\colon()^{\uparrow_{\mathrm{L}}} \tag{23}$$

Now by applying (Par) above, and by $x\colon\mathsf{ref}_s\langle\tau\rangle \odot x^{\mathtt{w}}\colon\overline{\mathsf{ref}_s\langle\tau\rangle} = x\colon\mathsf{ref}_s\langle\tau\rangle$, we have (22), as desired.

### C.2 $\pi^{\mathrm{LA}}$ with Inflation

In this subsection we prove the subject reduction for $\pi^{\mathrm{LA}}$ with inflation. For brevity of the proof, we consider the unary fragment and use the following restricted variant of (BOut). Below we write $P^{y}$ when $P$ is input-prefixed by $P$:

$$(\mathsf{BOut}') \quad \frac{\vdash_{\mathsf{sec}} \Pi P_i^{y_i} \ \triangleright\ C^{\vec{y}:\vec{\tau}} \quad C \asymp x\colon(\vec{\tau})^{p_{\mathrm{O}}}}{\vdash_{\mathsf{sec}} \overline{x}(\vec{y})\Pi P_i \ \triangleright\ C/\vec{y} \odot x\colon(\vec{\tau})^{p_{\mathrm{O}}}} \tag{24}$$

This rule does not lose generality in comparison with the original (BOut) since, up to $\equiv$, all bound output can be decomposed into this form combined with parallel composition. We also assume (Weak) is combined with input prefix rules when discussing derivations. This does not lose generality since (Weak) is only needed (if ever) just before each prefix rule and restriction.

**Convention C.4** In the present subsection the secrecy typing is for $\pi^{\mathrm{LA}}$ combined with inflation, i.e. we write $\vdash_{\mathsf{sec}} P \ \triangleright\ A$ when the sequent is derivable by the basic secrecy typing for $\pi^{\mathrm{LA}}$ combined with (Inf) and (Str) in § 3.5.

We first observe:

**Lemma C.5**

(1) $(A \sqcup s) \sqcup s' = A \sqcup (s \sqcup s')$.
(2) $\mathsf{tamp}(A) \sqcup s = \mathsf{tamp}(A \sqcup s)$. *Thus* $\mathsf{inf}(A) \sqcup s = \mathsf{inf}(A \sqcup s)$. *Further* $(A \otimes B) \sqcup s = (A \sqcup s) \otimes (B \sqcup s)$.
(3) *If* $A \asymp B$ *then we have* $(A \sqcup s) \asymp (B \sqcup s)$ *and* $(A \sqcup s) \odot (B \sqcup s) = (A \odot B) \sqcup s$.

(4) $(\mathsf{In}^{\downarrow_{\mathrm{L}}})$ *commutes over* $(\mathsf{Inf})$*, i.e.* $(\mathsf{In}^{\downarrow_{\mathrm{L}}})$ *followed by* $(\mathsf{Inf})$ *can be equivalently replaced by* $(\mathsf{Inf})$ *followed by* $(\mathsf{In}^{\downarrow_{\mathrm{L}}})$*. Similarly for* $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$, $(\mathsf{In}^{!_{\mathrm{L}}})$, $(\mathsf{In}^{!_{\mathrm{A}}})$ *as well as their branching counterpart. The same holds for* $(\mathsf{BOut'})$ *when the prefix is linear.*

(5) *If* $\vdash_{\mathsf{sec}} P \triangleright A$ *then* $\vdash_{\mathsf{sec}} P \triangleright A \sqcup s$ *for each* $s$*.*

PROOF. (1) is immediate. (2) is from (1). (3) is easy. (4) is because the tamper level does not change by the application of these prefix rules and linear unary output (for example, $\mathsf{tamp}((\tau_1..\tau_n)^{\downarrow_{\mathrm{L}}}) = \sqcap_i \mathsf{tamp}(\tau_i)$). (5) is mechanical by rule induction of $\vdash_{\mathsf{sec}} P \triangleright A$, using (1,2,3). We only show a couple of cases. Let $s$ be arbitrary in each case below.

**(Par).** Let $\vdash_{\mathsf{sec}} P|Q \triangleright A \odot B$ be derived from $\vdash_{\mathsf{sec}} P \triangleright A$ and $\vdash_{\mathsf{sec}} P \triangleright B$. By (IH) we have $\vdash_{\mathsf{sec}} P \triangleright A \sqcup s$ and $\vdash_{\mathsf{sec}} P \triangleright B \sqcup s$. Since by $A \asymp B$ and (3) we have $(A \sqcup s) \asymp (B \sqcup s)$, hence $\vdash_{\mathsf{sec}} P|Q \triangleright (A \sqcup s) \odot (B \sqcup s)$, to which we apply (3) again.

**$(\mathsf{In}^{\downarrow_{\mathrm{A}}})$.** Let $\vdash_{\mathsf{sec}} x(\vec{y}).P \triangleright x : (\vec{\tau})^{\downarrow_{\mathrm{A}}}_{s'} \otimes \uparrow_{\mathrm{A}} A \otimes ?B$ from $\vdash_{\mathsf{sec}} P \triangleright \vec{y} : \vec{\tau} \otimes \uparrow_{\mathrm{A}} A \otimes ?B$ where, by the associated secrecy constraint, we have $s' \sqsubseteq \mathsf{tamp}(A \otimes B)$. From this condition we know $s' \sqcup s \sqsubseteq \mathsf{tamp}(A \otimes B) \sqcup s = \mathsf{tamp}(A \sqcup s \otimes B \sqcup s)$. We can now use (IH) and apply (2) again.

**(Inf).** Suppose $\vdash_{\mathsf{sec}} P \triangleright A$ from $\vdash_{\mathsf{sec}} P \triangleright \mathsf{inf}(A)$. By (IH) we have $\vdash_{\mathsf{sec}} P \triangleright (\mathsf{inf}(A)) \sqcup s$ which is equivalent to $\vdash_{\mathsf{sec}} P \triangleright \mathsf{inf}(A \sqcup s)$ by (2), to which we apply (Inf) to obtain $\vdash_{\mathsf{sec}} P \triangleright A \sqcup s$, as required.

Other cases are immediate from the induction hypotheses. □

We can now prove the main results. Below it suffices to show the case when $(\mathsf{Par})$ is the last rule since, if $(\mathsf{Inf})$ is used, we can apply $(\mathsf{Inf})$ to the new derivation.

**Proposition C.6**

(1) *If* $\vdash_{\mathsf{sec}} x(\vec{y}).P|\overline{x}(\vec{y})Q \triangleright A$ *is derived using* $(\mathsf{Par})$ *as the last applied rule, then* $\vdash_{\mathsf{sec}} (\boldsymbol{\nu}\, \vec{y})(P|Q) \triangleright A/x$*. Similarly for selection.*

(2) *If* $\vdash_{\mathsf{sec}} !x(\vec{y}).P|\overline{x}(\vec{y})Q \triangleright A$ *is derived using* $(\mathsf{Par})$ *as the last applied rule, then* $\vdash_{\mathsf{sec}} !x(\vec{y}).P|(\boldsymbol{\nu}\, \vec{y})(P|Q) \triangleright A$*. Similarly for selection.*

PROOF. By Lemma C.5 (1), the only non-trivial case is when the prefix is not unary linear in (1). We show the case when it is unary affine. W.l.o.g. we prove the monadic case. Suppose $\vdash_{\mathsf{sec}} x(y).P|\overline{x}(y)Q \triangleright A$ is derived using $(\mathsf{Par})$ as the last applied rule, hence

$$\vdash_{\mathsf{sec}} x(y).P \triangleright x{:}(\tau)^{\downarrow_{\mathrm{A}}}_{s} \otimes B \quad \text{and} \quad \vdash_{\mathsf{sec}} \overline{x}(y)Q \triangleright x{:}(\overline{\tau})^{\uparrow_{\mathrm{A}}}_{s} \otimes C \quad \text{with } B \asymp C \tag{25}$$

By Lemma C.5 (1), we can permute the ordering of $(\mathsf{Inf})$ and $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$, so we can assume $x(y).P$ is immediately derived by $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$ as follows.

$$\vdash_{\mathsf{sec}} P \triangleright y{:}\tau \otimes B \quad \text{with} \quad s \sqsubseteq \mathsf{tamp}(B) \tag{26}$$

Next suppose $\overline{x}(y)Q$ is derived by applying first using $(\mathsf{BOut'})$ with $s_0$ being the level of affine output, followed by $(\mathsf{Inf})$ as the last rule. Note that $C$ only has $?$-mode since $y$ has $!$-mode. Hence $\mathsf{tamp}(x : (\tau')^{\uparrow_{\mathrm{A}}}_{s_0} \otimes C) = s_0$, so we can set

$s_0 = s = \mathsf{tamp}(B)$ because $((\tau')^{\uparrow_{\mathrm{A}}}_{s_0}) \sqcup s_0 = ((\tau' \sqcup s)^{\uparrow_{\mathrm{A}}}_{s_0}) = (\overline{\tau})^{\uparrow_{\mathrm{A}}}_{s}$. Now we can assume the following derivation.

$$\frac{\vdash_{\mathrm{sec}} Q \triangleright y{:}\tau' \otimes C'}{\vdash_{\mathrm{sec}} \overline{x}(y)Q \triangleright x{:}(\tau')^{\uparrow_{\mathrm{A}}}_{s} \otimes C'} \quad \text{with } \tau' = \overline{\tau} \sqcup s,\ C' = C \sqcup s \tag{27}$$

Then by Lemma C.5 (3), from (26), we have:

$$\vdash_{\mathrm{sec}} P \triangleright y{:}\tau \sqcup s \otimes (B \sqcup s) \tag{28}$$

We know $B \asymp C$ implies $B \sqcup s \asymp C \sqcup s$. Since $\mathsf{md}(\overline{\tau}) \in \mathcal{M}_!$, we have $(\tau \sqcup s) \odot (\overline{\tau} \sqcup s) = \overline{\tau} \sqcup s$. Hence by applying (Par) and (Res), we have:

$$\vdash_{\mathrm{sec}} (\boldsymbol{\nu}\, y)(P|Q) \triangleright (B \sqcup s) \odot (C \sqcup s) \tag{29}$$

Note that $\mathsf{tamp}((B \sqcup s) \odot (C \sqcup s)) = \mathsf{tamp}(B \sqcup s) = \mathsf{tamp}(B) = s$ [since $\mathsf{md}(C) \sqsubseteq \mathcal{M}_?$; and we can set $B = \uparrow_{\mathrm{A}} B_1 \otimes ?B_2$, so $\mathsf{tamp}(B_1) \sqcup s = \mathsf{tamp}(B_1) = \mathsf{tamp}(A) = s$]. Hence finally applying (Inf) to the above, we have $\vdash_{\mathrm{sec}} (\boldsymbol{\nu}\, y)(P|Q) \triangleright A/x$, as desired. □

### C.3 $\pi^{\mathrm{LAM}}$ with Inflation

Lemma C.5 holds without change, with (4) applied to added actions except for immediately tampering $?_{\mathrm{LM}}/?_{\mathrm{M}}$-actions. We prove the cases which involve immediately tampering actions, starting from affine input/output.

**Proposition C.7** *If $\vdash_{\mathrm{sec}} x(\vec{y}).P|\overline{x}(\vec{y})Q \triangleright A$ is derived using* (Par) *as the last applied rule with $x$ being affine, then $\vdash_{\mathrm{sec}} (\boldsymbol{\nu}\, \vec{y})(P|Q) \triangleright A/x$.*

Proof. As before we assume $\vdash_{\mathrm{sec}} \overline{x}(y)Q \triangleright C' \otimes x{:}\sigma$ is inferred first using (BOut') with $s_0$ being the level of affine output, followed by (Inf) as the last rule, and $\vdash_{\mathrm{sec}} Q \triangleright C \otimes y{:}\tau'$ be the sequent preceding (BOut'). Further we let $\vdash_{\mathrm{sec}} P \triangleright B \otimes y{:}\tau$ so that $B \asymp C'$. We set $\mathsf{tamp}(C) = s_1$ and $\mathsf{tamp}(B) = s \sqsupseteq s_0$. Thus we have $C' \sqcup (s_0 \sqcap s_1) = C$. Let $s' = s \sqcap s_1$. By $s \sqsupseteq s_0$ we have $(B \sqcup s') \asymp (C \sqcup s')$ as well as $\tau' \sqcup s' = \overline{\tau \sqcup s'}$. The rest is the same as Proposition C.6. □

The cases for stateful replication are the same, of which we show unary stateful replication. The case for references is the same.

**Proposition C.8** *If $\vdash_{\mathrm{sec}} !x(\vec{y}).P|\overline{x}(\vec{y})Q \triangleright A$ is derived using* (Par) *as the last applied rule such that the mode of $x$ is $!_{\mathrm{A}}$, then we have $\vdash_{\mathrm{sec}} !x(\vec{y}).P|(\boldsymbol{\nu}\, \vec{y})(P|Q) \triangleright A$.*

Proof. We assume, without loss of generality, the following derivations preceding (Par). For the output side, with $C(x) = (\overline{\tau\rho})^{?_{\mathrm{M}}}_{s}$:

$$\dfrac{\dfrac{\vdash_{\mathrm{sec}} Q \triangleright \mathsf{inf}(C) \otimes y{:}\tau' \otimes z{:}\rho'}{\vdash_{\mathrm{sec}} \overline{x}(yz)Q \triangleright \mathsf{inf}(C)}\ (\mathsf{Inf})}{\vdash_{\mathrm{sec}} \overline{x}(yz)Q \triangleright C}\ (\mathsf{BOut'})$$

For the input side, noting $(\mathsf{In}^{!_{\mathrm{M}}})$ can be permuted over (Inf) as other input prefix rules, we simply assume:

$$(\mathsf{In}^{!_{\mathrm{M}}})\ \frac{\vdash_{\mathrm{sec}} P \triangleright B \otimes y{:}\tau \otimes z{:}\rho}{\vdash_{\mathrm{sec}} !x(yz).P \triangleright x{:}(\tau\rho)^{!_{\mathrm{M}}}_{s_0} \otimes B}$$

We now set:

(1) $\mathsf{tamp}(C) = s_1(\sqsubseteq s)$ and $\mathsf{tamp}(B) = s_2(\sqsupseteq s)$.
(2) $A' = B \odot C$ (hence $A = A' \odot x{:}(\tau\rho)^{!_{\mathrm{M}}}_{s_0}$).

By $B \asymp C$ we have $(B \sqcup s_1) \asymp (C \sqcup s_1)$. Further by $(\overline{\tau\rho})^{?_{\mathrm{M}}}_{s} \sqcup s_1 = (\tau'\rho')^{?_{\mathrm{M}}}_{s}$ we have $\tau' \sqcup s_1 = \overline{\tau \sqcup s_1}$ and $\rho' \sqcup s_1 = \overline{\rho \sqcup s_1}$. Hence we obtain $\vdash_{\mathsf{sec}} (\boldsymbol{\nu}\, yz)(P|Q) \triangleright (B \sqcup s_1) \odot (C' \sqcup s_1)$, that is $\vdash_{\mathsf{sec}} (\boldsymbol{\nu}\, yz)(P|Q) \triangleright A' \sqcup s_1$. Since $B$ and $C'$ do not overlap at compensating names, we have $\mathsf{tamp}(A') = \mathsf{tamp}(B) \sqcap \mathsf{tamp}(C') = s_1 \sqcap s_2 = s_1$. Thus we have $\vdash_{\mathsf{sec}} (\boldsymbol{\nu}\, yz)(P|Q) \triangleright A'$. By (Par) we obtain the required sequent. □

## D. FURTHER DISCUSSIONS ON DCC

### D.1 Illustration of DCC Typing Rules

The presented typing rules for DCC are semantically the same as the original presentation [3] but are extended for the subject reduction to hold (see Remark 4.4 for the discussion on the violation of subject reduction in the original DCC). Together with these differences we give a brief illustration of each typing rule.

- $[Var]$ says that if there is a flow from $x : T$ then it will safely be outputted at the same or higher level. $[Unit]$ says the constant for the unit type can have an arbitrary level (since it does not receive information from anywhere). $[Lam]$ says that if information from $\Gamma$ and $x{:}T$ safely flows via $M$, then the same is true with $x$ substituted for any term of type $T$.
- In the original presentation [3] $[App]$ has the standard shape. Here we allow the type of the agurment to be raised if the answer type is sufficiently high (see the illustration of $[BindM]$ below). We note this application can be semantically representable as $\mathtt{bind}\ x = N\ \mathtt{in}\ Mx$. The extension is done for subject reduction.
- $[Inl]$ is standard. Its dual $[Case]$ says that if $M$ emits information at some secrecy level, the resulting processes should not reveal this level. The original presentation [3] uses the least secrecy level as the level of $M$, which is semantically enough when combined with $[BindM]$. This extension is however necessary for subject reduction.
- $[UnitM]$ says that if information never non-trivially flows down to the level as low as $T$ via $M$, then the same is true if we raise the level of $T$. Its symmetric rule $[BindM]$ says that $M$ can use $N$ at the level higher than originally ensured to be safe, as far as the resulting datum has a sufficiently high level. This is the most interesting rule in DCC, so that we illustrate this rule through examples.

  (1) Starting from $y{:}\mathbb{B}_H, x{:}\mathbb{B}_L \vdash \mathtt{case}\ x^L\ \mathtt{of}\ \mathtt{inl}().\mathtt{inl}()\ \mathtt{or}\ \mathtt{inr}().\mathtt{inr}() : \mathbb{B}_H$, we can infer $y{:}\mathbb{B}_H \vdash \mathtt{bind}\ x = y\ \mathtt{in}\ \mathtt{case}\ x^L\ \mathtt{of}\ \mathtt{inl}().\mathtt{inl}()\ \mathtt{or}\ \mathtt{inr}().\mathtt{inr}() : \mathbb{B}_H$. $x$ is originally low, to which a high-level datum $y$ flows down. However this is still secure since it is in fact used to produce a high-level datum.

  (2) Starting from $w : \mathbb{B}^M \Rightarrow \mathbb{B}^H, y : \mathbb{B}^L \Rightarrow \mathbb{B}^M, x : \mathbb{B}^L, z : \mathbb{B}^H \vdash w(yx) : \mathbb{B}^H$, we infer $w{:}\mathbb{B}^M \Rightarrow \mathbb{B}^H, y{:}\mathbb{B}^L \Rightarrow \mathbb{B}^H, z{:}\mathbb{B}^H \vdash \mathtt{bind}\ x = z\ \mathtt{in}\ w(yx) : \mathbb{B}^H$ (where $M$ is a secrecy level between $H$ and $L$). Because this term uses (in effect) a high-level $z$ to feed $y$ which expects a low-level datum, there is a local

secrecy violation. However, even if $y$ does use the argument (i.e. there is a non-trivial frow from the argument to the result), what $y$ would produce by $yx$ can only be $H$-level information, from which $w$ will again produce $H$-level information, so, at the end of the day, it is safe.

Note (2) is more subtle than (1). In (1), the result of subtitution (regarding `bind` as let) is `case` $y^H$ `of inl().inl() or inr().inr()` : $\mathbb{B}_H$, which is evidently secure (cf. Figure 6 in Section 4). However in (2) the result of substution is $w(yz)$ which contains a locally unsecure subterm $yz$. This aspect of [*BindM*] also leads to an issue in subject reduction, as discussed in Remark 4.4 in Section 4.

—[*Lift*] produces a pointed type to which [*Rec*] can be applied. The distinction between pointed and non-pointed types thus allows separation of total types from possibly diverging types. [*Seq*] waits for a recursion to terminate at a lifted type of some secrecy level, and uses the resulting datum with a unlifted type at a higher level. Since $N$ may diverge, $T'$ should be partial too.

**Remarks D.1** ([*Seq*] and [*BindM*]) While similar in shape, there is a significant difference between [*Seq*] and [*BindM*]. In the original presentation of DCC, [*Seq*] involves cancellation of lift, while [*BindM*] involves cancellation of coersion. The coersion is less significant operationally than lift: for example, in implicit typing systems with subtyping in general, the construct for coercion is turned into subtyping without a constructor (as we do here). The reduction rule $\eta_s M \longrightarrow M$ given in [3] would be understood in this spirit.

## D.2 Subject Reduction in DCC/DCCv

The proof of subject reduction is by the substitution closure of the following form.

**Lemma D.2** *(1) If* $\Gamma \cdot x : (T)_s \vdash M : T'$, $\Gamma \vdash N : T$ *and* $s \sqsubseteq \mathsf{protect}(T')$ *then* $\Gamma \vdash M\{N/x\} : T'$.

*(2) If* $\Gamma \cdot x : T \vdash M : T'$, $\Gamma \vdash \mathtt{lift}(N) : \llcorner T \lrcorner_s$ *and* $T'$ *pointed and* $s \sqsubseteq \mathsf{protect}(T')$ *then* $\Gamma \vdash M\{N/x\} : T'$.

Proof. For (1) we prove the following strengthened property by rule induction of the extended DCC typing rules.

> If $\Gamma \cdot x : (T)_s \vdash M : T'$, $\Gamma \vdash N : T$ and $s \sqsubseteq \mathsf{protect}(T')$ then for each $s'$ we have $\Gamma \vdash M\{N/x\} : (T')_{s'}$.

We show the reasoning for [*App*] and [*Seq*]. Below for simplicity we assume $(T')_{s'} = T'$ (this loses no generality).

- *The last applied rule is* [*App*]. Suppose $\Gamma \cdot x : T \vdash M_1 M_2 : T'$ is inferred from $\Gamma \cdot x : T \vdash M_1 : T_0 \Rightarrow T'$ and $\Gamma \cdot x : T \vdash M_2 : T_0$. Let $\Gamma \vdash N : (T)_s$ and $s \sqsubseteq \mathsf{protect}(T')$. By induction hypothesis $\Gamma \vdash M_2\{N/x\} : (T_0)_s$ as well as $\Gamma \vdash M_1\{N/x\} : T_0 \Rightarrow T'$. By applying [*App*] we obtain $\Gamma \vdash (M_1 M_2)\{N/x\} : T'$.
- *The last applied rule is* [*Seq*]. Suppose $\Gamma \cdot x : T \vdash \mathtt{seq}\ y = M_1\ \mathtt{in}\ M_2 : T'$ is inferred from $\Gamma \cdot x{:}T \cdot y{:}T_0 \vdash M_2 : T'$ and $\Gamma \cdot x{:}T \vdash M_1 : \llcorner T_0 \lrcorner_{s_0}$ with $T'$ pointed and $s_0 \sqsubseteq \mathsf{protect}(T')$, and let $\Gamma \vdash N : (T)_s$ with $s \sqsubseteq \mathsf{protect}(T')$. By induction

hypothesis we have both $\Gamma \vdash M_1\{N/x\} : \llcorner T_0 \lrcorner_{s \sqcup s_0}$ and $\Gamma \cdot y{:}T_0 \vdash M_2\{N/x\} : T'$. By assumption we have $s \sqcup s_0 \sqsubseteq \mathsf{protect}(T')$, hence we can apply [*Seq*] to conclude $\Gamma \cdot x{:}T \vdash \mathtt{seq}\ y = M_1\{N/x\}\ \mathtt{in}\ M_2\{N/x\} : T'$.

Other cases are similar. (2) is easy (note we placed no restriction on $s$, which is enough by the shape of the term $\mathtt{lift}(N)$). □

The subject reduction of DCC is now easily established using Lemma D.2 by structural induction. We show two cases which use the strengthened substitution lemma non-trivially.

—Assume $\Gamma \vdash \lambda x.M : T \Rightarrow T'$, $\Gamma \vdash N : (T)_s$ such that $s \sqsubseteq \mathsf{protect}(T')$ and $(\lambda x.M)N \longrightarrow M\{N/x\}$. By assumption $\Gamma \cdot x{:}T \vdash M : T'$. By Lemma D.2 we have $\Gamma \vdash M\{N/x\} : T'$ hence done.

—Assume $\Gamma \vdash \mathtt{bind}\ x = N\ \mathtt{in}\ M : T'$ and $\mathtt{bind}\ x = N\ \mathtt{in}\ M \longrightarrow M\{N/x\}$. By assumption we have $\Gamma \vdash N : (T)_s$ and $\Gamma \cdot x{:}T \vdash M : T'$ where $s \sqsubseteq \mathsf{protect}(T')$. By Lemma D.2 we are done.

Other cases are easier.

Finally we briefly discuss the subject reduction for DCCv. The proof does not differ from that for DCC, based on (the call-by-value version of) the inflated substitution lemma. As an aside, it is notable that the subject reduction is violated if we introduce [*BindM*] which also acts on partial types (in spite of our presentation in [30]). As an example, let $M \stackrel{\text{def}}{=} \mathtt{bind}\ x = \Omega\ \mathtt{in}\ x$ with $\Omega \stackrel{\text{def}}{=} (\mu x.\lambda y.xy)0$. We can easily check $\vdash M : \mathbb{N}$. However $M \longrightarrow \Omega$ and $\Omega : \mathbb{N}$ is not well-typed. Note we can amend this by combining [*BindM*] with [*Seq*], requiring the partiality of the whole term.

## E. CONSERVATIVITY RESULT FOR THE SMITH CALCULUS

We fitst observe we have only to use *[Var,VarM]* and *[Const]* from Figure 12 by the restriction to the first-order value types. Thus $x := v$ always tampers at $s$ whenever $E(x) = \mathsf{ref}_s\langle \mathbb{N}_s \rangle$. By induction we can show $s$ in $E \vdash c : s\,\mathtt{cmd} \Uparrow_{s'}$ in [57] and $\mathsf{tamp}(E)$ in $E \vdash c : \mathtt{cmd} \Uparrow_{s'}$ coincide (up to the downward subsumption of $s$) by induction.

For while rules, two side conditions look different but they are in fact equivalent. From the condition in our rule we derive:

$$s \sqsubseteq \mathsf{tamp}(E) \sqcap s_0 \ \wedge\ s_0 \sqsubseteq \mathsf{tamp}(E),$$

from which we trivially obtain (1) $s \sqsubseteq \mathsf{tamp}(E)$ and (2) $s_0 \sqcup s = s_0$, the latter being the termination level of the resulting command in [57]. On the other hand, assume given the condition by Smith:

$$s \sqsubseteq \mathsf{tamp}(E) \ \wedge\ s_0 \sqsubseteq \mathsf{tamp}(E).$$

Since we can always raise the termination level $s_0$ of the command in the antecedent by subsumption, we let the raised level be $s_0' \stackrel{\text{def}}{=} s_0 \sqcup s$. The condition is now rewritten for $s_0'$ as:

$$s \sqsubseteq \mathsf{tamp}(E) \sqcap s_0' \ \wedge\ s_0' \sqsubseteq \mathsf{tamp}(E),$$

that is $s \sqsubseteq s'_0 \sqsubseteq \mathsf{tamp}(E)$, with the resulting termination level $s'_0$, which is the condition given in the present paper. The remaining rules are directly mutually translatable.

## F. THE EMBEDDABILITY OF THE EXTENDED SMITH-VOLPANO CALCULUS

### F.1 Proposition 7.7 (subtyping)

$T_1^\circ \leq T_2^\circ$ iff $T_1^\bullet \leq T_2^\bullet$ is direct by definition. Thus we only have to show $T_1 \leq T_2$ iff $T_1^\circ \leq T_2^\circ$, which is proved by rule induction of $T_1 \leq T_2$ defined in Section 7.3. Here we only show the "only if"-direction for the case of $T_i = S_i \stackrel{s_i}{\triangleq} U_i$, $(i = 1, 2)$. Others are similar. Assume $S_2 \leq S_1$, $U_1 \leq U_2$ and $s_2 \sqsubseteq s_1$. Then by induction, $\overline{S_1^\circ} \leq \overline{S_2^\circ}$ and $U_1^\bullet \leq U_2^\bullet$. Note for the input type, the security level is contravariant, while the carried types are covariant. Hence we have: $(S_1 \stackrel{s_1}{\triangleq} U_1)^\circ \stackrel{\text{def}}{=} (\overline{S_1^\circ} U_1^\bullet)^{?_{\mathrm{M}}}_{s_1} \leq (\overline{S_2^\circ} U_2^\bullet)^{?_{\mathrm{M}}}_{s_2} \stackrel{\text{def}}{=} (S_2 \stackrel{s_2}{\triangleq} U_2)$, as required.

### F.2 Proposition 7.8 (coincidence of protection/tampering levels)

By induction on the size of types. For **(1)**, we have $\mathsf{tamp}(S^\bullet) = \mathsf{tamp}((S^\circ)^{\uparrow_{\mathrm{L}}}) = \mathsf{tamp}(S^\circ) = \mathsf{tamp}(\llcorner S \lrcorner^\circ_s)$ (Note $\llcorner S \lrcorner^\circ_s \stackrel{\text{def}}{=} S^\circ$). **(2)** is by simultaneous induction. We use (1). Here we only prove the first part of the statement. Suppose $T = \mathbb{N}_s$. Then $\mathsf{protect}(\mathbb{N}_s) = s$. Also by (1), $\mathsf{tamp}(T^\bullet) = \mathsf{tamp}(\mathbb{N}^\circ_s) = \mathsf{tamp}([\otimes_\omega\,]^{\uparrow_{\mathrm{L}}}_s) = s$. Similarly for the case $T = \llcorner S \lrcorner_s$. Next suppose $T = S \stackrel{s}{\triangleq} U$. Then we have:

$$\begin{array}{rcll}
\mathsf{protect}(S \stackrel{s}{\triangleq} U) & = & \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(U) & \text{(by definition in Section 7.4)} \\
& = & \mathsf{tamp}(\overline{S^\circ}) \sqcap \mathsf{tamp}(U^\bullet) & \text{(by inductive hypothesis)} \\
& = & \mathsf{tamp}((\overline{S^\circ} U^\bullet)^{!_{\mathrm{M}}}_s) & \text{(by Definition 6.1)} \\
& = & \mathsf{tamp}((S \stackrel{s}{\triangleq} U)^\circ) & \text{(by definition of } \circ) \\
& = & \mathsf{tamp}((S \stackrel{s}{\triangleq} U)^\bullet) & \text{(by (1))}
\end{array}$$

The cases $T = S \Rightarrow T$ is similar. If $T = \mathtt{ref}_s(S)$, we have:

$$\begin{array}{rcll}
\mathsf{protect}(\mathtt{ref}_s(S)) & = & \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(S) & \text{(by definition in Section 7.4)} \\
& = & \mathsf{tamp}(\overline{S^\circ}) \sqcap \mathsf{tamp}(S^\bullet) & \text{(by inductive hypothesis)} \\
& = & \mathsf{tamp}(\overline{S^\circ}) \sqcap \mathsf{tamp}(S^\circ) & \text{(by (1))} \\
& = & \mathsf{tamp}([(\overline{S^\circ})^{\uparrow_{\mathrm{L}}} \& S^\circ()^{\uparrow_{\mathrm{L}}}]^{?_{\mathrm{LM}}}_s) & \text{(by definition of } \circ) \\
& = & \mathsf{tamp}(\mathtt{ref}_s(S)^\circ) & \\
& = & \mathsf{tamp}(\mathtt{ref}_s(S)^\bullet) & \text{(by (1))}
\end{array}$$

For **(3)**, it is enough to prove the case when $E$ is a singleton. Let $E = \{x^\delta : T\}$. By (1), if $\delta = \mathtt{w}$, then $\mathsf{tamp}(\{x^{\mathtt{w}} : S\}) = \mathsf{protect}^{\mathcal{E}}(S) = \mathsf{tamp}(\overline{S^\circ}) = \mathsf{tamp}(\{x^{\mathtt{w}} : S\}^\circ)$, as required. The case $\delta = \varepsilon$ is easy by $\mathsf{tamp}(E) = \top = \mathsf{tamp}(E^\circ)$.

### F.3 Proposition 7.10 (well-typedness of the encoding of commands)

The proof is by mechanical induction. We only show *[LamM]* and *[Lam]* for expressions, and *[Seq]*, *[Sub]* and *[Deref]* for commands. *[While]* is discussed in the main section (§7.7). Other cases are similar.

**Case** *[LamM]*. Let $E \cdot x^\delta : S \vdash e : U$. By inductive hypothesis, we have $\vdash_{\mathsf{sec}} \langle e \rangle_u \triangleright E^\circ \otimes x^\delta : S^\circ \otimes u : U^\bullet$. By the condition $s \sqsubseteq \mathsf{tamp}(E) = \mathsf{tamp}(E^\circ)$ we

know $s \sqsubseteq \mathsf{tamp}(E^\circ)$, hence by $(\mathsf{In}^{!_{\mathrm{M}}})$:

$$\vdash_{\mathrm{sec}} !c(xm).\langle e\rangle_m \triangleright E^\circ \otimes c\colon (\overline{S^\circ}U^\bullet)^{!_{\mathrm{M}}}_s \tag{30}$$

Hence by (Out), we have

$$\vdash_{\mathrm{sec}} \overline{u}(c)!c(xm).\langle e\rangle_m \triangleright E^\circ \otimes u\colon (S \overset{s}{\Rightarrow} U)^\bullet,$$

as required.

**Case** *[Lam]*. Suppose in this rule, $M$ has a total type $T'$. If $\langle e\rangle_m$ contains mutable names except $x$, then we cannot type the process (30) because in $(\mathsf{In}^{!_{\mathrm{L}}})$ we only allow $?_{\mathrm{L}}A \otimes ?_{\mathrm{A}}B$ appears as free in its body. But this is guaranteed by the side condition (1) of $(*)$, which says that for all $y$, $E(y) = S$ immutable (note then $\mathsf{md}(E(y)^\circ) \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}\}$).

Next suppose $M$ has a partial type $U$. In this case, we do not abstract the body which contains mutable variables, which is guranteed by the side condition (2) of $(*)$. Thus we can apply $(\mathsf{In}^{!_{\mathrm{A}}})$ to obtain (30).

**Case** *[Seq]*. By assumption, we know $\vdash_{\mathrm{sec}} [\![c_i]\!]_{u_i} \triangleright u_i\colon ()^{p_i} \otimes A_i$ such that $A_i = E_i^\circ$ and $A_1 \odot A_2 = E^\circ$ with $p_i \in \{\uparrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}\}$. First assume $\tau_1 = \Downarrow$, so that $p_1 = \uparrow_{\mathrm{L}}$. We observe we can prefix $[\![c_2]\!]_u$ by a unary linear input regardless of its secrecy level, since a unary linear input does not have the constraint on the secrecy level. Thus by $(\mathsf{In}^{\downarrow_{\mathrm{L}}})$, (Par) and (Res), we infer:

$$\vdash_{\mathrm{sec}} (\boldsymbol{\nu}\, u_1)([\![c_1]\!]_{u_1} \mid u_1.[\![c_2]\!]_{u_2}) \triangleright u_2\colon ()^{p_2}_{s_2} \otimes A_2$$

as required. If on the other hand $\tau = \Uparrow$, then $p_{1,2} = \uparrow_{\mathrm{A}}$. By the side condition and by noting $\mathsf{tamp}(E) = \mathsf{tamp}(A_2)$, we have $s_1 \sqsubseteq s_2 \sqcap \mathsf{tamp}(A_2) = \mathsf{tamp}(u_2\colon ()^{\uparrow_{\mathrm{A}}}_{s_2} \otimes A_2)$. This satisfies the constraint of $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$. Thus the following is well-typed:

$$\vdash_{\mathrm{sec}} u_1.[\![c_2]\!]_{u_2} \triangleright u_2 : ()^{\uparrow_{\mathrm{A}}}_{s_2} \otimes u_1 : ()^{\downarrow_{\mathrm{A}}}_{s_1} \otimes A$$

Hence by (Par) and (Res) we are done.

**Case** *[Sub]*. The only non-trivial case is when we replace $\Downarrow_s$ with $\Uparrow_s$, which we show by induction. Others are straightforward by (Weak), (Weak-w) and Proposition 7.7 for $T$. The base cases ($c \overset{\text{def}}{=} \mathtt{skip}$ and $c \overset{\text{def}}{=} x := v$) are trivial. For induction, for $[\![c_1; c_2]\!]_u$ we note $[\![c_1]\!]_e$ outputs at $e$ linearly), similarly for $[\![\mathtt{if}\ x\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2]\!]_u$. Other cases are direct from the induction hypothesis.

**Case** *[Deref]*. We show the case when $T$ is mutable (the reasoning when $T$ is immutable is easier). Assume $E \vdash z : \mathtt{ref}_s(T)$ and $E \cdot x : T \vdash c : \mathrm{cmd}\,\tau_{s_0}$ with $T$ mutable. By induction hypothesis, we have, with $p \in \{\uparrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}\}$,

$$\vdash_{\mathrm{sec}} [\![c]\!]_u \triangleright E^\circ \otimes x^{\mathrm{w}} : \overline{T^\circ} \otimes u\colon ()^p_{s_0}$$

Note $\mathtt{ref}_s(T)^\circ = [(T^\circ)^{\uparrow_{\mathrm{L}}} \& \overline{T^\circ}()^{\uparrow_{\mathrm{L}}}]^{!_{\mathrm{LM}}}$. Since $T$ is mutable, $x$ should have the write mode. Hence we have

$$\vdash_{\mathrm{sec}} \overline{z}\mathtt{inl}(c)c(x).[\![c]\!]_u \triangleright E/z^\circ \otimes z^{\mathrm{w}} : \overline{\mathsf{ref}_s\langle T^\circ\rangle} \otimes u\colon ()^p_{s_0}$$

Finally noting the above action type is $\langle \mathrm{cmd}\,\tau_s\rangle^E_u$, we are done.

Throughout we set $T = [T_1..T_{n-1}\gamma]$ and $\vec{x} = x_1...x_{n-1}$. Annotated types are determinined from a given DCC sequent up to secrecy levels (which is enough for unique translation). $[x \to y]^T$ and $\overline{x}\langle\vec{y}\rangle^{\vec{\tau}}$ are given in Appendix B.

$$[\![x : T]\!]_u \stackrel{\text{def}}{=} [u \to x]^{T^\circ}$$

$$[\![() : \mathtt{unit}]\!]_u \stackrel{\text{def}}{=} !u(x).\overline{x}$$

$$[\![\lambda x_0.M : T_0 \Rightarrow T]\!]_u \stackrel{\text{def}}{=} !\,u(x_0\vec{x}z).(\boldsymbol{\nu}\, u')([\![M : T]\!]_{u'} \mid \overline{u'}\langle\vec{x}z\rangle^{\overline{T_1^\circ}..\overline{T_{n-1}^\circ}\gamma^\bullet})$$

$$[\![MN : T]\!]_u \stackrel{\text{def}}{=} !\,u(\vec{x}z).(\boldsymbol{\nu}\, mx_0)([\![M : T_0 \Rightarrow T]\!]_m \mid [\![N : T_0]\!]_{x_0} \mid \overline{m}\langle x_0\vec{x}z\rangle^{\overline{T_0^\circ}..\overline{T_n^\circ}\gamma^\bullet})$$

$$[\![\mathtt{inl}(M) : T_1 + T_2]\!]_u \stackrel{\text{def}}{=} !\,u(c).\overline{c}\mathtt{in}_1(m)[\![M : T_1]\!]_m$$

$$[\![\mathtt{case}\, L\, \mathtt{of}\, \mathtt{inl}(x_1^{T_1})M_1\, \mathtt{or}\, \mathtt{inr}(x_2^{T_2})M_2 : T]\!]_u \stackrel{\text{def}}{=} !u(\vec{z}).(\boldsymbol{\nu}\, l)([\![L : T_1 + T_2]\!]_l \mid \mathsf{Sum}\langle l\vec{z}, (x_i)M_i^T\rangle)$$
$$\text{where}\ \ \mathsf{Sum}\langle l\vec{z}, (x_i)M_i^T\rangle \stackrel{\text{def}}{=} \overline{l}(c)c[\&_{i=1,2}(x_i).(\boldsymbol{\nu}\, m)([\![M_i : T]\!]_m \mid \overline{m}\langle\vec{z}^{\overline{T_1}..\overline{T_{n-1}^\circ}\gamma^\bullet}\rangle)]$$

$$[\![\mathtt{bind}\ x = N : T'\ \mathtt{in}\ M : T]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, x)([\![M : T]\!]_u \mid [\![N : T']\!]_x)$$

$$[\![\mathtt{lift}(M) : \llcorner T \lrcorner_s]\!]_u \stackrel{\text{def}}{=} !u(c).\overline{c}(m)[\![M : T]\!]_m$$

$$[\![\mathtt{seq}\, x^{T'} = N\, \mathtt{in}\, M : T]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, n)(\,!u(\vec{z}).\overline{n}(c)c(x).P \mid [\![N : T']\!]_n) \quad ([\![M : T]\!] \stackrel{\text{def}}{=} !u(\vec{z}).P)$$

$$[\![\mu x.M : T]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)([\![M : T]\!]_u \mid [x \to u]^{T^\circ})$$

We omit $\mathtt{inr}(M)$.

Fig. 16. Encoding of Dependency Core Calculus

## G. ENCODINGS

Figures 16, 17 and 18 list the encodings of DCC, DCCv and the extended Smith and Volpano-calculus. The encoding of (call-by-name) DCC uses bound name passing, while DCCv uses free name passing. We can change one form into the other by turning genuine free output to the encoded free output and vice versa. Free name passing makes the encoding concise, while bound name passing tends to give a more uniform encoding and dispenses with the need of subtyping (as discussed in §3.4).

$\langle x\rangle_u \stackrel{\text{def}}{=} \overline{u}\langle x\rangle \qquad \langle n\rangle_u \stackrel{\text{def}}{=} \overline{u}(c)[\![n]\!]_c \qquad \langle \lambda x.M\rangle_u \stackrel{\text{def}}{=} \overline{u}(c)!\,c(xm).\langle M\rangle_m$

$\langle \mathtt{succ}(y)\rangle_u \stackrel{\text{def}}{=} \overline{u}(c)!c(e).\overline{y}(c')c'[\&_i\ .\overline{e}\mathtt{in}_{i+1}]$

$\langle MN\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle M\rangle_m \mid m(a).(\boldsymbol{\nu}\, n)(\langle N\rangle_n \mid n(b).\overline{a}\langle bu\rangle)$

$\langle \mathtt{seq}\ x = N\ \mathtt{in}\ M\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, n)(\langle N\rangle_n \mid n(x).\langle M\rangle_u)$

$\langle \mu x.\lambda y.M\rangle_u \stackrel{\text{def}}{=} \overline{u}(c)(\boldsymbol{\nu}\, x)(P \mid !x(yz).\overline{c}\langle yz\rangle) \qquad (\langle \lambda y.M\rangle_u \equiv \overline{u}(c)P)$

$\langle \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle M\rangle_m | m(c).\overline{c}(e)e[\&_i\langle N_i\rangle_u]) \quad (N_j = N_2 \text{ if } i \geq 2)$

∗ Formally we regard $\overline{x}\langle\vec{y}\rangle$ as its bound output encoding.

Fig. 17. Encoding of Call-by-Value DCC

**(Expression)**

$\langle c\ \mathtt{return}\ e\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)([\![c]\!]_m \mid m.\langle e\rangle_u)$

Others are the same as DCCv defined in Figure 17.

**(Command)**

$[\![\mathtt{skip}]\!]_u \stackrel{\text{def}}{=} \overline{u}$

$[\![x := v]\!]_u \stackrel{\text{def}}{=} \begin{cases} (\boldsymbol{\nu}\, y)(\overline{x}\mathtt{inr}\langle yu\rangle \mid P) & (\langle v\rangle_m \stackrel{\text{def}}{=} \overline{m}(y)P) \\ \overline{x}\mathtt{inr}\langle yu\rangle & (\langle v\rangle_m \stackrel{\text{def}}{=} \overline{m}\langle y\rangle) \end{cases}$

$[\![c_1\,;c_2]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, e)([\![c_1]\!]_e \mid e.[\![c_2]\!]_u)$

$[\![\mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2]\!]_u \stackrel{\text{def}}{=} \begin{cases} (\boldsymbol{\nu}\, y)(P \mid \mathtt{ifzero}\langle y, [\![c_1]\!]_e, [\![c_2]\!]_u\rangle) & (\langle v\rangle_m = \overline{m}(y)P) \\ \mathtt{ifzero}\langle y, [\![c_1]\!]_e, [\![c_2]\!]_u\rangle & (\langle v\rangle_m = \overline{m}\langle y\rangle) \end{cases}$

$[\![\mathtt{while}\ !\,x\ \mathtt{do}\ c]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, e)(\overline{e}\langle u\rangle \mid !f(k).\overline{e}\langle k\rangle$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid !e(k).\overline{x}\mathtt{inl}(c)c(y).\mathtt{ifzero}\langle y, (\boldsymbol{\nu}\, l)([\![c]\!]_l \mid l.\overline{f}\langle k\rangle), \overline{k}\rangle)$

$[\![\mathtt{let}\ x = e\ \mathtt{in}\ c]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle e\rangle_m \mid m(x).[\![c]\!]_u)$

$[\![\mathtt{new}\ x \mapsto v\ \mathtt{in}\ c]\!]_u \stackrel{\text{def}}{=} \begin{cases} (\boldsymbol{\nu}\, xy)(\mathsf{Ref}\langle xy\rangle \mid P \mid [\![c]\!]_u) & (\langle v\rangle_m \stackrel{\text{def}}{=} \overline{m}(y)P) \\ (\boldsymbol{\nu}\, x)(\mathsf{Ref}\langle xy\rangle \mid [\![c]\!]_u) & (\langle v\rangle_m \stackrel{\text{def}}{=} \overline{m}\langle y\rangle) \end{cases}$

$[\![\mathtt{let}\ x = !\,y\ \mathtt{in}\ c]\!]_u \stackrel{\text{def}}{=} \overline{y}\mathtt{inl}(e)e(x).[\![c]\!]_u$

$[\![\Pi_i c_i]\!]_{\vec{u}} \stackrel{\text{def}}{=} \Pi_i [\![c_i]\!]_{u_i}$

$\mathtt{ifzero}\langle x, P, Q\rangle \stackrel{\text{def}}{=} \overline{x}(z)z[\&_i R_i] \qquad (R_0 = P,\ R_i = Q \text{ if } i \geq 1)$

Fig. 18. Encoding of VS-Calculus with Reference and Procedure

$$\frac{}{\vdash_{\mathsf{sec}} \mathbf{0} \triangleright \_}\ (\mathsf{Zero})\quad -$$

$$\frac{\vdash_{\mathsf{sec}} P_i \triangleright A_i \quad (i=1,2) \qquad A_1 \asymp A_2}{\vdash_{\mathsf{sec}} P_1|P_2 \triangleright A_1 \odot A_2}\ (\mathsf{Par})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright A^{x:\tau} \qquad \mathsf{md}(\tau) \in \mathcal{M}_! \cup \{\updownarrow\}}{\vdash_{\mathsf{sec}} (\boldsymbol{\nu}\, x)P \triangleright A/x}\ (\mathsf{Res})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright A^{\text{-}x} \qquad \mathsf{md}(\tau) \in \mathcal{M}_? \cup \{\updownarrow\}}{\vdash_{\mathsf{sec}} P \triangleright A \otimes x:\tau}\ (\mathsf{Weak})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright \vec{y}{:}\vec{\tau} \otimes \uparrow_{\mathrm{L}} A^{\text{-}x} \otimes \uparrow_{\mathrm{A}} ?\, B^{\text{-}x}}{\vdash_{\mathsf{sec}} x(\vec{y}).P \triangleright (x:(\vec{\tau})^{\downarrow_{\mathrm{L}}} \to A) \otimes B}\ (\mathsf{In}^{\downarrow_{\mathrm{L}}})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright \vec{y}{:}\vec{\tau} \otimes ?_{\mathrm{L}} A^{\text{-}x} \otimes ?_{\mathrm{A}} B^{\text{-}x}}{\vdash_{\mathsf{sec}!} x(\vec{y}).P \triangleright (x:(\vec{\tau})^{!_{\mathrm{L}}} \to A) \otimes B}\ (\mathsf{In}^{!_{\mathrm{L}}})$$

$$\frac{s \sqsubseteq \mathsf{tamp}(A) \qquad \vdash_{\mathsf{sec}} P \triangleright \vec{y}{:}\vec{\tau} \otimes \uparrow_{\mathrm{A}} ?\, A^{\text{-}x}}{\vdash_{\mathsf{sec}} x(\vec{y}).P \triangleright x:(\vec{\tau})_s^{\downarrow_{\mathrm{A}}} \otimes A}\ (\mathsf{In}^{\downarrow_{\mathrm{A}}})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright \vec{y}{:}\vec{\tau} \otimes ?_{\mathrm{L}} ?_{\mathrm{A}} ?_\mu^{\varepsilon} A^{\text{-}x}}{\vdash_{\mathsf{sec}!} x(\vec{y}).P \triangleright x:(\vec{\tau})^{!_{\mathrm{A}}} \otimes A}\ (\mathsf{In}^{!_{\mathrm{A}}})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright C^{\vec{y}:\vec{\tau}} \quad C \asymp x:(\vec{\tau})_s^{p_{\mathrm{O}}}}{\vdash_{\mathsf{sec}} \overline{x}(\vec{y})P \triangleright C/\vec{y} \odot x:(\vec{\tau})_s^{p_{\mathrm{O}}}}\ (\mathsf{BOut})\quad p_{\mathrm{O}} \neq ?_{\mathrm{M}}$$

$$\frac{s \sqsubseteq \mathsf{tamp}(A) \qquad \vdash_{\mathsf{sec}} P \triangleright \vec{y}{:}\vec{\tau} \otimes ?\, A^{\text{-}x}}{\vdash_{\mathsf{sec}} !x(\vec{y}).P \triangleright x:(\vec{\tau})_s^{!_{\mathrm{M}}} \otimes A}\ (\mathsf{In}^{!_{\mathrm{M}}})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright C^{\vec{y}:\vec{\tau}} \quad C \asymp x:(\vec{\tau})_s^{?_{\mathrm{M}}}}{\vdash_{\mathsf{sec}} \overline{x}(\vec{y})P \triangleright C/\vec{y} \odot x^{\mathtt{w}}:(\vec{\tau})_s^{?_{\mathrm{M}}}}\ (\mathsf{BOut}^{?_{\mathrm{M}}})$$

$$\frac{\vdash_{\mathsf{sec}} P_i \triangleright \vec{y}_i:\vec{\tau}_i \otimes \uparrow_{\mathrm{L}} A^{\text{-}x} \otimes \uparrow_{\mathrm{A}} ?\, B^{\text{-}x}}{\vdash_{\mathsf{sec}} x[\&_i(\vec{y}_i).P_i] \triangleright (x:[\&_i \vec{\tau}_i]_s^{\downarrow_{\mathrm{L}}} \to A) \otimes B}\ (\mathsf{Bra}^{\downarrow_{\mathrm{L}}})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright \vec{y}_i:\vec{\tau}_i \otimes ?_{\mathrm{L}} A^{\text{-}x} \otimes ?_{\mathrm{A}} B^{\text{-}x}}{\vdash_{\mathsf{sec}!} x[\&_i(\vec{y}_i).P_i] \triangleright (x:[\&_i \vec{\tau}_i]^{!_{\mathrm{L}}} \to A) \otimes B}\ (\mathsf{Bra}^{!_{\mathrm{L}}})$$

$$\frac{s \sqsubseteq \mathsf{tamp}(A) \qquad \vdash_{\mathsf{sec}} P_i \triangleright \vec{y}_i:\vec{\tau}_i \otimes \uparrow_{\mathrm{A}} ?\, A^{\text{-}x}}{\vdash_{\mathsf{sec}} x[\&_i(\vec{y}_i).P_i] \triangleright x:[\&_i \vec{\tau}]_s^{\downarrow_{\mathrm{A}}} \otimes A}\ (\mathsf{Bra}^{\downarrow_{\mathrm{A}}})$$

$$\frac{\vdash_{\mathsf{sec}} P_i \triangleright \vec{y}_i:\vec{\tau}_i \otimes ?_{\mathrm{L}} ?_{\mathrm{A}} ?_\mu^{\varepsilon} A^{\text{-}x}}{\vdash_{\mathsf{sec}!} x[\&_i(\vec{y}_i).P_i] \triangleright x:[\&_i \vec{\tau}_i]^{!_{\mathrm{A}}} \otimes A}\ (\mathsf{Bra}^{!_{\mathrm{A}}})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright C^{\vec{y}:\vec{\tau}_j} \quad C \asymp x:[\otimes \vec{\tau}_i]_s^{p_{\mathrm{O}}}}{\vdash_{\mathsf{sec}} \overline{x}\mathtt{in}_j(\vec{y})P \triangleright C/\vec{y} \odot x:[\otimes \vec{\tau}_i]_s^{p_{\mathrm{O}}}}\ (\mathsf{BSel})\quad p_{\mathrm{O}} \notin \{?_{\mathrm{M}}, ?_{\mathrm{LM}}\}$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright A \otimes x:\tau \quad \mathsf{md}(\tau) \in \{?_{\mathrm{LM}}, ?_{\mathrm{M}}\}}{\vdash_{\mathsf{sec}} P \triangleright A \otimes x^{\mathtt{w}}:\tau}\ (\mathsf{Weak}\text{-}\mathtt{w})$$

$$\frac{\text{if } \mathsf{md}(\tau) \in \{!_{\mathrm{LM}}, !_{\mathrm{M}}\} \text{ then } \delta = \mathtt{w} \text{ else } \delta = \varepsilon}{\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \triangleright x:\mathsf{ref}_s\langle\tau\rangle \otimes y^{\delta}:\overline{\tau}}\ (\mathsf{Ref})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright A^{c:(\tau)^{\downarrow_{\mathrm{L}}}}}{\vdash_{\mathsf{sec}} \overline{x}\mathtt{inl}(c)P \triangleright x^{\delta}:\overline{\mathsf{ref}_s\langle\tau\rangle} \otimes A/c}\ (\mathsf{Read})$$

$$\frac{\vdash_{\mathsf{sec}} P \triangleright A^{vc:\tau()^{\uparrow_{\mathrm{L}}}}}{\vdash_{\mathsf{sec}} \overline{x}\mathtt{inr}(vc)P \triangleright x^{\mathtt{w}}:\overline{\mathsf{ref}_s\langle\tau\rangle} \otimes A/vc}\ (\mathsf{Write})$$

$$\frac{s \sqsubseteq s' \qquad \text{if } \mathsf{md}(\tau) \in \{!_{\mathrm{LM}}, !_{\mathrm{M}}\} \text{ then } \delta = \mathtt{w} \text{ else } \delta = \varepsilon}{\vdash_{\mathsf{sec}} [x \to y]^{\mathsf{ref}_s\langle\tau\rangle} \triangleright x:\mathsf{ref}_s\langle\tau\rangle \to y^{\delta}:\overline{\mathsf{ref}_{s'}\langle\tau\rangle}}\ (\mathsf{CRef})$$

Fig. 19. Summary of Secrecy Typing for $\pi^{\mathsf{LAM}}$ (with bound output)