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 π-calculus is a process calculus in which we can compositionally represent dynamics of various algorithmic entities, among others major programming constructs, by decomposing them into name passing. This work reports our experience in using a linear/affine typed π-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 π-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 London, Huxley Building, 180 Queen's Gate, London SW7 2BZ, UK.

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 [53; 61; 9]. 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, for a piece of code, $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$ 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 [15; 68; 43; 67; 71]. 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 π-Calculus. The π-calculus [48; 46; 13; 28] is an extension of CCS [45] based on name passing. A basic form of its dynamics can be written down as the following reduction.

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

Here a vector of names $\vec{w}$ are communicated, via channel x, to an input process, resulting in a new process after 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. This embeddability is due to the fine-grained nature of the dynamics and algebra of the π-calculus, which can readily decompose those of language constructs. We are thus prompted by the following question: can we have a foundational type structure for this calculus, similar to those for the λ-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. [46; 41; 59; 72], have shown that, even though operational encodings of diverse typed calculi into the π-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 that 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 π-calculus [46; 59; 24; 72; 41] and game semantics [4; 5; 37; 31], the present authors, with Martin Berger, recently reported [10; 74] that a class of type structures for the π-calculus which precisely capture existing type structure for programming languages do exist, allowing, among others, fully abstract translation of prominent functional typed calculi. In our initial work [10; 74], we have presented two type disciplines for the π-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, [10; 74] have shown that PCF and strongly normalising λ-calculi are fully abstractly embeddable in the affine and linear π-calculus, respectively. In spite of faithfulness of embeddings, the shape of types and type disciplines 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 λ-calculi are embeddable into a single typing system. Starting from these two central notions, a family of basic classes of typed behaviours are identified, each allowing a precise embedding of programming languages. Some of the these embedding results are reported in [12; 26].

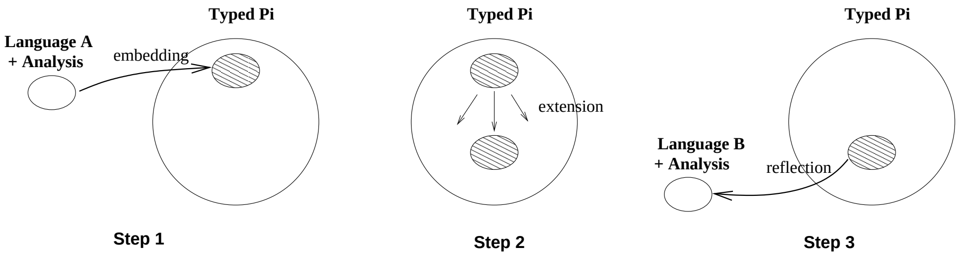

Fig. 1. Type-based Program Analysis using the π-Calculus

Using the π-Calculus. The uniform embeddability of typed programming languages in the π-calculus with linear/affine typing suggests its potential usage as an integrated basis for their description, reasoning and analysis. As a possible area for experimenting with such possibility, we consider *type-based program analyses*, which are program analyses which use 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 into the typed π-calculus, and there is a type-based analysis on that language, then we can transfer the 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 should be established, which 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 π-calculus. This is done by reflecting the analysis in the π-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 π-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 (in the present inquiry, the use of duality-based types plays the key rôle for enabling this process). 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 π-calculus for type-based program analyses, taking secure information flow analysis (also called secrecy analysis) [3; 21; 57; 58; 64; 65; 66; 70] as an application domain. In this analysis, we use a typing system to ensure the safety of information flow in a given program, in the sense that a high-level (secure) data never flows down to low-level (public) channels. Information flow analysis needs precise understanding

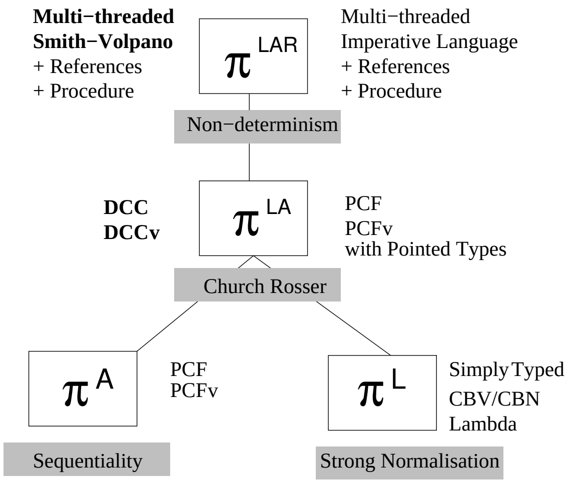

Fig. 2. A Family of Linear/Affine π-Calculi

of observable behaviour of program phrases and their interplay, because of the existence of covert channels [16; 42]. In the π-calculus representation of programming languages, computational dynamics in a program is decomposed into name passing interaction, making explicit dynamics associated with each language construct, as well as the notion of observables. For these reasons, the π-calculus may be used as an effective tool for analysing subtle information flow in program phrases, as studied in our earlier work [27]. Further, in many type-based information flow analyses, distinction between totality (termination) and partiality (potential divergence) in types is crucial, both in functional [3] and imperative [70] settings, strongly suggesting its connection to linear/affine type structures. A uniform treatment of diverse elements in programming languages, including call-by-name, call-by-value, pure functions, stateful computation, sequentiality and concurrency, is another motivation for using the π-calculus.

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

- A typed π-calculus for secure information flow based on linear/affine type disciplines, which enjoys a noninterference property. We present its purely functional version as well as its extension to concurrency and state.
- The sound embeddability of the dependency core calculus (DCC) [3] in the secrecy-enhanced linear/affine π-calculus, and a simple operational proof of its noninterference property through the embedding. We also present a novel call-by-value version of DCC, whose soundness can again be shown through the embeddability into the secure π-calculus.
- The introduction of a new secrecy type discipline for a basic concurrent imperative programming language with general references and higher-order procedures.

Its embeddability in the linear/affine π-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 π-calculus with a specific type structure. "L", "A" and "R" mean linear, affine and state, respectively, where the stateful calculus also incorporates, in the present study, concurrency. The right-hand side of the box shows systems we can embed in the basic typed π-calculus. The left-hand side shows secure languages we can embed in the secure version of the π-calculus. The grey box shows a basic property satisfied by the calculus.

Outline. This paper is a full version of [32], with complete definitions and detailed proofs. The emphasis is on illustrating, through concrete examples, how the typed π-calculus can be used for developing and justifying a non-trivial type-based analysis of programming languages. In this spirit, the presentations of the call-by-value version of DCC and the extended version of Smith-Volpano language are considerably simplified in comparison with [32], through the reformulation of encodings and simplification of the stateful extension of the linear/affine π-calculus itself. The present version also gives more comparisons with related work, including a formal conservativity result with respect to Smith's recent secure imperative language [65].

In the remainder, Section 2 introduces the π-calculus with a linear/affine type discipline, integrating type disciplines presented in [10; 74]. Section 3 studies a type-based secrecy analysis for the linear/affine π-calculus. Section 4 embeds DCC in the secure π-calculus and develops its call-by-value version, both justified via the secure π-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 π-calculi and their applications. Stephen Zdancewic pointed out several mistakes in the conference version for which we are particularly grateful. Discussions with Andrew Myers, Francois Pottier, Geoffrey Smith and Vincent Simonet deepened our understanding of secrecy analysis. Discussions with Chris Hankin have broadened our perspective on the use of the π-calculus for program analyses, part of which is reflected in Introduction. The first author is partially supported by EPSRC grant GR/S55545. The second author is partially supported by EPSRC grant GR/R33465 and GR/S55538.

2. LINEAR/AFFINE π-CALCULUS

2.1 Processes (1): Syntax

Throughout the present study we shall use the asynchronous version of the π-calculus [13; 28] extended with branching/selections [23; 27; 29; 20]. Fix a countable set of *names* (also called *channels*), ranged over by $x, y, \ldots, a, b, \ldots$. We write $\vec{y}$ for a finite, possibly null, string of names. We let $P, Q, \ldots$ range over processes, with subscripts etc. as necessary, whose grammar is given by:

$$\begin{array}{rcl} P & ::= & x(\vec{y}).P \mid\ !x(\vec{y}).P \mid \overline{x}\langle\vec{y}\rangle \mid x[\&_{i\in I}(\vec{y_i}).P_i] \mid\ !x[\&_{i\in I}(\vec{y_i}).P_i] \mid \overline{x}\mathtt{in}_i\langle\vec{y}\rangle \\ & \mid & P|Q \mid (\boldsymbol{\nu}\, x)P \mid \mathbf{0} \end{array}$$

In the grammar, $x(\vec{y}).P$ is an input, which would receive a vector of names via a channel x and instantiating them into (pairwise distinct) formal parameters $\vec{y}$. $(\vec{y})$ is a binder over P (i.e. it binds the free occurrences of $\vec{y}$ in P). $!x(\vec{y}).P$ is its replicated version, again $(\vec{y})$ acting as a binder. $\overline{x}\langle\vec{y}\rangle$ is a message to a channel x carrying names $\vec{y}$.

The construct $x[\&_{i\in I}(\vec{y_i}).P_i]$ is called *branching*, where $(\vec{y_i})$ is a binder over P_i for each i. $!x[\&_{i\in I}(\vec{y_i}).P_i]$ is its replicated version, again $(\vec{y_i})$ being a binder over P_i for each i. In $x[\&_{i\in I}(\vec{y_i}).P_i]$ and $!x[\&_{i\in I}(\vec{y_i}).P_i]$, we assume the indexing set I to be a countable or finite non-empty set, including booleans $\mathbb{B}$ and natural numbers $\mathbb{N}$ (in an infinitary branching, we assume each summand is of finite height: see Remark 2.2 for discussions on the use of infinite branching). $\overline{x}\mathtt{in}_i\langle\vec{y}\rangle$ is *selection*, which is a message that selects the i-th branch from the branching and communicates names $\vec{y}$. While all programming languages we shall treat in the present paper can be operationally encodable into the branch-free syntax, the use of branching makes the syntax and dynamics of processes directly reflect the type structure introduced later, just as the case construct is useful in typed λ-calculi with sum types. Among others its use enables a clean type-directed encoding of various language constructs.

$P|Q$ is the parallel composition of P and Q. $(\boldsymbol{\nu}\, x)P$ is P such that its free occurrences of x, if any, become private to the processes. $(\boldsymbol{\nu}\, x)$ is a binder for free occurrences of x in P. $\mathbf{0}$ denotes the inaction, indicating there is no process. The sets of free and bound names, written $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$, as well as the alpha equality $\equiv_\alpha$, standard. We always assume the standard bound name convention.

We use the following abbreviations throughout the paper.

—$(\boldsymbol{\nu}\, y_1..y_n)P$ stands for $(\boldsymbol{\nu}\, y_1)..(\boldsymbol{\nu}\, y_n)P$.

—We omit the empty parameters, e.g. $x.P$ stands for $x().P$ and $\overline{x}\mathtt{in}_i$ for $\overline{x}\mathtt{in}_i\langle\rangle$.

—$\overline{x}(\vec{y})P$, called *bound output*, stands for $(\boldsymbol{\nu}\, \vec{y})(\overline{x}\langle\vec{y}\rangle|P)$ (assuming names in $\vec{y}$ are pairwise distinct). This process asynchronously sends new names $\vec{y}$ local to P.

—Similarly $\overline{x}\mathtt{in}_i(\vec{y})P$, called *bound selection*, stands for $(\boldsymbol{\nu}\, \vec{y})\overline{x}\mathtt{in}_i(\vec{y})P$, again assuming names in $\vec{y}$ are pairwise distinct.

—We write $x[(\vec{y_1}).P_1\&(\vec{y_2}).P_2]$ for a binary branching and $\overline{x}\mathtt{inl}\langle\vec{v}\rangle$ or $\overline{x}\mathtt{inr}\langle\vec{v}\rangle$ for binary selections. We also often omit the indexing set I from $x[\&_{i\in I}(\vec{y_i}).P_i]$, writing simply $x[\&_i(\vec{y_i}).P_i]$.

2.2 Processes (2): Structural Rules and Reduction

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 $x \notin \mathsf{fn}(Q)$.

The dynamics of processes is defined as the reduction relation $\longrightarrow$ using the structural congruence. First, there are two rules for the unary name passing.

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

where $P\{\vec{v}/\vec{y}\}$ is a simultaneous substitution of v_i for free occurrences of x_i, assuming $\vec{x}$ and $\vec{v}$ have the same length. The dynamics of branching involves selection of one branch, discarding the remaining ones, as well as name passing.

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

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

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

The multi-step reduction is given by $\twoheadrightarrow \stackrel{\text{def}}{=} \equiv \cup \longrightarrow^*$. Appendix E lists the full definition of the structure rules and the reduction rules.

Simple examples of processes and their reduction follow.

Example 2.1 (processes and reduction) In the following we show a couple of processes and their reduction, which in fact correspond to the process encoding of programs which we formally define later. For a program M, we write $[\![M]\!]_u$ for its encoding, where u represents the name at which M is located.

(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. This agent is invoked through its free channel u by sending a channel c to which output should be sent. Here c plays the rôle of a continuation of interaction. As an example of the reduction, we have $[\![2]\!]_u|\overline{u}\langle e\rangle \longrightarrow [\![2]\!]_u|\overline{e}\mathtt{in}_2$. Similarly we can define a *unit agent*, $[\![()]\!]_u \stackrel{\text{def}}{=} !u(c).\overline{c}$, which always returns just an activation.

(2) The process $\overline{u}(c)c[\&_{n\in\mathbb{N}}.\overline{e}\mathtt{in}_{n+1}]$ behaves as the successor of the natural number agent above. This agent invokes the natural number with continuation c; then if it's i-th branch is selected via c, it emits the answer $i+1$ via e. In the following reduction, we detail the use of $\equiv$ for illustration.

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

The essence of this encoding lies in a precise depiction of the functional behaviour as an interacting process: one may as well consider reductions as above as microscopic abstraction of interactions a program has with its environment.

(3) Let $\mathtt{fw}\langle xy\rangle \stackrel{\text{def}}{=} !x(c).\overline{y}\langle c\rangle$. This is a *forwarder*, which, when asked at x, simply forwards the value to y, thus linking two locations x and y. Having this agent in-between does not change the whole behaviour. For example:

$$[\![()]\!]_y|\mathtt{fw}\langle xy\rangle|\overline{x}\langle e\rangle \;\longrightarrow\; [\![()]\!]_y|\mathtt{fw}\langle xy\rangle|\overline{y}\langle e\rangle \;\longrightarrow\; [\![()]\!]_y|\mathtt{fw}\langle xy\rangle|\overline{e}$$

which is the same as $[\![()]\!]_y|\overline{y}\langle e\rangle \longrightarrow [\![()]\!]_y|\overline{e}$ save some internal reductions. There is an agent which has precisely the same functionality but which only uses only bound output, called *copycat* [37; 4], defined as (in the present case) $[x \rightarrow y] \stackrel{\text{def}}{=} !x(c).\overline{y}(c')c'.\overline{c}$. One can check that this agent induces essentially the same transformation as the forwarder above.

(4) $\overline{f}(ac)([\![1]\!]_a|\mathtt{fw}\langle ce\rangle)$ represents an open λ-term $f1$ with f of type $\mathbb{N}\Rightarrow\mathbb{N}$. 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 in the environment, which it simply forwards to e. We can check, with $[\![\mathsf{succ}]\!]_f \stackrel{\text{def}}{=} !f(ue).\overline{u}(c)c[\&_{n\in\mathbb{N}}.\overline{e}\mathtt{in}_{n+1}]$:

$$[\![\mathsf{succ}]\!]_f|\overline{f}(ac)([\![1]\!]_a|\mathtt{fw}\langle ce\rangle) \twoheadrightarrow [\![\mathsf{succ}]\!]_f|\overline{e}\mathtt{in}_2|(\boldsymbol{\nu}\, a)[\![1]\!]_a$$

Note $(\boldsymbol{\nu}\, a)[\![1]\!]_a$ remains as a garbage without any further potential interaction, so that the resulting process is in fact the same thing as $[\![\mathsf{succ}]\!]_f|\overline{e}\mathtt{in}_2$.

(5) Let $\Omega_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(\mathtt{fw}\langle uy\rangle|\mathtt{fw}\langle yu\rangle)$. This is called an *omega agent* which immediately diverges after the initial invocation at u; we can check $\Omega_u|\overline{u}\langle e\rangle \longrightarrow\longrightarrow \Omega_u|\overline{u}\langle e\rangle \longrightarrow\longrightarrow \Omega_u|\overline{u}\langle e\rangle \longrightarrow\longrightarrow ....$

Remark 2.2 (infinitary branching) Infinite branching plays a significant rôle for tractable embedding of programming languages in the present paper. Their use is restricted to those cases where we can operationally encode using finite branching (thus in effect without branching, cf. [46; 28]). An unrestricted use of infinitary branching can lead to the following two kinds of pathologies.

(1) A process may have no finite bound in its height and its set of free names.

(2) A process may represent an uncomputable function, which leads to the lack of definability in embedding results, cf. [11].

(1) makes it necessary to use well-founded induction in structural induction, and to make the whole collection of names a proper class. We eliminate the former possibility by assuming there is a finite upper-bound in the height; for the latter each of the type disciplines introduced in the present paper automatically guarantees the finiteness of free names. (2) can be eliminated by restricting indices of selections so that they are computable (with respect to indices of enclosing branches), even though its existence or elimination does not affect the subsequent technical development.

2.3 Linear/Affine Typing (1) Action Modes and Channel Types

Many notions of types for the π-calculus have been studied, ramifying Milner's sorting [47]. The key elements common to most of the existing types for the π-calculus (including the one which we are going to use in the present inquiry) may be summarised as follows. In each type discipline:

(1) To each typable process, say P, we assign a type A. The type assignment ensures P has a certain property represented by A.

(2) Type assignment is compositional: for example, for parallel composition, if P has a type A, Q has a type B and $A \odot B$ is defined for a partial operator $\odot$

with the resulting value C, then we assign C to $P|Q$. If $A \odot B$ is not defined, the composition is not allowed. Similarly for other operators.

A type A is often simply a finite map from names to what we often call *channel types*, but it can be more complex. In this way, types control in what way a process can be constructed and, as a result, they guarantee a certain property of a process.

The type structure we shall introduce below is based on two concepts, *linearity* and *affinity*, the notions which have long been studied in programming language theories starting from Linear Logic [20]. Concretely, it integrates the linear type discipline in [74] and the affine one in [10], allowing fine-grained mixture of linear (or convergent) computation and affine (or possibly divergent) computation in a single type structure, which is crucial in the secrecy analysis in the next section.

Following this general framework, the linear/affine type discipline we shall use in the present paper first introduces *channel types*, which are types assigned to channels. Types assigned to processes, called *action type*, are then finite maps from names to channel types augmented with causality edges for linearly typed channels. In the following we introduce the fundamental elements of this type discipline one by one.

2.3.1 Action Modes. Channel types in the linear/affine type discipline use *action modes* [10; 24; 29; 74] as its essential element. Action modes prescribe different ways processes interact at channels, and are given as follows.

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

We also use the mode $\updownarrow$. We let $p, p', \ldots$ range over action modes. In the above table, the modes in the left column are *input modes* while those in the right are *output modes.* $p_{\rm I}$ (resp. $p_{\rm O}$) denotes input (resp. output) modes. The pair of modes in each row are *dual* to each other, writing $\overline{p}$ for the dual of p. We often write $\mathbf{?}$ to denote either $\mathbf{?}_{\rm L}$ or $\mathbf{?}_{\rm A}$. Similarly for $\mathbf{!}, \uparrow, \downarrow$. Some illustration of action modes follow.

—A "$\downarrow$" mode is associated with an input (e.g. x in $x(y).P$). In more detail, $\downarrow_{\rm L}$ indicates a linear input, at which an interaction takes place *precisely once*; while $\downarrow_{\rm A}$ indicates an affine input, at which an interaction takes place *at most once*.

—A "!" mode is associated with a replicated input (e.g. x in $!x(y).P$). In more detail, $\mathbf{!}_{\rm L}$ indicates a convergent replication (a replication at which no divergence occurs and, hence, ensures an answer is returned), while $\mathbf{!}_{\rm A}$ indicates a possibly divergent replication (so that invocation at that channel may not be returned). For example, a channel u in Ω_u in Example 2.1 (5) has the mode $\mathbf{!}_{\rm A}$.

—"$\updownarrow$" indicates that a channel is no longer available for *further* composition with the outside.

For example, if $\overline{x}\langle\vec{y}\rangle$ is composed with $x(\vec{y}).P$, then x in the message has a $\uparrow$ -mode, and if it is to be composed with $!x(\vec{y}).P$, then x should have a ?-mode. As to $\updownarrow$, if $x.\mathbf{0}$ has the $\downarrow_{\rm L}$-mode and $\overline{x}$ has the $\uparrow_{\rm L}$-mode, then $x.\mathbf{0} \mid \overline{x}$ has the $\updownarrow$-mode at x. The $\updownarrow$-mode at x indicates that the process $x.\mathbf{0} \mid \overline{x}$ cannot be composed with any process that has x as a free name.

2.3.2. Channel Types. Next we define *channel types* by the following grammar.

$$\tau \ ::= \ \tau_{\text{I}} \mid \tau_{\text{O}} \mid \updownarrow \qquad \tau_{\text{I}} \ ::= \ (\vec{\tau})^{p_{\text{I}}} \ \mid \ [\&_{i\in I}\vec{\tau_i}]^{p_{\text{I}}} \qquad \tau_{\text{O}} \ ::= \ (\vec{\tau})^{p_{\text{O}}} \ \mid \ [\oplus_{i\in I}\vec{\tau_i}]^{p_{\text{O}}}$$

$(\vec{\tau})^{p_{\text{I}}}$ and $(\vec{\tau})^{p_{\text{O}}}$ are *unary input and output types*, respectively, while $[\&_i\vec{\tau_i}]^p$ and $[\oplus_i\vec{\tau_i}]^p$ are *branching and selection types*. We sometimes write $[\vec{\tau}_1\&\vec{\tau}_2]^p$ or $[\vec{\tau}_1 \oplus \vec{\tau}_2]^p$ for binary branching or selection type. $\mathsf{md}(\tau)$ denotes the outermost mode of τ except we set $\mathsf{md}(\updownarrow) = \updownarrow$. The *dual of* τ, written $\overline{\tau}$, is the result of dualising all action modes and exchanging & and $\oplus$ in τ.[1]

The following well-formedness condition is an integral part of the present type discipline, largely coming from game semantics [37; 4] except for the distinction between linearity and affinity. Among others they demand that names used for input carry only output names and vice versa (IO-alternation) and that a replicated input channel carries a unique linearly/affinely used channel (the unique answer condition). We use the following notations.

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

Definition 2.3 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 τ_i is well-formed and, moreover, $\mathsf{md}(\tau_i) \in \mathcal{M}_{\mathbf{?}}$ for each i. Dually when $p \in \mathcal{M}_{\uparrow}$.

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

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

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

Remark 2.4 (well-formedness of channel types)

(1) (C1) means that a linear or affine output carries names of some servers (which may be considered as data carried in a message). Dually a linear or affine input receives these names and can ask back through them.

(2) (C2) says a channel of a linear server includes a unique linear/affine output channel, as well as some channels for querying back servers (the latter may be considered as arguments of an invocation). (C3) is similar to (C2), saying in addition that the unique output channel should now only be affine.

(3) By (C2) and (C3), a linear output $\uparrow_{\text{L}}$ can only be carried by a linear replicated input $\mathbf{!}_{\text{L}}$ (and dually). This is crucial for consistently integrating linearity and affinity so that an invocation at linear replication eventually leads to a linear output, as we now illustrate using an example. Recall Ω_u in Example 2.1 (5). Because computation at x and y do not terminate, x, y should have mode $\mathbf{!}_{\text{A}}$ and $\mathbf{?}_{\text{A}}$. If we type c and c' with $\downarrow_{\text{L}}$ and $\uparrow_{\text{L}}$ (contradicting C3), then $\overline{x}(c)c.\overline{w}$ can never receive an input at c, i.e. we lose linearity at c.

[1] In accordance with the discussion in Remark 2.2, we may as well set a finite bound on the set of summands in a branching/selectionon type.

Example 2.5 (channel types)

(1) $()^{\uparrow_{\mathrm{L}}}$ is a type indicating an output without carrying any value which takes place exactly once; $()^{\uparrow_{\mathrm{A}}}$ 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 $\mathbb{N}^{\bullet} = [\oplus_{i\in\mathbb{N}}]^{\uparrow_{\mathrm{L}}}$. This type represents an output of a natural number 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. Its affine version, $([\oplus_{i\in\mathbb{N}}]^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}$, is the same except it allows the possibility of not returning a number. We have $\overline{\mathbb{N}^{\bullet}} = [\&_{i\in\mathbb{N}}]^{\downarrow_{\mathrm{L}}}$ and $\overline{\mathbb{N}^{\circ}} = ([\&_{i\in\mathbb{N}}]^{\downarrow_{\mathrm{L}}})^{?_{\mathrm{L}}}$. The latter is a type which inquires at a replicated channel carrying a unique name, and via that name receives a natural number precisely once.

(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 because of the well-formed condition (C3).

2.4 Linear/Affine Typing (2) 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 $x : \tau \rightarrow y : \tau'$ such that either (1) $\mathsf{md}(\tau) = \downarrow_{\mathrm{L}}$ and $\mathsf{md}(\tau') = \uparrow_{\mathrm{L}}$ or (2) $\mathsf{md}(\tau) = !_{\mathrm{L}}$ and $\mathsf{md}(\tau') = ?_{\mathrm{L}}$.

We write $x \rightarrow y$ if $x : \tau \rightarrow y : \tau'$ for some τ and τ', in a given action type. If x occurs in A and for no y we have $y \rightarrow x$ then we say x is *active in* A. $|A|$ (resp. $\mathsf{fn}(A)$, $\mathsf{md}(A)$) denotes the set of nodes (resp. names, modes) in A. For example, if $A = x : \tau_1 \rightarrow y : \tau_2, z : \tau_3$, then x, z are active; $|A| = \{x : \tau_1, y : \tau_2, z : \tau_3\}$; $\mathsf{fn}(A) = \{x, y, z\}$; and $\mathsf{md}(A) = \{\mathsf{md}(\tau_1), \mathsf{md}(\tau_2), \mathsf{md}(\tau_3)\}$. We often write $x : \tau \in A$ for $x : \tau \in |A|$, and write $A(x)$ for the channel type assigned to x in A.

The following partial operations and relations on channel/action types are used for controlling composition of processes.

Definition 2.6 ($\odot$ and $\asymp$ on channel/action types)

(1) $\odot$ on channel types is the least commutative partial operation such that:

(1-a)	$\tau \odot \overline{\tau} = \updownarrow$	$(\mathsf{md}(\tau) \in \mathcal{M}_{\downarrow})$		
(1-b)	$\tau \odot \tau = \tau$	$(\mathsf{md}(\tau) \in \mathcal{M}_{?})$	$\tau \odot \overline{\tau} = \tau$	$(\mathsf{md}(\tau) \in \mathcal{M}_{!})$

If $\tau \odot \tau'$ is defined, we write $\tau \asymp \tau'$ and say τ *and* τ' *compose.*

(2) The relation $\asymp$ on action types is given as: $A \asymp B$ iff:
—whenever $x : \tau \in A$ and $x : \tau' \in B$, $\tau \odot \tau'$ is defined; and
—whenever $x_1 \rightarrow x_2$, $x_2 \rightarrow x_3$, $\ldots$, $x_{n-1} \rightarrow x_n$ alternately in A and B $(n \geq 2)$, we have $x_1 \neq x_n$.

(3) Finally the operation $A \odot B$, 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 $\tau = \tau' \odot \tau''$.
—$x \rightarrow y$ in $A \odot B$ iff $x : \tau_{\mathrm{I}}, y : \tau_{\mathrm{O}} \in |A \odot B|$ and $x = z_1 \rightarrow z_2$, $z_2 \rightarrow z_3, \ldots, z_{n-1} \rightarrow z_n = y$ $(n \geq 2)$ alternately in A and B.

Remark 2.7 (composition of channel/action types)

(1) Clause (1-a) in Definition 2.6 says that once we compose two processes at a shared linear channel, one using it for input and another for output, then that channel becomes no longer composable. Clause (1-b) says that a server should be unique, to which an arbitrary number of clients can request interactions.

(2) $\tau \asymp \tau'$ captures how we want composition of processes to be coherent. For example, we shall see $!x(y).P \mid !x(y).Q$ is never typed because $(\tau)^{!_\mathrm{L}} \not\asymp (\tau)^{!_\mathrm{L}}$ for any τ; similarly $\overline{x} \mid \overline{x}$ is untyped if x is either linear or affine because $()^{\uparrow_\mathrm{L}} \not\asymp ()^{\uparrow_\mathrm{L}}$ and $()^{\uparrow_\mathrm{A}} \not\asymp ()^{\uparrow_\mathrm{A}}$; whereas, if x is affine, we shall see $\overline{x} \mid x.\mathbf{0}$ is typable by $()^{\uparrow_\mathrm{A}} \asymp ()^{\downarrow_\mathrm{A}} = \updownarrow$, while $\overline{x} \mid x.\mathbf{0} \mid \overline{x}$ is untypable since $\updownarrow \not\asymp ()^{\uparrow_\mathrm{A}}$. Note also the composition between affine and linear types is prohibited; for example, $()^{\downarrow_\mathrm{L}} \not\asymp ()^{\uparrow_\mathrm{A}}$ and $(()^{\uparrow_\mathrm{L}})^{!_\mathrm{L}} \not\asymp (()^{\downarrow_\mathrm{A}})^{?_\mathrm{A}}$, whereas we have $(()^{\uparrow_\mathrm{A}})^{!_\mathrm{A}} \odot (()^{\downarrow_\mathrm{A}})^{?_\mathrm{A}} = (()^{\uparrow_\mathrm{A}})^{!_\mathrm{A}}$.

(3) We can easily check that $\odot$ on action types is a symmetric and associative partial operation, with the identity $\emptyset$ (the empty action type). This allows us to write the n-fold composition $\odot_i A_i$ without ambiguity.

Notation 2.8 (action types)

- $A^{\text{-}x}$ indicates A such that $x \notin \mathsf{fn}(A)$.
- $\vec{p}A$ indicates A such that $\mathsf{md}(A) \subset \{\vec{p}\}$. $?A$ indicates A such that $\mathsf{md}(A) \subset \mathcal{M}_?$.
- A, B denotes the disjoint union of A and B, assuming $\mathsf{fn}(A) \cap \mathsf{fn}(B) = \emptyset$.
- The *hiding* $A/\vec{x}$ is the result of taking off nodes with names in $\vec{x}$ from A.
- The *prefix* $x : \tau \to A$ adds an edge from a new node $x : \tau$ to each node in A.
- We can describe an action type by the following grammar, with possible duplicating ?-nodes [74, Example 1].

$$A ::= \emptyset \mid a : \tau \mid A, B \mid a : \tau \to (b_1 : \tau_1, b_2 : \tau_2, \cdots, b_n : \tau_n)$$

  where we assume that "$\to$" is stronger than ",".

Example 2.9 (operations on action types)

(1) (hiding) Assume $A_1 = a : \tau \to (b_1 : \tau_1, b : \tau_2)$. Then $A_1/a = (b_1 : \tau_1, b_2 : \tau_2)$.

(2) (prefix) Assume $A_2 = (b_1 : \tau_1, b_2 : \tau_2)$. Then $a : \tau \to A_2 = A_1$.

(3) (affine composability) Assume $A_3 = a : ()^{\downarrow_\mathrm{A}}, b : ()^{\uparrow_\mathrm{A}}$ and $A_4 = a : ()^{\uparrow_\mathrm{A}}, b : ()^{\downarrow_\mathrm{A}}$. Then $A_3 \asymp A_4$ and $A_3 \odot A_4 = a : \updownarrow, b : \updownarrow$. Assume $A_5 = a : (\tau)^{!_\mathrm{A}}, b : (\overline{\tau})^{?_\mathrm{A}}$ and $A_6 = b : (\tau)^{!_\mathrm{A}}, a : (\overline{\tau})^{?_\mathrm{A}}$. Then $A_5 \asymp A_6$ and $A_5 \odot A_6 = a : (\tau)^{!_\mathrm{A}}, b : (\tau)^{!_\mathrm{A}}$.

(4) (linear composability) Assume $A_7 = a : ()^{\downarrow_\mathrm{L}} \to b : ()^{\uparrow_\mathrm{L}}$ and $A_8 = b : ()^{\downarrow_\mathrm{L}} \to a : ()^{\uparrow_\mathrm{L}}$. Then $A_7 \not\asymp A_8$ because of cycle between channel a and channel b. Similarly $(a : (\tau)^{!_\mathrm{L}} \to b : (\overline{\tau})^{?_\mathrm{L}}) \not\asymp (b : (\tau)^{!_\mathrm{L}} \to a : (\overline{\tau})^{?_\mathrm{L}})$. The new causality between linear channels is created by composing two types: for example,

$$\begin{aligned} & a : ()^{\downarrow_\mathrm{L}} \to (b : ()^{\uparrow_\mathrm{L}}, c : ()^{\uparrow_\mathrm{L}}) \ \odot \ b : ()^{\downarrow_\mathrm{L}} \to (d : ()^{\uparrow_\mathrm{L}}, e : ()^{\uparrow_\mathrm{L}}) \\ = \ & a : ()^{\downarrow_\mathrm{L}} \to (c : ()^{\uparrow_\mathrm{L}}, d : ()^{\uparrow_\mathrm{L}}, e : ()^{\uparrow_\mathrm{L}}), \ b : \updownarrow \end{aligned}$$

The replication newly creates more complex causality as follows.

$$\begin{aligned} & a : (\tau)^{!_\mathrm{L}} \to (b : (\overline{\tau})^{?_\mathrm{L}}, c : (\overline{\tau})^{?_\mathrm{L}}) \ \odot \ b : (\tau)^{!_\mathrm{L}} \to (d : (\overline{\tau})^{?_\mathrm{L}}, e : (\overline{\tau})^{?_\mathrm{L}}) \\ = \ & a : (\tau)^{!_\mathrm{L}} \to (c : (\overline{\tau})^{?_\mathrm{L}}, d : (\overline{\tau})^{?_\mathrm{L}}, e : (\overline{\tau})^{?_\mathrm{L}}), \ b : (\tau)^{!_\mathrm{L}} \to (d : (\overline{\tau})^{?_\mathrm{L}}, e : (\overline{\tau})^{?_\mathrm{L}}) \end{aligned}$$

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

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

$$(\mathsf{In}^{\downarrow_{\mathrm{A}}})\ \frac{\vdash P \triangleright \vec{y}\!:\!\vec{\tau}\,, \uparrow_{\mathrm{A}}?A^{\text{-}x}}{\vdash x(\vec{y}).P \triangleright x\!:\!(\vec{\tau})^{\downarrow_{\mathrm{A}}}, A} \qquad (\mathsf{In}^{!_{\mathrm{A}}})\ \frac{\vdash P \triangleright \vec{y}\!:\!\vec{\tau}, ?A^{\text{-}x}}{\vdash !\, x(\vec{y}).P \triangleright x\!:\!(\vec{\tau})^{!_{\mathrm{A}}}, A} \qquad (\mathsf{Out})\ \frac{-}{\vdash \overline{x}\langle \vec{y}\rangle \triangleright x\!:\!(\vec{\tau})^{p_{\mathrm{O}}},\ \vec{y}\!:\!\overline{\vec{\tau}}}$$

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

2.5 Linear/Affine Typing (3) Typing Rules

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. We give a brief illustration of typing rules.

—(Zero) starts from the empty action type.

—(Par) uses $\asymp$ for controlling composition. For example, if P has type $x : ()^{\uparrow_{\mathrm{L}}}$ and Q has type $x : ()^{\uparrow_{\mathrm{L}}}$, then $P \mid Q$ is not typable because $()^{\uparrow_{\mathrm{L}}} \not\asymp ()^{\uparrow_{\mathrm{L}}}$.

—(Res) allows hiding of a name only when its mode is $\updownarrow$ or replicated (so that channels of modes $\uparrow$, $\downarrow$ or $?$ should be compensated by their duals before restricted).

—(Weak) weakens $\updownarrow$ and $?$-nodes since we allow the possibility of having no action at these channels. Their weakening is necessary for subject reduction.

—$(\mathsf{In}^{\downarrow_{\mathrm{L}}})$ records the causality from linear input to linear output. $A^{\text{-}x}$ and $B^{\text{-}x}$ ensure the unique occurrence of x. In this and other input rules including replicated ones, an input never prefixes another input (*IO-alternation*, cf. [10; 11; 75; 32]).

—In $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$, $\downarrow_{\mathrm{A}}$ never prefixes $\uparrow_{\mathrm{L}}$, 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 [73]).

—$(\mathsf{In}^{!_{\mathrm{L}}})$ records the causality from a replicated input to $?_{\mathrm{L}}$-outputs. The condition $A^{\text{-}x}$ is required to ensure acyclicity. $(\mathsf{In}^{!_{\mathrm{A}}})$ is the same as $(\mathsf{In}^{!_{\mathrm{L}}})$ except it does not record causality. $(\mathsf{In}^{!_{\mathrm{L}}})$ and $(\mathsf{In}^{!_{\mathrm{A}}})$ never prefix $\uparrow_{\mathrm{L}}/\uparrow_{\mathrm{A}}$ actions (except an abstracted one): otherwise unicity of a linear/affine name would be lost. For example, z cannot be linear in $!x(\vec{y}).(\overline{z} \mid Q)$ because z is used at each invocation.

—In (Out), the map $\vec{y} : \vec{\tau}$ implicitly indicates the condition $\tau_i = \tau_j$ if $y_i = y_j$ (this ensures that a duplicated object name is assigned the same type). The rule assigns the dual of the corresponding carried type to each y_i. This is because a

(Bra$^{\downarrow_{\mathrm{L}}}$)
$$\frac{\vdash P_i \triangleright \vec{y}_i : \vec{\tau}_i, \uparrow_{\mathrm{L}} A^{\text{-}x}, \uparrow_{\mathrm{A}} ?B^{\text{-}x}}{\vdash x[\&_i(\vec{y}_i).P_i] \triangleright (x : [\&_i \vec{\tau}_i]^{\downarrow_{\mathrm{L}}} \to A), B}$$

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

(Bra$^{\downarrow_{\mathrm{A}}}$)
$$\frac{\vdash P_i \triangleright \vec{y}_i : \vec{\tau}_i, \uparrow_{\mathrm{A}} ?A^{\text{-}x}}{\vdash x[\&_i(\vec{y}_i).P_i] \triangleright x : [\&_i \vec{\tau}_i]^{\downarrow_{\mathrm{A}}}, A}$$

(Bra$^{!_{\mathrm{A}}}$)
$$\frac{\vdash P_i \triangleright \vec{y}_i : \vec{\tau}_i, ?A^{\text{-}x}}{\vdash ! \, x[\&_i(\vec{y}_i).P_i] \triangleright x : [\&_i \vec{\tau}_i]^{!_{\mathrm{A}}}, A}$$

(Sel)
$$\frac{-}{\vdash \overline{x}\mathtt{in}_j\langle \vec{y} \rangle \triangleright x : [\oplus_i \vec{\tau}_i]^{p_{\mathrm{O}}}, \ \vec{y} : \overline{\vec{\tau}_j}}$$

Fig. 4. Typing Branching and Selection

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 $\overline{\mathbb{N}^\circ} = ([\&_{i\in\mathbb{N}}]^{\downarrow_{\mathrm{L}}})^{?_{\mathrm{L}}}$, while e, via which 2 will be outputted, should have a type $[\oplus_{i\in\mathbb{N}}]^{\uparrow_{\mathrm{L}}}$).

—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 λ-calculus and additives in Linear Logic. In the selection, we assume $\tau_{jk} = \tau_{jl}$ if $y_k = y_l$, just as in (Out).

The following variant of (Out), which is specialised for bound output [10; 11; 29; 74; 75], 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 [74]). Below $C(\vec{y})$ is the obvious extension of the notationon $C(y)$ to vectors.

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

Similarly for selection with bound output, which we omit.

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

Example 2.10 (typed processes) Let $\mathtt{unit}^\circ \stackrel{\text{def}}{=} (()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$ and $\mathbb{N}^\circ \stackrel{\text{def}}{=} ([\oplus_{i\in\mathbb{N}}]^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$. In brief these channel types represent the mapping of the unit type and $\mathbb{N}$ type in simply typed λ-calculi, respectively. We also write $\mathtt{unit}^{\mathrm{A}} = (()^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}$ to denote the affine version of unit. All processes are from Example 2.1.

(1) $\vdash x.\overline{y} \triangleright x : ()^{\downarrow_{\mathrm{L}}} \to y : ()^{\uparrow_{\mathrm{L}}}$, $\vdash x.\overline{y} \triangleright x : ()^{\downarrow_{\mathrm{L}}}, y : ()^{\uparrow_{\mathrm{A}}}$ and $\vdash x.\overline{y} \triangleright x : ()^{\downarrow_{\mathrm{A}}}, y : ()^{\uparrow_{\mathrm{A}}}$ are well-typed, but $\vdash x.\overline{y} \triangleright x : ()^{\downarrow_{\mathrm{A}}}, y : ()^{\uparrow_{\mathrm{L}}}$ is not by the condition of (In$^{\downarrow_{\mathrm{A}}}$).

(2) $\vdash [\![n]\!]_x \triangleright x : \mathbb{N}^\circ$ and $\vdash [\![()]\!]_x \triangleright x : \mathtt{unit}^\circ$ are well-typed.

(3) $\vdash [\![\mathsf{succ}]\!]_x \triangleright x : (\overline{\mathbb{N}^{\circ}}\mathbb{N}^{\bullet})^{!_{\mathrm{L}}}$ is well-typed.

(4) The forwarder $\mathtt{fw}\langle ux\rangle$ is typed as $\vdash \mathtt{fw}\langle ux\rangle \triangleright u : \mathtt{unit}^{\circ} \rightarrow x : \overline{\mathtt{unit}^{\circ}}$. This typed process may be considered as the encoding of $x^{\mathtt{unit}}$, located at u. Note we can also type $\mathtt{fw}\langle ux\rangle$ with a different action type: $\vdash \mathtt{fw}\langle ux\rangle \triangleright u : \mathtt{unit}^{\mathrm{A}}, x : \overline{\mathtt{unit}^{\mathrm{A}}}$.

(5) The divergent agent Ω_u is typed as $\vdash \Omega_u \triangleright u : \mathtt{unit}^{\mathrm{A}}$, starting from (4) above. Note Ω_u cannot have type $\mathtt{unit}^{\circ}$ since for that purpose we should compose $u : \mathtt{unit}^{\circ} \rightarrow x : \overline{\mathtt{unit}^{\circ}}$ and $x : \mathtt{unit}^{\circ} \rightarrow u : \overline{\mathtt{unit}^{\circ}}$ which is not possible due to circularity.

We write π^{LA} for the resulting typed calculus.

2.6 Basic Syntactic Properties of π^{LA}

The following syntactic properties of π^{LA} are worth recording. In (3), $P \Downarrow_x$ means $P \twoheadrightarrow P' \equiv (\boldsymbol{\nu}\, \vec{y})(S|R)$, where S has the shape $\overline{x}\langle \vec{w}\rangle$ or $\overline{x}\mathtt{in}_i\langle \vec{w}\rangle$ with $x \notin \{\vec{y}\}$.

Proposition 2.11 (syntactic properties of π^{LA})

(1) (subject reduction) *If* $\vdash P \triangleright A$ *and* $P \twoheadrightarrow Q$ *then* $\vdash Q \triangleright A$.

(2) (confluence) *If* $\vdash P \triangleright A$ *for some* A, $P \longrightarrow Q_1$ *and* $P \longrightarrow Q_2$, *then for some* R *we have* $Q_1 \longrightarrow R$ *and* $Q_2 \longrightarrow R$.

(3) (linear liveness) *Assume* $\vdash P \triangleright \,!\updownarrow A, x : \tau$ *with* $\mathsf{md}(\tau) = \uparrow_{\mathrm{L}}$, *then for all* P' *such that* $P \twoheadrightarrow P'$, *we have* $P' \Downarrow_x$.

PROOF. The proof of (1) precisely follows [74, Prop. 2.2 (1)], and is subsumed by the proof of the corresponding result in the next section (where we consider a secrecy-enhanced version of π^{LA}). The proof of (2) precisely follows [74, Prop. 2.2 (2)]. (3) is proved in [73, Theorem 1]. □

(1) is the standard property. For (2), the statement indicates π^{LA} is about deterministic computation. Technically this comes from the lack of conflicting outputs to a linear/affine input (which eliminates the racing condition). For the applications in Sections 3 and 4, having deterministic behaviour suffices. In Section 5 and later, we shall consider an integration of state and nondeterminacy in π^{LA}. (3) means a linear output channel, either unary or selection, always guarantees an output action by typability. In detail, the statement says:

> *If P is typed with a channel whose type has a linear output mode, and, moreover, if P does not need to ask other processes for information to emit that output, then any $\longrightarrow$-derivative of P, including P itself, will eventually output at x.*

This is the liveness property, demonstrating linearity at linear channels is ensured even under fine-grained mixture of linear types and affine types (we can further show the process will never output at x again). The property will become important when we develop secrecy analysis.

Other salient features of the calculus includes fully abstract embeddability of various functional calculi in π^{LA}, including the simply typed λ-calculus with products and sums, call-by-name PCF and call-by-value PCF. In the light of these properties, π^{LA} may be considered as giving a meta-language for various (monomorphic) pure functional calculi, both terminating and potentially diverging.

3. SECRECY ANALYSIS IN THE LINEAR/AFFINE π-CALCULUS

3.1 Secrecy-Annotated Channel Types and Tampering Level

In this section we enhance π^{LA} so that the typability guarantees a certain notion of safe information flow (secrecy). This is done in two stages.

(1) Annotation of channel types of π^{LA} with secrecy levels.
(2) Refinement of typing rules which are now sensitive to these secrecy levels.

These steps will lead to the sequent of the shape:

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

which guarantees not only the linear/affine typability in π^{LA} but also safety in information flow, in the sense that, through the typed interface specified by A, P will never transform an incoming effect, or difference, at a high-level channel to an outgoing effect, or difference, at a low-level channel. In $\vdash_{\mathsf{sec}} P \rhd A$, the action type A records not only linear/affine types but also levels at which P may receive and emit information. In this and the next subsection, we shall discuss the steps (1) and (2) above one by one, illustrating each newly introduced idea with examples.

Remark 3.1 (process as information transformer) Before entering technical discussions, it is worth illustrating what we mean by "transform an incoming effect/difference to an outgoing effect/difference". In the present context, the terms "effect" and "difference" mean distinction of behaviours up to the standard contextual congruence on processes. As a simple example, consider the following composition of two processes. Assume y is typed as $()^{\uparrow_{\mathrm{A}}}$.

$$\overline{x}\mathtt{inl} \mid x[\,.\overline{y} \,\&\, .\omega_y\,]$$

where we set $\omega_y \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, u)(\Omega_u | \overline{u}\langle y\rangle)$, which simply diverges. In this composition, the right-hand process transmits the difference induced at x by $\overline{x}\mathtt{inl}$ (which is non-trivial since it could be, for example, a semantically distinct $\overline{x}\mathtt{inr}$) to the difference induced at y by $\overline{y}$ (which is again non-trivial since it could be the divergent ω_y).

3.1.1 Channel Types. Fix a non-trivial complete lattice $(\mathcal{L}, \sqsubseteq, \mathtt{H}, \mathtt{L})$ of secrecy levels, with the partial order $\sqsubseteq$, the top element $\mathtt{H}$ (the most secret) and the bottom element $\mathtt{L}$ (the most public). $s, s', \ldots$ range over the secrecy levels.

We first annotate channel types with these secrecy levels. For simplicity we still write $\tau, \tau', \ldots$ for secrecy-enhanced channel types. The grammar follows.

$$\tau \;::=\; \tau_{\mathrm{I}} \mid \tau_{\mathrm{O}} \mid \updownarrow \qquad \begin{array}{lcl} \tau_{\mathrm{I}} & ::= & (\vec{\tau})^{\downarrow_{\mathrm{A}}}_{s} \mid (\vec{\tau})^{\downarrow_{\mathrm{L}}} \mid [\&_i\vec{\tau_i}]^{\downarrow_{\mathrm{A}}}_{s} \mid [\&_i\vec{\tau_i}]^{\downarrow_{\mathrm{L}}}_{s} \mid [\&_i\vec{\tau_i}]^{!_{\mathrm{L}}} \mid [\&_i\vec{\tau_i}]^{!_{\mathrm{A}}} \\ \tau_{\mathrm{O}} & ::= & (\vec{\tau})^{\uparrow_{\mathrm{A}}}_{s} \mid (\vec{\tau})^{\uparrow_{\mathrm{L}}} \mid [\oplus_i\vec{\tau_i}]^{\uparrow_{\mathrm{A}}}_{s} \mid [\oplus_i\vec{\tau_i}]^{\uparrow_{\mathrm{L}}}_{s} \mid [\oplus_i\vec{\tau_i}]^{?_{\mathrm{L}}} \mid [\oplus_i\vec{\tau_i}]^{?_{\mathrm{A}}} \end{array}$$

A secrecy annotation at each type indicates the secrecy level at which an interaction may take place. For example, $()^{\uparrow_{\mathrm{A}}}_{s}$ means that a channel may be used as an affine output at secrecy level s, while $()^{\downarrow_{\mathrm{A}}}_{s}$ says that a channel may be used for receiving an affine output at level s. Only those channel types which (represent actions that) directly transmit information are secrecy annotated. We present detailed illustration of these secrecy annotations in the following. The fine-grained nature of channel types (in comparison with types in standard programming languages) enables a clear understanding of the operational content of these secrecy annotations.

Duality on secrecy-enhanced channel types is defined respecting secrecy levels (e.g. $()^{\uparrow_\mathrm{A}}_s$ and $()^{\downarrow_\mathrm{A}}_{s'}$ are dual iff $s = s'$). Then $\odot$ and $\asymp$ between types are defined identically as in § 2.3. For example, $()^{\uparrow_\mathrm{A}}_{\mathtt{H}} \odot ()^{\downarrow_\mathrm{A}}_{\mathtt{L}}$ is undefined, while $()^{\uparrow_\mathrm{A}}_{\mathtt{H}} \odot ()^{\downarrow_\mathrm{A}}_{\mathtt{H}} = \updownarrow$. We write $\mathsf{sec}(\tau)$ for the outermost secrecy level of τ, e.g. $\mathsf{sec}((\tau)^p_s) = s$.

Remark 3.2 (on secrecy annotation) While the ultimate justification for annotating channel types as given above comes from the noninterference theorem presented later, the operational intuition underlying these annotations is important when we construct the secrecy typing rules (a further elucidation will be given in connection to tampering levels in Example 3.5 later).

(1) Affine unary types, $(\vec{\tau})^{\downarrow_\mathrm{A}}_s$ and $(\vec{\tau})^{\uparrow_\mathrm{A}}_s$, are secrecy-annotated. This is because processes receive/emit non-trivial immediate effect at affine unary channels because of the distinction between divergence and convergence (this may be understood by noting even the simplest affine type $x : ()^{\uparrow_\mathrm{A}}$ has two distinct inhabitants modulo the standard contextual congruence, cf. Remark 3.1 above).

(2) Linear unary types are not secrecy annotated, because interactions at unary linear channels are pre-determined by types so that no immediate effect is transmitted. Though channels carried in a unary linear communication may as well transmit non-trivial information later, these effects only take place when those channels are used, and can be recorded at each carried type, so they do not have to be recorded at this point (note there is a sharp separation between carrying potential information and directly transmitting information: we only annotate a channel type if some information can be actually used at the corresponding interaction).

(3) Branching/selection linear/affine types are secrecy annotated since, as we can immediately observe, there is immediate transfer of information at associated channels.

(4) Replicated linear/affine unary types as well as their duals are not secrecy-annotated since there is no immediate effect transmitted at each interaction at associated channels (technically this can be understood via the fact that the action type $x\!:\!(()^{\uparrow_\mathrm{A}})^{!_\mathrm{A}}$ semantically has two inhabitants which directly mirrors those of $()^{\uparrow_\mathrm{A}}$: there is nothing new in this type in addition to what is in its constituent type, $()^{\uparrow_\mathrm{A}}$, as far as information content goes).

(5) Replicated linear/affine branching types are understood as above: immediate transmission of information comes at types (channels) carried by these channels, if ever, and not at the replicated types themselves.

3.1.2 Action Types and Tamper Level. Action types are defined precisely 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, which we call *tamper level*, since it indicates the lower bound of the levels at which the process may affect, or tamper, the environment.[2] The tamper level is first defined on channel types, and is extended to action types.

[2] The tamper level corresponds to an explicitly secrecy level (called "pc" for a program counter) in [16] and subsequent work, though here it is calculated from its types. For more discussions on this point, see Section 8.

Definition 3.3 (1) τ is *immediately tampering* if either $\tau = [\oplus_i \vec{\tau}_i]_s^{\uparrow_{\mathrm{L}}}$ or $\mathsf{md}(\tau) = \uparrow_{\mathrm{A}}$. (2) τ is *innocuous* if $\mathsf{md}(\tau) \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}, \updownarrow\}$.

Definition 3.4 The *tamper level of* τ, denoted $\mathsf{tamp}(\tau)$, is inductively given by:

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

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

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, via an immediately tampering type it carries, so that it may have a non-trivial tampering level. However an innocuous type does not even have such a latent effect: $?_{\mathrm{L}}/?_{\mathrm{A}}$-actions just create a new copy of the resource, leaving the environment as it originally was (because replication is stateless in π^{LA}). Concrete examples of calculation of tamper levels follow.

Example 3.5 (tampering level)

(1) $()^{\uparrow_{\mathrm{L}}}$ is not immediately tampering: in fact, its tamper level is $\mathtt{H}$. This is because this type represents a behaviour which necessarily sends an empty output precisely once. The behaviour is completely determined by its type, so no information is transmitted by interaction (the type has semantically a unique inhabitant, cf. Remark 3.2 (2)). On the other hand, $[\oplus_{i\in\mathbb{N}}]_s^{\uparrow_{\mathrm{L}}}$ is immediately tampering (with a countable inhabitants), by emitting one of the natural numbers, and has a non-trivial tamper level s.

(2) Both $()_s^{\uparrow_{\mathrm{A}}}$ and $[\oplus_{i\in\mathbb{N}}]_s^{\uparrow_{\mathrm{A}}}$ are immediately tampering, with the 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 $\mathtt{unit}^\circ = (()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$. Then $\mathsf{tamp}(\mathtt{unit}^\circ) = \mathtt{H}$. A process inhabiting $x : \mathtt{unit}^\circ$ is always ready to receive at x: then it necessarily outputs an empty message via that name precisely once (as can be seen from its sole inhabitant $[\![()]\!]_x \stackrel{\mathrm{def}}{=} !x(c).\overline{c}$ modulo the observational congruence). Thus neither interaction contains information.

(4) Let $\mathbb{N}_s^\circ \stackrel{\mathrm{def}}{=} (\mathbb{N}_s^\bullet)^{!_{\mathrm{L}}}$ with $\mathbb{N}_s^\bullet \stackrel{\mathrm{def}}{=} [\oplus_{i\in\mathbb{N}}]_s^{\uparrow_{\mathrm{L}}}$. 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 $x : \mathbb{N}_s^\circ$ for each s. This process does contain information. To see this, take P below, with $R_0 \stackrel{\mathrm{def}}{=} \overline{y}$ and $R_i \stackrel{\mathrm{def}}{=} \omega_y$ for each $i \sqsubseteq 1$.

$$P \stackrel{\mathrm{def}}{=} \overline{x}(c)c[\&_i.R_i]$$

When composed with $[\![2]\!]_x$, P receives information in $[\![2]\!]_x$ at a carried channel c, and transmits it to the distinction between convergence and divergence at y.

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

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

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

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

(5) As (4) above, $\mathtt{unit}^{\mathrm{A}}_s \stackrel{\mathrm{def}}{=} (()^{\uparrow_{\mathrm{A}}}_s)^{!_{\mathrm{A}}}$ is not immediately tampering but its tampering level is non-trivial, with $\mathsf{tamp}(\mathtt{unit}^{\mathrm{A}}_s) = s$. To see this is justifiable, consider $[\![()]\!]_x$ and Ω_x in Example 2.1 (5) and Example 2.10 (5), both of which are typed with $x : \mathtt{unit}^{\mathrm{A}}_s$. These two processes are obviously semantically distinct since the former emits c after invocation, while the latter never does so.

(6) By definition, $\mathsf{tamp}(\tau) = \mathtt{H}$ for any τ such that $\mathsf{md}(\tau) \in \{\mathbf{?}_{\mathrm{L}}, \mathbf{?}_{\mathrm{A}}\}$. This is because a $\mathbf{?}_{\mathrm{L}}$ or $\mathbf{?}_{\mathrm{A}}$-action only creates a copy of a stateless replicated process, leaving the environment unchanged, even though they themselves do receive information. In fact, we can check processes typable by $x : \tau$ for such τ are behaviourally equivalent to $\mathbf{0}$, see [75] for detail.

3.2 Secrecy Typing

As we already noted, the secrecy typing uses the sequent of the form $\vdash_{\mathsf{sec}} P \rhd A$. Other than the use of secrecy-annotated channel types, the typing rules are refined to guarantee the secure information flow. The rule of thumb for refinement may be summarised thus:

> *Whenever we introduce a construct which receives non-trivial immediate effect, we require its level is either lower than, or equal to, the tamper level of the behaviour resulting from it.*[3]

We list the refined typing rules in Figure 5. Except for the listed three rules, all rules remain as in Figures 3 and 4, changing the sequent from $\vdash P \rhd A$ to $\vdash_{\mathsf{sec}} P \rhd A$. Each of the refined rules introduces a channel type which immediately receives information/effects, and adds a condition (emphasised) on secrecy levels, following the policy summarised above. We say P *is securely-typed under* A, or simply P *is secure under* A, when $\vdash_{\mathsf{sec}} P \rhd A$ is derivable from the secrecy typing rules. If P is not secure under A, we say P *is insecure under* A.

Below we list a few simple examples of (non-)securely typed processes, using processes from Example 2.10 and secrecy types from Example 3.5.

[3]We here take the standard view that it is dangerous for information to be transmitted to an incompatible level. If some information is classified to be read by principals of a specific rank, then those who are at the same or higher ranks may as well read it, but not those in unrelated ranks (see Remark 3.13 later for discussions on related points).

Example 3.6 (secure typing) Below we recall, from Example 3.5, $\mathbb{N}^{\circ}_s = (\mathbb{N}^{\bullet}_s)^{!_{\mathrm{L}}}$, $\mathbb{N}^{\bullet}_s = [\oplus_{i\in\mathbb{N}}]^{\uparrow_{\mathrm{L}}}_s$ and $\mathtt{unit}^{\mathrm{A}}_s = (()^{\uparrow_{\mathrm{A}}}_s)^{!_{\mathrm{A}}}$.

(1) $\vdash_{\mathsf{sec}} [\![n]\!]_x \triangleright x\!:\!\mathbb{N}^{\circ}_s$ is well-typed for each s. Intuitively, $[\![n]\!]_x$ only emits information, hence it is safe. Similarly $\vdash_{\mathsf{sec}} \overline{u}(x)[\![n]\!]_x \triangleright u:(\mathbb{N}^{\circ}_s)^{\uparrow_{\mathrm{L}}}$ is secure for each s. As a further example, $\vdash_{\mathsf{sec}} !u(c).\overline{c}(xy)([\![3]\!]_x \mid [\![6]\!]_y) \triangleright u\!:\!((\mathbb{N}^{\circ}_{s_1}\mathbb{N}^{\circ}_{s_2})^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$ is well-typed for each s_1 and s_2. This process encodes a pair $\langle 3,6\rangle$; the type $((\mathbb{N}^{\circ}_{s_1}\mathbb{N}^{\circ}_{s_2})^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$ encodes the product $\mathbb{N}^{\circ}_{s_1}\times\mathbb{N}^{\circ}_{s_2}$. The process emits/receives no information at the initial two interactions: all information it has is in two numbers, both of which emit information. We can check its tamper level is indeed $s_1 \sqcap s_2$.

(2) $\vdash_{\mathsf{sec}} x.\overline{y} \triangleright x:()^{\downarrow_{\mathrm{A}}}_{\mathtt{L}}, y:()^{\uparrow_{\mathrm{A}}}_{\mathtt{H}}$ is well-typed, since it receives information at the low level which is transmitted to the high level; whereas the same process $x.\overline{y}$ is insecure under $x\!:\!()^{\downarrow_{\mathrm{A}}}_{\mathtt{H}}, y\!:\!()^{\uparrow_{\mathrm{A}}}_{\mathtt{L}}$ since it leaks high-level information to a low-level channel. Generally $\vdash_{\mathsf{sec}} x.\overline{y} \triangleright x\!:\!()^{\downarrow_{\mathrm{A}}}_s, y\!:\!()^{\uparrow_{\mathrm{A}}}_{s'}$ is typable iff $s \sqsubseteq s'$.

(3) As in (2) above, $\vdash_{\mathsf{sec}} \mathtt{fw}\langle xy\rangle \triangleright x:\mathtt{unit}^{\mathrm{A}}_s,\ y:\overline{\mathtt{unit}^{\mathrm{A}}_{s'}}$ is secure iff $s' \sqsubseteq s$. Also $\vdash_{\mathsf{sec}} [\![()]\!]_x \triangleright x:\mathtt{unit}^{\circ}$, $\vdash_{\mathsf{sec}} [\![()]\!]_x \triangleright x:\mathtt{unit}^{\mathrm{A}}_s$ and $\vdash_{\mathsf{sec}} \Omega_x \triangleright x:\mathtt{unit}^{\mathrm{A}}_s$ are well-typed. This example concretely shows the significance of distinction between affinity and linearity in secrecy analysis. When a channel x is typed by $\mathtt{unit}^{\mathrm{A}}_s$, there *is* information held at x since we cannot predict whether interaction at x terminates or not, in contrast to $\mathtt{unit}^{\circ} = (()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$.

(4) Let $P \stackrel{\mathrm{def}}{=} u[.\overline{x}(y)[\![1]\!]_y \& .\overline{x}(y)[\![2]\!]_y]$, which is typable in π^{LA} with the action type $u:[\ \&\ ]^{\downarrow_{\mathrm{L}}} \to x:(\mathbb{N}^{\circ})^{\uparrow_{\mathrm{L}}}$. This process outputs 1 or 2 depending on which branching is selected at u (as we shall see later, the process is an encoding of the phrase $\mathtt{if}\ u\ \mathtt{then}\ 1\ \mathtt{else}\ 2$). Observe that, in P, the action at u affects the output at x. Let us check P is securely typed under $u\!:\![\ \&\ ]^{\downarrow_{\mathrm{L}}}_s \to x\!:\!(\mathbb{N}^{\circ}_{s'})^{\uparrow_{\mathrm{L}}}$ with $s \sqsubseteq s'$. Below we show the last (and only non-trivial) step of the inference.

$$\frac{\vdash_{\mathsf{sec}} \overline{x}(y)[\![1]\!]_y \triangleright x\!:\!(\mathbb{N}^{\circ}_{s'})^{\uparrow_{\mathrm{L}}} \quad \vdash_{\mathsf{sec}} \overline{x}(y)[\![2]\!]_y \triangleright x\!:\!(\mathbb{N}^{\circ}_{s'})^{\uparrow_{\mathrm{L}}} \quad s \sqsubseteq s' = \mathsf{tamp}(x\!:\!(\mathbb{N}^{\circ}_{s'})^{\uparrow_{\mathrm{L}}})}{\vdash_{\mathsf{sec}} u[.\overline{x}(y)[\![1]\!]_y \& .\overline{x}(y)[\![2]\!]_y] \triangleright u\!:\![\ \&\ ]^{\downarrow_{\mathrm{L}}}_s \to x\!:\!(\mathbb{N}^{\circ}_{s'})^{\uparrow_{\mathrm{L}}}}$$

Thus P is secure under $u\!:\![\ \&\ ]^{\downarrow_{\mathrm{L}}}_s \to x:(\mathbb{N}^{\circ}_{s'})^{\uparrow_{\mathrm{L}}}$. For example, if $s = \mathtt{H}$ (which means the process receives information at the highest level) we can only have $s' = \mathtt{H}$ (which means the process also emits information at the highest level).

(5) $\vdash_{\mathsf{sec}} [\![\mathsf{succ}]\!]_u \triangleright u:(\overline{\mathbb{N}^{\circ}_s}\mathbb{N}^{\bullet}_{s'})^{!_{\mathrm{L}}}$ is secure iff $s \sqsubseteq s'$. Remembering the definition of $[\![\mathsf{succ}]\!]_u$ from Example 2.1, we check, under $s \sqsubseteq s'$:

$$\dfrac{\dfrac{\dfrac{\dfrac{-}{\vdash_{\mathsf{sec}} \overline{e}\mathtt{in}_{n+1} \triangleright e\!:\!\mathbb{N}^{\bullet}_{s'} \quad s \sqsubseteq s'}\ (\mathsf{Sel}^{\uparrow_{\mathrm{L}}})}{\vdash_{\mathsf{sec}} c[\&_{n\in\mathbb{N}}.\overline{e}\mathtt{in}_{n+1}] \triangleright c\!:\!\overline{\mathbb{N}^{\bullet}_s} \to e\!:\!\mathbb{N}^{\bullet}_{s'}}\ (\mathsf{Bra}^{\downarrow_{\mathrm{L}}})}{\vdash_{\mathsf{sec}} \overline{y}(c)c[\&_{n\in\mathbb{N}}.\overline{e}\mathtt{in}_{n+1}] \triangleright y\!:\!(\overline{\mathbb{N}^{\bullet}_s})^{?_{\mathrm{L}}}, e\!:\!\mathbb{N}^{\bullet}_{s'}}\ (\mathsf{Out}^{?_{\mathrm{L}}})}{\vdash_{\mathsf{sec}} [\![\mathsf{succ}]\!]_u \triangleright u\!:\!(\overline{\mathbb{N}^{\circ}_s}\mathbb{N}^{\bullet}_{s'})^{!_{\mathrm{L}}}}\ (\mathsf{In}^{!_{\mathrm{L}}})$$

But if $s \not\sqsubseteq s'$ the inference is not possible.

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 in π^{LA}.

3.3 Basic Properties of Secrecy Typing in π^{LA}

This subsection discusses key results in the secrecy analysis for π^{LA}. We first show that the typability is preserved under reduction. Then we establish the property which the secure typing guarantees for typable processes, noninterference.

Proposition 3.7 (subject reduction) *If* $\vdash_{\mathsf{sec}} P \triangleright A$ *and* $P \twoheadrightarrow Q$ *then* $\vdash_{\mathsf{sec}} Q \triangleright A$.

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

The rest is a routine, by rule induction on the reduction rules. See Appendix A for details. Note this result immediately implies the subject reduction for the calculus in Section 2 by making the secrecy lattice trivial. $\Box$

Next we prove the non-interference property. Its formulation uses a family of contextual congruences relativised by secrecy levels, which is a significant element of the present theory in its own right. Some preliminaries:

Definition 3.8

(1) 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 $x \notin \{\vec{z}\}$.
(2) A *typed context* $C[\,\cdot\,]^B_A$ is a context with a hole which, when filled with a term of type A, will produce a term of type B.
(3) A *typed congruence* is an equivalence on π^{LA}-terms s.t.: (i) it relates two terms with the same action type and (ii) it is closed under typed contexts.

In (1) above, it is enough to take only an output observable [28; 30] (in fact, in the following definition of contextual congruence, we only consider an output of an affine type). A typed context is often simply written $C[\,\cdot\,]$, omitting type scripts. We now introduce the secrecy sensitive congruence.

Definition 3.9 (secrecy-sensitive contextual congruence) Fix some s. Then *s-sensitive contextual congruence*, denoted $\cong_s$, is the maximum typed congruence which satisfies the following condition: whenever $P_1 \Downarrow_x$ and $\vdash_{\mathsf{sec}} P_1 \cong_s P_2 \triangleright x : ()^{\uparrow_{\mathsf{A}}}_{s'}$ such that $s' \sqsubseteq s$, we have $P_2 \Downarrow_x$. When $s = \mathtt{H}$ (the top element of the secrecy lattice), we write $\cong$ for $\cong_s$.

The notation $\cong$ for $\cong_{\mathtt{H}}$ makes sense because the latter coincides with the contextual congruence in π^{LA} without secrecy.

Intuitively, $\cong_s$ ignores actions which should not be observable from the level s. More concretely, consider a process which can legitimately receive information at level s (hence also at levels lower than s). $\cong_s$ can then be considered as equality from the viewpoint of such processes. By definition we immediately obtain:

Proposition 3.10 *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,* $\vdash_{\mathsf{sec}} P_1 \cong P_2 \triangleright A$ *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. The proof given below follows the standard method, cf. [41].

Lemma 3.11 (context lemma) *Let* $\vdash_{\mathsf{sec}} P_i \triangleright A$ $(i = 1, 2)$. *Then* $P_1 \cong_s P_2$ *if and only if, for each* $\vdash_{\mathsf{sec}} R \triangleright \overline{A}, x\!:\!()^{\uparrow_{\mathrm{A}}}_s$, $(\boldsymbol{\nu}\,\mathsf{fn}(A))(P_1|R) \Downarrow_x$ *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 $x\!:\!()^{\uparrow_{\mathrm{A}}}_s$ with x fresh. Assume, for each $\vdash_{\mathsf{sec}} R \triangleright \overline{A}, x\!:\!()^{\uparrow_{\mathrm{A}}}_s$, $(\boldsymbol{\nu}\,\mathsf{fn}(A))(P_1|R) \Downarrow_x$ iff $(\boldsymbol{\nu}\,\mathsf{fn}(A))(P_2|R) \Downarrow_x$. If the hole of $C[\,\cdot\,]$ is not under an input prefix, then we already have $C[\,\cdot\,] \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,\mathsf{fn}(A))(R|[\,\cdot\,])$. 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 σ) then, since P_1 could not contribute to the reduction, 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 forwarders, we can represent σ by parallel composition and hiding, so that the former condition gives us $C[P_2] \Downarrow_x$, as required. □

One of the central 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 which behave at a secrecy level strictly higher than, or incompatible with, s can always be equated by $\cong_s$. The levels of observable behaviour of P and Q are simply calculated to be the tampering level of their type A.

Proposition 3.12 (non-interference in π^{LA}) *If* $\vdash_{\mathsf{sec}} P_i \triangleright A$ $(i = 1, 2)$ *such that* $\mathsf{tamp}(A) = s$ *and* $s \not\sqsubseteq s'$, *then* $\vdash_{\mathsf{sec}} P_1 \cong_{s'} P_2 \triangleright A$.

Remark 3.13 (treatment of incompatibility in non-interference results) In the statements for non-interference properties found in the literature, the condition on s' given above, $s \not\sqsubseteq s'$, often takes a weaker form, "s' is strictly smaller than s". Here we give intuition why this stronger condition is reasonable. Fix s and s' such that $s \not\sqsubseteq s'$ but s' is not strictly lower than s (so they are distinct but there is no ordering between them), and consider the following simple process:

$$\vdash_{\mathsf{sec}} \overline{u} \triangleright u\!:\!()^{\uparrow_{\mathrm{A}}}_s$$

Should the information this process contains, which is an output at u, be transferred to the level s'? Assume a secure process, say P, receives information at s via u from the above process. Then P can transmit the effect only to an action at s or higher. Let the level of that action be s''. Then by $s \not\sqsubseteq s'$ and $s \sqsubseteq s''$, we know $s'' \not\sqsubseteq s'$. Thus s'' either has no ordering w.r.t. s' or, if not, s'' is strictly higher than s'. In either case, an observer of level s' can never observe information. Thus the more general condition in Proposition 3.12 is justifiable.

There are at least two methods which may be used for proving Proposition 3.12. One is based on the secrecy-sensitive bisimilarity studied in [75], which would shed light on the semantic aspects of the present secrecy analysis. The bisimilarity can also be used for enlarging typability when used in combination with secrecy typing. The proof based on this method is discussed in [75] for the secrecy typing in the linear π-calculus. Another method is based on an inductive analysis of causal chains of actions in secure processes, and offers a basic insight on the syntactic/operational structure ensured by the present secrecy typing and its extensions. The proof of Proposition 3.12 (including the corresponding results for the extensions of the secrecy typing we shall discuss in the following) based on this method, together with the development of the ambient theory, is presented in a separate paper [33].

Because $\cong_s$ is a congruence on securely typed processes, Proposition 3.12 immediately gives us the following compositional principle for secrecy analysis on linear/affine processes.

Corollary 3.14 *Let* $\vdash_{\mathsf{sec}} P_i \triangleright A$ *$(i = 1, 2)$ such that* $\mathsf{tamp}(A) \not\sqsubseteq s$. *Let* $C[\,\cdot\,]_A^B$ *be a typed context. Then we have* $\vdash_{\mathsf{sec}} C[P_1] \cong_s C[P_2] \triangleright B$.

Example 3.15 (reasoning based on noninterference) We shall explain the use of Corollary 3.14 for a compositional noninterference analysis. Let us consider the following processes, with $\tau \stackrel{\text{def}}{=} [\oplus]_{\mathtt{H}}^{\uparrow_{\mathtt{L}}}$.

$$\vdash_{\mathsf{sec}} [\![\mathtt{left}]\!]_w \stackrel{\text{def}}{=} !w(c).\overline{c}\mathtt{inl} \triangleright w\!:\!\tau, \qquad \vdash_{\mathsf{sec}} [\![\mathtt{right}]\!]_w \stackrel{\text{def}}{=} !w(c).\overline{c}\mathtt{inr} \triangleright w\!:\!\tau.$$

Then $[\![\mathtt{left}]\!]_w \cong_{\mathtt{L}} [\![\mathtt{right}]\!]_w$. By Corollary 3.14, for any well-typed context $C[\;]_{w:\tau}^A$, we have $C[[\![\mathtt{left}]\!]_w] \cong_{\mathtt{L}} C[[\![\mathtt{right}]\!]_w]$. As a concrete context, we can take:

$$C[\,\cdot\,]_{w:\tau}^{x:\tau} \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, w)([\,\cdot\,] \mid \overline{w}(u)P) \quad \text{where} \quad \tau = (\mathbb{N}_{\mathtt{H}}^{\circ})^{\uparrow_{\mathtt{L}}},\ P \stackrel{\text{def}}{=} u[.\overline{x}(y)[\![1]\!]_y \& .\overline{x}(y)[\![2]\!]_y].$$

Note P is from Example 3.6 (4). Then we have $C[[\![\mathtt{left}]\!]_w] \cong_{\mathtt{L}} C[[\![\mathtt{right}]\!]_w]$, which relies on the typability of this context: different outputs from the two filled contexts depend on two high-level inputs, hence the former cannot be observed by low-level observers (one may observe that the secure typability of contexts means that context is safe as a transformer of information/behaviour). Note this equality says nothing about the equality for high-level observers. Indeed, as can be easily seen, we have $C[[\![\mathtt{left}]\!]_w] \not\cong_{\mathtt{H}} C[[\![\mathtt{right}]\!]_w]$ (which does not contradict Corollary 3.14 since $\vdash_{\mathsf{sec}} [\![\mathtt{left}]\!]_w \not\cong_{\mathtt{H}} [\![\mathtt{right}]\!]_w \triangleright w\!:\!\tau$).

More substantial applications of Corollary 3.14 will be presented in the next section, where the noninterference in π^{LA} is reflected onto the corresponding property in two prototypical secure programming languages via their process encoding.

In the rest of Section 3, we explore two natural refinements of the secrecy typing we have just introduced. Not only are these refinements useful in practice (the first refinement, subtyping, will be used throughout our embedding results, while the second one, called inflation, will be used in the embedding of the extended Smith-Volpano calculus), but also they offer new insights on the secrecy analysis in π^{LA} and connects it to significant ideas in the existing secrecy analyses in the literature. Those readers whose main interests lie in applications may safely skip the rest of the section, referring back to it as needed.

3.4 Refinement (1): Subtyping

A type-based secrecy analysis is often associated with subtyping [1; 29; 70; 66]. In the following, we present a natural subtyping relation for the secrecy typing given in Figure 5, based on the ideas from [29].

Let us first illustrate the key ideas of this secrecy subtyping using simple examples. Below we show two cases of subtyping on channel types, one for output and one for input. Let $\mathtt{L} \lneq \mathtt{M} \lneq \mathtt{H}$.

$$()^{\uparrow_{\mathrm{A}}}_{\mathtt{M}} \leq ()^{\uparrow_{\mathrm{A}}}_{\mathtt{H}} \qquad ()^{\downarrow_{\mathrm{A}}}_{\mathtt{M}} \leq ()^{\downarrow_{\mathrm{A}}}_{\mathtt{L}} \tag{2}$$

Note the ordering is reversed (dualised) between input and output. For both cases, we use the following simple agent for illustration.

$$\vdash_{\mathsf{sec}} x.\overline{y} \rhd x : ()^{\downarrow_{\mathrm{A}}}_{\mathtt{M}}, \quad y : ()^{\uparrow_{\mathrm{A}}}_{\mathtt{M}} \tag{3}$$

Note this process is immediately secure. Applying the standard subsumption to (3) using the output subtyping in (2), we obtain:

$$\vdash_{\mathsf{sec}} x.\overline{y} \rhd x : ()^{\downarrow_{\mathrm{A}}}_{\mathtt{M}}, \quad y : ()^{\uparrow_{\mathrm{A}}}_{\mathtt{H}} \tag{4}$$

which still remains secure. Intuitively raising an output level still keeps secrecy since, if the output is done as a result of an input at some level, we may as well raise the level of the former. The input is completely dual: we again apply the subsumption to (3), this time using the input subtyping in (2), to obtain:

$$\vdash_{\mathsf{sec}} x.\overline{y} \rhd x : ()^{\downarrow_{\mathrm{A}}}_{\mathtt{L}}, \quad y : ()^{\uparrow_{\mathrm{A}}}_{\mathtt{H}} \tag{5}$$

which remains secure. The justification for this subsumption is dual to what we gave for the output subtyping for (4).

The subtyping we obtained above carries over to carried types, nested at an arbitrary depth. For example, from the following obviously secure process:

$$\vdash_{\mathsf{sec}} !u(y).\overline{w}(x)x.\overline{y} \rhd x : ((()^{\downarrow_{\mathrm{A}}}_{\mathtt{M}})^{?_{\mathrm{A}}}, \quad y : ((()^{\uparrow_{\mathrm{A}}}_{\mathtt{M}})^{!_{\mathrm{A}}} \tag{6}$$

we may as well derive:

$$\vdash_{\mathsf{sec}} !u(y).\overline{w}(x)x.\overline{y} \rhd x : ((()^{\downarrow_{\mathrm{A}}}_{\mathtt{L}})^{?_{\mathrm{A}}}, \quad y : ((()^{\uparrow_{\mathrm{A}}}_{\mathtt{H}})^{!_{\mathrm{A}}} \tag{7}$$

which is again safe. Thus we may as well have the following subtyping:

$$((()^{\uparrow_{\mathrm{A}}}_{\mathtt{M}})^{!_{\mathrm{A}}} \leq ((()^{\uparrow_{\mathrm{A}}}_{\mathtt{H}})^{!_{\mathrm{A}}} \qquad ((()^{\downarrow_{\mathrm{A}}}_{\mathtt{M}})^{?_{\mathrm{A}}} \leq ((()^{\downarrow_{\mathrm{A}}}_{\mathtt{L}})^{?_{\mathrm{A}}} \tag{8}$$

in which the ordering is covariant with respect to (8). This covariance in subtyping, which follows [29], is because each type abstracts the behaviour of the process rather than that of the environment in the present system (in contrast to [59]).

We can now formalise the above ideas as subtyping rules.

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

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*. We can then extend the secrecy analysis by adding the following standard subsumption rule.

$$(\mathsf{Subs})\ \frac{\vdash_{\mathsf{sec}} P \triangleright A \quad A \leq B}{\vdash_{\mathsf{sec}} P \triangleright B} \tag{9}$$

The resulting system satisfies the subject reduction (the proof follows [29]). The following example shows that the subsumption strictly adds typability:

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

How do we know that the resulting typed process is secure? We first observe $\overline{x}\langle y\rangle$ behaves precisely as $\overline{x}(u)u.\overline{y}$ up to the contextual equality in π^{LA}. But, with the above action type, $\overline{x}(u)u.\overline{y}$ is indeed typable without subsumption. Hence, by Proposition 3.12, $\vdash_{\mathsf{sec}} \overline{x}\langle y\rangle \triangleright x:(()_{\mathtt{L}}^{\downarrow_{\mathrm{A}}})^{?_{\mathrm{A}}}, y:()_{\mathtt{H}}^{\uparrow_{\mathrm{A}}}$ should also be secure.

The translation we used above — transforming a free output into a bound output using transforming agents — is a key method used in the noninterference result, which states (Subs) is semantically sound. Before proving it, we first introduce the general transforming agents, the *copy-cats*, extending the construction for the unit type in Example 2.1 (4). Below $\Pi_i P_i$ denotes $P_1 \mid ... \mid P_n$ for some $n \geq 1$.

Definition 3.16 (copycat) Let $\mathsf{md}(\tau)$ be an input type in π^{LA}. Then a *copy-cat of type* τ *from* x *to* x', written $[x \to x']^{\tau}$, is given by the following induction (we omit secrecy annotations since they are irrelevant).

$$\begin{array}{rcll}
[x \to x']^{(\vec{\tau})^p} & \stackrel{\mathrm{def}}{=} & x(\vec{y}).\overline{x'}\langle\vec{y}\rangle^{\vec{\tau}} & (p \in \mathcal{M}_{\downarrow}) \\
[x \to x']^{(\vec{\tau})^p} & \stackrel{\mathrm{def}}{=} & !x(\vec{y}).\overline{x'}\langle\vec{y}\rangle^{\vec{\tau}} & (p \in \mathcal{M}_{!}) \\
[x \to x']^{[\&_i\vec{\tau}_i]^p} & \stackrel{\mathrm{def}}{=} & x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i\langle\vec{y}_i\rangle^{\overline{\tau_{ij}}}] & (p \in \mathcal{M}_{\downarrow}) \\
[x \to x']^{[\&_i\vec{\tau}_i]^p} & \stackrel{\mathrm{def}}{=} & !x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i\langle\vec{y}_i\rangle^{\vec{\tau_{ij}}}] & (p \in \mathcal{M}_{!}) \\
\overline{x}\langle y_1..y_n\rangle^{\tau_1..\tau_n} & \stackrel{\mathrm{def}}{=} & \overline{x}(y'_1..y'_n)\Pi_i[y'_i \to y_i]^{\overline{\tau_i}} & \\
\overline{x}\mathtt{in}_j\langle y_1..y_n\rangle^{\tau_1..\tau_n} & \stackrel{\mathrm{def}}{=} & \overline{x}\mathtt{in}_j(y'_1..y'_n)\Pi_i[y'_i \to y_i]^{\overline{\tau_i}} &
\end{array}$$

where, in the last two lines, each τ_i is an output mode.

Processes $\overline{x}\langle y_1..y_n\rangle^{\tau_1..\tau_n}$ and $\overline{x}\mathtt{in}_j\langle y_1..y_n\rangle^{\tau_1..\tau_n}$ emulate free name passing by bound name passing. Just like a forwarder, a copycat precisely transmits the behaviour at one point to the behaviour at another point, as the following example shows.

$$\begin{array}{rcl}
[\![2]\!]_y|[x \to y]^{\mathbb{N}^\circ}|\overline{x}\langle e\rangle & \longrightarrow & [\![2]\!]_y|[x \to y]^{\mathbb{N}^\circ}|\overline{y}(c')c'[\&_i.\overline{e}\mathtt{in}_i] \\
& \longrightarrow & [\![2]\!]_y|[x \to y]^{\mathbb{N}^\circ}|(\boldsymbol{\nu}\, c')(\overline{c'}\mathtt{in}_2|c'[\&_i\overline{e}\mathtt{in}_i]) \\
& \longrightarrow & [\![2]\!]_y|[x \to y]^{\mathbb{N}^\circ}|\overline{e}\mathtt{in}_2
\end{array}$$

We contrast the above reduction with one using a forwarder, which uses free name passing. Let $\mathtt{fw}\langle xy\rangle^{\mathbb{N}^\circ} \stackrel{\mathrm{def}}{=} !x(c).\overline{y}\langle c\rangle$ below.

$$\begin{array}{rcl}
[\![2]\!]_y|\mathtt{fw}\langle xy\rangle^{\mathbb{N}^\circ} \mid \overline{x}\langle e\rangle & \longrightarrow & [\![2]\!]_y|\mathtt{fw}\langle xy\rangle^{\mathbb{N}^\circ} \mid \overline{y}\langle e\rangle \\
& \longrightarrow & [\![2]\!]_y|\mathtt{fw}\langle xy\rangle^{\mathbb{N}^\circ} \mid \overline{e}\mathtt{in}_2.
\end{array}$$

Proposition 3.17 (flows in copycats) Below the typability is that of the secrecy typing without (Subs).

(1) Let $\mathsf{md}(\tau_1) \in \{!_{\text{L}}, !_{\text{A}}, \downarrow_{\text{L}}, \downarrow_{\text{A}}\}$ and $\tau_2 \leq \tau_1$, we have $\vdash_{\mathsf{sec}} [x \to y]^{\tau} \triangleright x{:}\tau_1,\ y{:}\overline{\tau_2}$.
(2) $\vdash_{\mathsf{sec}} \overline{x}\langle\vec{y}\rangle^{\vec{\tau}} \triangleright x{:}\tau, \vec{y}{:}\vec{\tau}$ if $(\vec{\overline{\tau}})^{\uparrow_{\text{L}}} \leq \tau$ or $(\vec{\overline{\tau}})^{\uparrow_{\text{A}}}_{s} \leq \tau$.
(3) $\vdash_{\mathsf{sec}} \overline{x}\mathtt{in}_j\langle\vec{y}\rangle^{\vec{\tau}_j} \triangleright x{:}\tau,\ \vec{y}{:}\vec{\tau}_i$ if $[\oplus_i\overline{\vec{\tau}_j}]^{\uparrow_{\text{L}}}_{s} \leq \tau$ or $[\oplus_i\overline{\vec{\tau}_j}]^{\uparrow_{\text{A}}}_{s} \leq \tau$.

Proof. By induction on the structure of secrecy annotated channel type τ. Firstly, the base cases are: (a) $\tau = ()^{\uparrow_{\text{L}}}$, which is immediate from $\vdash_{\mathsf{sec}} \overline{x} \triangleright x : ()^{\uparrow_{\text{L}}}$; (b) $\tau = ()^{\uparrow_{\text{A}}}_{s}$, which is again immediate from $\vdash_{\mathsf{sec}} \overline{x} \triangleright x : ()^{\uparrow_{\text{A}}}_{s'}$ when $s \sqsubseteq s'$; and (c) the corresponding branching cases, which are similarly reasoned. For induction, we only show the case of unary linear/affine types. Assume $\tau' = (\vec{\overline{\tau}})^{\downarrow_{\text{L}}}$ and $\rho \leq \tau'$. Then we reason, starting from induction hypothesis:

$$\frac{\vdash_{\mathsf{sec}} \overline{y}\langle\vec{z}\rangle^{\vec{\tau}} \triangleright y{:}\overline{\rho},\ \vec{z}{:}\overline{\vec{\overline{\tau}}}}{\vdash_{\mathsf{sec}} [x \to y]^{\tau} \triangleright x{:}\tau',\ y{:}\overline{\rho}}$$

The case when $\tau' = (\vec{\tau})^{\downarrow_{\text{A}}}_{s'}$ needs to take secrecy levels into consideration. Let $\rho \leq \tau'$ so that, with $\mathsf{tamp}(\overline{\rho}) = s$, we have $s' \sqsubseteq s$. Then we infer:

$$\frac{\vdash_{\mathsf{sec}} \overline{y}\langle\vec{z}\rangle^{\vec{\tau}} \triangleright y{:}\overline{\rho},\ \vec{z}{:}\overline{\vec{\overline{\tau}}} \quad s' \sqsubseteq \mathsf{tamp}(\overline{\rho})(\stackrel{\text{def}}{=} s)}{\vdash_{\mathsf{sec}} [x \to y]^{\tau} \triangleright x{:}\tau',\ y{:}\overline{\rho}}$$

Other cases are similar. □

These copycats bridge different types related by the subtyping, so that they explicitly embody the notion of subsumptions as behaviours. This observation can now be used for proving the non-interference of the secrecy typing without subsumption.

Proposition 3.18 (non-interference with subtyping) *If* $\vdash_{\mathsf{sec}} P_i \triangleright A$ *($i = 1, 2$) in the secrecy analysis with (Subs) and, moreover,* $\mathsf{tamp}(A) = s$ *and* $s \not\sqsubseteq s'$*, then* $\vdash_{\mathsf{sec}} P_1 \cong_{s'} P_2 \triangleright A$.

Proof. Given a derivation in the system with (Subs), replace each instance of the application of (Subs) by parallel composition with copycats which mediate two levels (for which we do not need (Subs)). For example, in the simplest case, we replace, assuming τ is an output type:

$$(\mathsf{Subs})\frac{\vdash_{\mathsf{sec}} P \triangleright x : \tau \quad \tau \leq \tau'}{\vdash_{\mathsf{sec}} P \triangleright x : \tau'}$$

with, assuming y fresh:

$$(\mathsf{Par,\ Res})\frac{\vdash_{\mathsf{sec}} P\{y/x\} \triangleright y : \tau \quad \vdash_{\mathsf{sec}} [y \to x]^{\tau} \triangleright y{:}\overline{\tau}, x{:}\rho \quad \tau \leq \tau'}{\vdash_{\mathsf{sec}} (\boldsymbol{\nu}\, y)(P\{y/x\}|[y \to x]^{\tau}) \triangleright x : \tau'}$$

Symmetrically when τ is an input type. Since subsumption can be done channel by channel, this generalises to the case when an action type contains multiple names. Since composition with copycats do not change the behaviour up to $\cong$ (see e.g. [10]), we know whenever $\vdash_{\mathsf{sec}} P \triangleright A$ with (Subs) we have $\vdash_{\mathsf{sec}} P' \triangleright A$ without (Subs) such that $P' \cong P$. By Propositions 3.10 and 3.18, this implies the noninterference of P. □

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 inside a process while guaranteeing global secrecy. To illustrate the idea, take Q below.

$$Q \stackrel{\text{def}}{=} \overline{y}(ab)(!a(c).\overline{z}(c')c'^{\mathtt{H}}.\overline{c}^{\mathtt{L}} \mid b^{\mathtt{H}}.\overline{e}^{\mathtt{H}}) \tag{10}$$

where we attach the secrecy levels on channels for readability. Let us define $B \stackrel{\text{def}}{=} y\!:\!(\mathtt{unit}^{\mathrm{A}}_{\mathtt{L}}\ ()^{\downarrow\mathrm{A}}_{\mathtt{H}})^{?_{\mathrm{L}}},\ z\!:\!\overline{\mathtt{unit}^{\mathrm{A}}_{\mathtt{H}}},\ e\!:\!()^{\uparrow\mathrm{A}}_{\mathtt{H}}$, As the annotation in (10) indicates, Q under B has a secrecy violation since a high-level input c' suppresses a low-level output c. Observe however this process only affects the environment at e, and never at y and z. For example, we may compose Q with:

$$R \stackrel{\text{def}}{=} !y(ab).\overline{a}(c)c^{\mathtt{L}}.\overline{b}^{\mathtt{H}} \mid !z(c').\overline{c}'^{\mathtt{H}} \tag{11}$$

Note how the insecure action at $c^{\mathtt{L}}$ by Q is, so to speak, "nullified" by the high-level action at $b^{\mathtt{H}}$ by R: in fact, given Q is in effect going to output only at $e^{\mathtt{H}}$, there can be no semantically insecure flow induced by Q. In other words, the local secrecy violation in Q is as a whole ineffective.

We now introduce the refined secrecy analysis which allows processes such as Q to be well-typed. We use an operation called *inflation*, which essentially acts as a nullifier of a local secrecy violation in view of the global tampering level of the process.

Definition 3.19 (inflation) The *inflation of τ by s*, written $\tau \sqcup s$, is the result of taking the join of each secrecy level in τ. Concretely:

$$\begin{array}{lll}
(\vec{\tau})^{p}_{s'} \sqcup s \stackrel{\text{def}}{=} (\vec{\tau} \sqcup s)^{p}_{s' \sqcup s} & [\&_i \vec{\tau}_i]^{p}_{s'} \sqcup s \stackrel{\text{def}}{=} [\&_i \vec{\tau}_i \sqcup s]^{p}_{s' \sqcup s} & [\oplus_i \vec{\tau}_i]^{p}_{s'} \sqcup s \stackrel{\text{def}}{=} [\oplus_i \vec{\tau}_i \sqcup s]^{p}_{s' \sqcup s} \\
(\vec{\tau})^{p} \sqcup s \stackrel{\text{def}}{=} (\vec{\tau} \sqcup s)^{p} & [\&_i \vec{\tau}_i]^{p} \sqcup s \stackrel{\text{def}}{=} [\&_i \vec{\tau}_i \sqcup s]^{p} & [\oplus_i \vec{\tau}_i]^{p} \sqcup s \stackrel{\text{def}}{=} [\oplus_i \vec{\tau}_i \sqcup s]^{p}
\end{array}$$

where $\vec{\tau} \sqcup s$ stands for $(\tau_1 \sqcup s)..(\tau_i \sqcup s)..(\tau_n \sqcup s)$ 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 3.20 *If* $\vdash_{\mathsf{sec}} P \triangleright A$ *then* $\vdash_{\mathsf{sec}} P \triangleright A \sqcup s$ *for each* s.

The extension of the analysis is done by incorporating the converse of Proposition 3.20 in a limited form.

Definition 3.21 We say $\vdash_{\mathsf{sec}} P \triangleright A$ *is well-typed with inflation* if it is typable with the secrecy typing in Figure 5, § 3.2, augmented with the following rule.

$$(\mathsf{Inf})\ \frac{\vdash_{\mathsf{sec}} P \triangleright \mathsf{inf}(A)}{\vdash_{\mathsf{sec}} P \triangleright A}$$

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

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. As a simple example of the use of (Inf), we show the derivation of Q under B above.

$$\text{(Inf)}\ \frac{\vdash_{\mathsf{sec}} Q \rhd y\!:\!(\mathtt{unit}^{\mathrm{A}}_{\mathtt{H}}\,()^{\downarrow\mathrm{A}}_{\mathtt{H}})^{?_{\mathrm{L}}},\ z\!:\!\overline{\mathtt{unit}^{\mathrm{A}}_{\mathtt{H}}},\ e\!:\!()^{\uparrow\mathrm{A}}_{\mathtt{H}}}{\vdash_{\mathsf{sec}} Q \rhd y\!:\!(\mathtt{unit}^{\mathrm{A}}_{\mathtt{L}}\,()^{\downarrow\mathrm{A}}_{\mathtt{H}})^{?_{\mathrm{L}}},\ z\!:\!\overline{\mathtt{unit}^{\mathrm{A}}_{\mathtt{H}}},\ e\!:\!()^{\uparrow\mathrm{A}}_{\mathtt{H}}}$$

For noninterference, we first define $\cong_s$ in precisely the same way as Definition 3.9, except we use processes typed with inflation this time. Then we have the same noninterference result as before.

Proposition 3.22 (noninterference with inflation) *If* $\vdash_{\mathsf{sec}} P_i \rhd A$ *($i = 1, 2$) is well-typed with inflation such that* $\mathsf{tamp}(A) = s$ *and* $s \not\sqsubseteq s'$*, then* $\vdash_{\mathsf{sec}} P_1 \cong_{s'} P_2 \rhd A$.

For the proof, see [34, §6.1] (which also illustrates how this operation arises naturally from the viewpoint of inductive flow analysis). Intuitively, a process with a tamper level s and which is typed with inflation will still transform information from levels which are the same as, or lower than, s, just as the one typed without inflation, since inflation is done only up to s.

The noninterference proof in [34, §6.1] does not depend on subject reduction of the secrecy type discipline with inflation. We nevertheless believe that, at least for sequential processes (in the sense of [10]) including its extension to stateful behaviours, the subject reduction holds.

4. SECRECY IN PURE FUNCTIONS: DEPENDENCY CORE CALCULUS

This section applies the secrecy analysis for π^{LA} presented in the previous section to the secrecy analysis for functional calculi, following the general framework we advocated in Introduction. First, in §4.1–4.4, we show that a representative existing secrecy analysis for pure functions is embeddable into the secrecy analysis for π^{LA}, taking the dependency core calculus by Abadi and others as an example. Second, in §4.5–4.7, we show how we can develop a new secrecy analysis for call-by-value higher-order functions by reflecting the secrecy analysis for π^{LA} onto the target calculus using the standard process encoding of call-by-functions [46; 31].

4.1 Dependency Core Calculus

The dependency core calculus [3] (DCC) is interesting in the present context at least in two ways. First, the calculus is one of the significant examples of a functional meta-language for type-based information flow analysis. Among others, [3] demonstrates how diverse forms of dependency/secrecy analyses for sequential programming languages can be analysed using the calculus. It is thus intriguing to see if its expressive secrecy analysis is embeddable into the secrecy-enhanced π^{LA}. Secondly, DCC uses *pointed types*[4] [35; 51] for a refined secrecy analysis. In Howard's framework, types are first non-pointed, which means they are for total (or terminating) programs. Pointed types are obtained by lifting non-pointed types, and represent behaviours which may diverge. In DCC, the fine-grained mixture of

[4] Howard [35] originally presented pointed types in a framework which also combines other (inductive and coinductive) type operators.

pointed and non-pointed types allows a refined type-based secrecy analysis that can take different kinds of causality — one for convergent computation and another for possibly divergent one — into account, by considering distinction between convergence/divergence in pointed types directly carries information. Thus we may ask whether the distinction between non-pointed types and pointed types relate to the one between linearity and affinity in π^{LA}.

Before answering these questions, we first review DCC. We present this calculus in the form which does not use explicit coercion of secrecy levels (both in types and terms). This does not result in any loss of precision or generality, while simplifying the presentation of the calculus and its embedding into the π-calculus.

The set of DCC-types are given by the following grammar. We use the same lattice $\mathcal{L}$ of secrecy levels as we set in Section 3, whose elements are ranged over by $s, s', \ldots$.

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

Unit, products, sums and function types should be familiar. The lifted type $\llcorner T \lrcorner_s$ is the so-called pointed type [35; 51], which indicates potential divergence. On these types we introduce the operation $(T)_s$ by the following clauses. This operation appears as an explicit type constructor in [3] with the notation $\mathbf{T}_s(T)$: in [3], the clauses below appear as type isomorphisms in the denotational universe.

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

We now introduce two key ideas in the DCC-types, the protection level and pointedness. Of them, the protection level is in direct correspondence with the tamper level in Section 3, as will become clear in the next subsection when we discuss the embedding of DCC in the secure π^{LA}. On the other hand, the pointedness indicates whether (intuitively) a type can host a divergent behaviour: again there is a direct correspondence with affinity, which will also be illustrated through the embedding in the next subsection.

Definition 4.1 (protection level and pointedness)

(1) The *protection level of* T, denoted $\mathsf{protect}(T)$, is given by:
- $\mathsf{protect}(\mathtt{unit}_s) = \mathsf{protect}(T_1 +_s T_2) = \mathsf{protect}(\llcorner T \lrcorner_s) = s$.
- $\mathsf{protect}(T_1 \times T_2) = \mathsf{protect}(T_1) \sqcap \mathsf{protect}(T_2)$ and:
- $\mathsf{protect}(T_1 \Rightarrow T_2) = \mathsf{protect}(T_2)$.

(2) $\llcorner T' \lrcorner_s$ is pointed; if T_1 and T_2 are pointed then $T_1 \times T_2$ is pointed; and if T' is pointed then $T \Rightarrow T'$ is pointed. These are all and only pointed types.

Proposition 4.2

(1) *T is pointed in [3] iff, regarding each $\mathbf{T}_s(T')$ occurring in T as $(T')_s$, it is pointed in the above sense.*

(2) *T is protected at s in [3] iff $s \sqsubseteq \mathsf{protect}(T)$, again through the translation of each $\mathbf{T}_s(T')$ in T to $(T')_s$.*

PROOF. By mechanical structural induction. □

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

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

$$[\mathit{Pair}]\quad \frac{E \vdash M_i : T_i \quad (i=1,2)}{E \vdash \langle M_1, M_2\rangle : T_1 \times T_2} \qquad [\mathit{Proj}]\quad \frac{E \vdash M : T_1 \times T_2}{E \vdash \pi_i(M) : T_i}$$

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

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

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

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

Fig. 6. Dependency Core Calculus

The sequent of DCC has the form $E \vdash M : T$ where M is a λ-preterm with units, sums, recursion, lifting and two let-like constructs, `bind` and `seq`, with annotations on bound names; while E is an *environment*, which is a finite map from variables to types. We often omit type annotations from DCC-terms. The reduction relation $\longrightarrow$ is the standard call-by-name one-step reduction, with the rules for `seq` and `bind` given as follows.

$$\begin{aligned} \mathtt{seq}\, x = \mathtt{lift}(N)\, \mathtt{in}\, M &\longrightarrow M\{N/x\} \\ \mathtt{bind}\, x = N\, \mathtt{in}\, M &\longrightarrow M\{N/x\} \end{aligned}$$

The typing rules are given in Figure 6. Apart from the lack of coercion, these rules are slightly strengthened without changing semantics, so that the typability is closed under reduction (this also makes [*UnitM*] and [*BindM*] redundant). See Appendix B.1 for illustration of the typing rules, including the difference from the original presentation. Simple examples of well-typed terms follow.

Example 4.3 (DCC-terms) Below let $\mathbb{B}_s \stackrel{\text{def}}{=} \mathtt{unit}_s +_s \mathtt{unit}_s$ (the levels of the unit types are in fact irrelevant).

(1) Let $L \stackrel{\text{def}}{=} \lambda x.x$. Then $L : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}}$ is well-typed. This is a function which outputs a low-level datum as a high-level datum, which is surely safe. This term is derived thus:

$$\text{(Lam)}\ \frac{\text{(Var)}\ \dfrac{-}{x:\mathbb{B}_{\mathtt{L}} \vdash x:(\mathbb{B}_{\mathtt{L}})_{\mathtt{H}}} \quad (\mathbb{B}_{\mathtt{L}})_{\mathtt{H}} = \mathbb{B}_{\mathtt{H}}}{\vdash \lambda x.x : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}}}$$

(2) Let $M \stackrel{\text{def}}{=} \lambda x.\mathtt{in}_1(())$. Then $M : \mathbb{B}_{\mathtt{H}} \Rightarrow \mathbb{B}_{\mathtt{L}}$ 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. For its derivation:

$$\dfrac{\dfrac{\dfrac{-}{x{:}\mathbb{B}_{\mathtt{H}} \vdash () : \mathtt{unit}_{\mathtt{L}}}\,(\mathsf{Unit})}{x{:}\mathbb{B}_{\mathtt{H}} \vdash \mathtt{inl}(()) : \mathbb{B}_{\mathtt{L}}}\,(\mathsf{Inl})}{\vdash \lambda x.\mathtt{inl}(()) : \mathbb{B}_{\mathtt{H}} \Rightarrow \mathbb{B}_{\mathtt{L}}}\,(\mathsf{Lam})$$

(3) Let $N = \mathtt{in}_1(()) : \mathbb{B}_{\mathtt{H}}$, which is obviously well-typed. Then, using M given in (2) above, $MN : \mathbb{B}_{\mathtt{L}}$ is well-typed, which is derived as:

$$(\mathsf{App})\ \dfrac{\vdash M : \mathbb{B}_{\mathtt{H}} \Rightarrow \mathbb{B}_{\mathtt{L}} \quad \vdash N : \mathbb{B}_{\mathtt{H}} \quad (\mathbb{B}_{\mathtt{H}})_{\mathtt{L}} = \mathbb{B}_{\mathtt{H}}}{\vdash MN : \mathbb{B}_{\mathtt{L}}}$$

(4) Using L in (1) and N in (3), $LN : \mathbb{B}_{\mathtt{H}}$ is well-typed and is derived as:

$$(\mathsf{App})\ \dfrac{\vdash L : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}} \quad \vdash N : \mathbb{B}_{\mathtt{H}} \quad (\mathbb{B}_{\mathtt{L}})_{\mathtt{H}} = \mathbb{B}_{\mathtt{H}}}{\vdash LN : \mathbb{B}_{\mathtt{H}}}$$

Here a high-level datum N is used as a low-level datum: however this is safe since, as a whole, the information is only emitted at the high-level.

(5) The term:

$$y : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}}, z : \mathbb{B}_{\mathtt{H}} \vdash \mathtt{bind}\ x = z\ \mathtt{in}\ yx : \mathbb{B}_{\mathtt{H}} \tag{12}$$

is well-typed as follows, with $E \stackrel{\text{def}}{=} x{:}\mathbb{B}_{\mathtt{L}},\ y{:}\mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}},\ z{:}\mathbb{B}_{\mathtt{H}}$.

$$\dfrac{\dfrac{E \vdash y : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}}, \quad E \vdash x : \mathbb{B}_{\mathtt{L}}}{E \vdash yx : \mathbb{B}_{\mathtt{H}}}\,(\mathsf{App}) \quad (\mathbb{B}_{\mathtt{L}})_{\mathtt{H}} = \mathbb{B}_{\mathtt{H}}}{y : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}},\ z : \mathbb{B}_{\mathtt{H}} \vdash \mathtt{bind}\ x = z\ \mathtt{in}\ yx : \mathbb{B}_{\mathtt{H}}}\,(\mathsf{Bind})$$

This example shows a subtle use of `bind`: a local violation of secrecy is permitted (a high-level z is used at a low-level) while retaining safe global flow.

Proposition 4.4

(*1*) (typability) *If* $E \vdash M : T$ *in [3] then* $E \vdash \mathsf{Erase}(M) : T$ *in the present system where, in the latter, we regard each* $\mathbf{T}_s(T')$ *occurring in* T *as* $(T')_s$ *and* $\mathsf{Erase}(M)$ *erases coercions from* M*.*

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

Proof. (1) is by rule induction via Prop. 4.2. (2) uses a strengthened substitution lemma, showing $E, x : T \vdash M : T'$ and $E \vdash N : (T)_s$ with $s \sqsubseteq \mathsf{protect}(T')$ implies $E \vdash M\{N/x\} : T'$. See Appendix B.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 4.5 *Fix some* s*. Then* $\cong_s^{\text{DCC}}$ *is the maximum typed congruence on DCC-terms such that whenever* $\vdash M_i : \mathbb{O}_s$ $(i = 1, 2)$*, we have* $M_1 \Downarrow$ *iff* $M_2 \Downarrow$*.*

4.2 Embedding DCC: Types

The embedding of DCC in π^{LA} is done by mapping non-pointed types to linear types and pointed types to affine types. The lifting $\llcorner T \lrcorner$ is replaced by a transformation from linearity to affinity. The scheme is based on Milner, Hyland-Ong, as well as our own studies [46; 37; 10]. Since the encoding of call-by-name product [37] may look slightly complex (though the framework itself is simple), we first work with types without product up to the noninterference result, presenting the extension to product types at the end of §4.4. The encoding of types follows. Below $[T_1T_2...T_{n-1}\gamma]$ stands for $T_1 \Rightarrow (T_2 \Rightarrow (\ldots(T_{n-1} \Rightarrow \gamma)\ldots)$ with γ is either a unit or a sum.

$$
\begin{array}{lll}
\text{(type)} & \mathtt{unit}^{\bullet}_{s} \stackrel{\mathrm{def}}{=} ()^{\uparrow_{\mathrm{L}}} \quad (T_1 +_s T_2)^{\bullet} \stackrel{\mathrm{def}}{=} [T_1^{\circ} \oplus T_2^{\circ}]_s^{\uparrow_{\mathrm{L}}} \quad \llcorner T \lrcorner^{\bullet}_{s} \stackrel{\mathrm{def}}{=} (T^{\circ})_s^{\uparrow_{\mathrm{A}}} & \\
& [T_1...T_{n-1}\gamma]^{\circ} \stackrel{\mathrm{def}}{=} \begin{cases} (\overline{T_1^{\circ}}...\overline{T_{n-1}^{\circ}}\gamma^{\bullet})^{!_{\mathrm{L}}} & \gamma \text{ non-pointed} \\ (\overline{T_1^{\circ}}...\overline{T_{n-1}^{\circ}}\gamma^{\bullet})^{!_{\mathrm{A}}} & \gamma \text{ pointed} \end{cases} & \\
\text{(environment)} & \emptyset^{\circ} \stackrel{\mathrm{def}}{=} \emptyset \qquad (E, x : T)^{\circ} \stackrel{\mathrm{def}}{=} E^{\circ}, x : \overline{T^{\circ}} & \\
\text{(action)} & \langle T \rangle_u^E \stackrel{\mathrm{def}}{=} \begin{cases} (u : T^{\circ} \to A), B & T \text{ non-pointed, } E^{\circ} = ?_{\mathrm{L}}A, ?_{\mathrm{A}}B \\ u : T^{\circ}, E^{\circ} & T \text{ pointed} \end{cases} &
\end{array}
$$

As examples, we have $\mathtt{unit}^{\circ}_{s} \stackrel{\mathrm{def}}{=} (()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$ and $\llcorner\mathtt{unit}\lrcorner^{\circ}_{s} \stackrel{\mathrm{def}}{=} ((\mathtt{unit}^{\circ})_s^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}$. Further we have $\mathbb{B}^{\bullet}_{s} = [\,\oplus\,]_s^{\uparrow_{\mathrm{L}}}$, $\mathbb{B}^{\circ}_{s} = ([\,\oplus\,]_s^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$, $\llcorner\mathbb{B}_s\lrcorner^{\bullet}_{\mathtt{L}} \stackrel{\mathrm{def}}{=} (\mathbb{B}^{\circ}_{s})_{\mathtt{L}}^{\uparrow_{\mathrm{A}}}$, and $(\mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}})^{\circ} = (\overline{\mathbb{B}^{\circ}_{\mathtt{L}}\mathbb{B}^{\bullet}_{\mathtt{H}}})^{!_{\mathrm{L}}}$. As an example of the encoding of action types, we have $\langle \mathbb{B}_{\mathtt{H}} \rangle_u^{x:\mathbb{B}_{\mathrm{L}}} = u : \mathbb{B}^{\circ}_{\mathtt{H}} \to x : \overline{\mathbb{B}^{\circ}_{\mathtt{L}}}$.

Below T is *trivial* when T is: (1) $\mathtt{unit}_s$; (2) $T_1 \times T_2$ where $T_{1,2}$ are trivial; and (3) $T_1 \Rightarrow T_2$ where T_2 is trivial.

Lemma 4.6 (1) $\mathsf{tamp}(T^{\circ}) = \mathtt{H}$ *if T is trivial.* $\mathsf{tamp}(T^{\circ}) = \mathsf{protect}(T)$ *if else.* (2) $\mathsf{tamp}(\langle T \rangle_u^E) = \mathsf{tamp}(T^{\circ})$. (3) $T^{\circ} \sqsubseteq ((T)_s)^{\circ}$.

PROOF. For (1), noting $\mathsf{tamp}([T_1...T_nT']^{\circ}) = \mathsf{tamp}(T')$ (since $\mathsf{md}(\overline{(T_i)^{\circ}_{s}}) \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}\}$), we have $\mathsf{tamp}([T_1..T_n\mathtt{unit}_s])^{\circ} = \mathsf{tamp}(\mathtt{unit}^{\bullet}_{s}) = H$ and $\mathsf{tamp}([T_1..T_n(T'_1 +_s T'_2)]^{\circ}) = s = \mathsf{tamp}([T_1'^{\circ} \oplus T_2'^{\circ}]_s^{\uparrow_{\mathrm{L}}})$, similarly for $[T_1..T_n\llcorner T'\lrcorner_s]$. (2) is because types in E° are innocuous. (3) is immediate. □

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 γ again (at a specific secrecy level). Further 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 each T_i has either $?_{\mathrm{L}}$ or $?_{\mathrm{A}}$ mode, so its tamper level is irrelevant.
- s in $\mathtt{unit}_s$ is ignored in translation. $[T_1..T_n\mathtt{unit}_s]^{\circ}$ says that, regardless of the results of interactions at $T_{1.n}$, it simply signals a unique output, hence in effect there is no flow of information (a similar treatment of the unit is found in [58]).
- The encoding of $T_1 +_s T_2$ is a linear selection type, which is immediately tampering and which needs a secrecy annotation.

Below we set $T = [T_1..T_{n-1}\gamma]$ and $\vec{x} = x_1...x_{n-1}$.

$[\![x^T : T]\!]_u \stackrel{\text{def}}{=} \mathtt{fw}\langle ux\rangle^{T^\circ}$ $\qquad [\![() : \mathtt{unit}_s]\!]_u \stackrel{\text{def}}{=} !u(x).\overline{x}$ $\qquad [\![n : \mathbb{N}_s]\!]_u \stackrel{\text{def}}{=} !u(x).\overline{x}\mathtt{in}_n$

$[\![\lambda x_0^{T_0}.M : T_0 \Rightarrow T]\!]_u \stackrel{\text{def}}{=} !\,u(x_0\vec{x}z).(\boldsymbol{\nu}\,u')([\![M : T]\!]_{u'} \mid \overline{u'}\langle\vec{x}z\rangle)$

$[\![MN : T]\!]_u \stackrel{\text{def}}{=} !\,u(\vec{x}z).(\boldsymbol{\nu}\,mx_0)([\![M : T_0 \Rightarrow T]\!]_m \mid [\![N : (T_0)_s]\!]_{x_0} \mid \overline{m}\langle x_0\vec{x}z\rangle)$

$[\![\mathtt{inl}(M) : T_1 +_s T_2]\!]_u \stackrel{\text{def}}{=} !\,u(c).\overline{c}\mathtt{in}_1(m)[\![M : T_1]\!]_m$

$[\![\mathtt{case}\, L\, \mathtt{of}\, \mathtt{inl}(x_1^{T_1})M_1\, \mathtt{or}\, \mathtt{inr}(x_2^{T_2})M_2 : T]\!]_u \stackrel{\text{def}}{=} !\,u(\vec{z}).(\boldsymbol{\nu}\,l)([\![L : T_1 + T_2]\!]_l \mid \mathsf{Sum}\langle l\vec{z}, (x_i)M_i^T\rangle)$

where $\mathsf{Sum}\langle l\vec{z}, (x_i)M_i^T\rangle \stackrel{\text{def}}{=} \overline{l}(c)c[\&_{i=1,2}(x_i).(\boldsymbol{\nu}\,m)([\![M_i : T]\!]_m \mid \overline{m}\langle\vec{z}\rangle)]$

$[\![\mathtt{bind}\; x^{T'} = N \;\mathtt{in}\; M : T]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,x)([\![M : T]\!]_u \mid [\![N : (T')_s]\!]_x)$

$[\![\mathtt{lift}(M) : \llcorner T \lrcorner_s]\!]_u \stackrel{\text{def}}{=} !\,u(c).\overline{c}(m)[\![M : T]\!]_m$

$[\![\mathtt{seq}\, x^{T'} = N\, \mathtt{in}\, M : T]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,n\vec{w})(\,!u(\vec{z}).\overline{n}(c)c(x).P \mid [\![N : \llcorner T \lrcorner_s]\!]_n \mid R\,)$

where $[\![M : T']\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,\vec{w})(\,!u(\vec{z}).P \mid R\,)$

$[\![\mu x^T.M : T]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,x)([\![M : T]\!]_u \mid \mathtt{fw}\langle xu\rangle^{T^\circ})$

Fig. 7. Encoding of Dependency Core Calculus

4.3 Embedding DCC: Terms

The translation of DCC-terms into processes, written $[\![M : T]\!]_u$ (often omitting T for brevity), closely follows that of types, and are defined inductively by the clauses in Figure 7 (omitting the obvious symmetric cases). The translation follows [46; 38] and does *not* rely on secrecy annotation of DCC-types, hence the translation is uniquely determined from a given DCC-term (however the translation does rely on the following property: each subterm of M in a DCC-term $E \vdash M : T$ is assigned a unique type if we neglect secrecy annotations). Some of the rules use a generalised forwarder $\mathtt{fw}\langle xy\rangle^\tau$ for an input type τ, given by:

$$\mathtt{fw}\langle xy\rangle^\tau \stackrel{\text{def}}{=} \begin{cases} x(z_1..z_n).\overline{y}\langle z_1..z_n\rangle & \text{if } \tau = (\rho_1..\rho_n)^p \text{ with } p \in \{\downarrow_{\rm L}, \downarrow_{\rm A}\} \\ !x(z_1..z_n).\overline{y}\langle z_1..z_n\rangle & \text{if } \tau = (\rho_1..\rho_n)^p \text{ with } p \in \{!_{\rm L}, !_{\rm A}\} \end{cases}$$

We discuss some of the key aspects of the encoding.

(1) $[\![M]\!]_u$ is always a replicated input (to be precise, of the form $(\boldsymbol{\nu}\,\vec{w})(!u(\vec{z}).P|R)$ with R having subjects $\vec{w}$). This corresponds to Milner's encoding [46] and game-based model [4; 37; 38] (in the latter corresponding to the property that interaction always starts from the opponent's question).

(2) A variable x is encoded into a forwarder located at u. The encoding of unit and natural numbers have appeared in Example 3.6.

(3) When M has a type $[T_0T_1....T_n\gamma]$ and N a type T_0, MN is translated into $[\![MN]\!]_u \stackrel{\text{def}}{=} !\,u(\vec{x}z).(\boldsymbol{\nu}\,mx_0)([\![M]\!]_m \mid [\![N]\!]_{x_0} \mid \overline{m}\langle x_0\vec{x}z\rangle)$. This agent, when invoked, calls $[\![M]\!]_m$ with parameters $\vec{x}$ plus x_0 standing for N.

(4) If $T = [T_0T_1....T_n\gamma]$, an abstraction $\lambda x_0.M$ of type T located at u is translated into $!\,u(x_0\vec{x}z).(\boldsymbol{\nu}\,u')([\![M]\!]_{u'} \mid \overline{u'}\langle\vec{x}z\rangle)$. When it is invoked at u with arguments $x_0\vec{x}z$, it in turn calls its body with the same arguments except the initial x_0 is taken off (which adjusts types). The agent will only affect the outside at its output at z, hence its level lies only at z. The behaviour is in clear contrast with the encoding of abstraction of the call-by-value version of DCC, which will be discussed in §4.5.

(5) An injection and a case construct are mapped into selection and branching, respectively. We can then justify the typing rule for $[Case]$ using the secrecy typing in π^{LA}. Let $[\![M_i]\!]_m \stackrel{\text{def}}{=} !m(\vec{z}y).P_i$ and $E, x : \alpha_1 +_s \alpha_2 \vdash_{\mathsf{sec}} M_{1,2} \triangleright \beta$ and $E \vdash_{\mathsf{sec}} L \triangleright \alpha_1 +_s \alpha_2$. By observing $(\boldsymbol{\nu}\,u')([\![M_i]\!]_m \mid \overline{m}\langle\vec{z}y\rangle)$ can be optimised to P_i, we infer:

$$(\mathsf{Par,Res})\;\frac{\begin{array}{l}\vdash_{\mathsf{sec}} [\![L]\!]_l \triangleright l\!:\!([T_1^\circ \oplus T_2^\circ]_s^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}, E^\circ \\ \vdash_{\mathsf{sec}} \bar{l}(c)c[\&_{i=1,2}(x_i).P_i] \triangleright l\!:\!([T_1^\circ \& T_2^\circ]_s^{\downarrow_{\mathrm{L}}})^{?_{\mathrm{L}}}, z_1\!:\!\overline{T_1^\circ}, .., z_n\!:\!\overline{T_n^\circ}, y\!:\!\gamma^\bullet, E^\circ\end{array}}{\vdash_{\mathsf{sec}} (\boldsymbol{\nu}\,l)([\![L]\!]_l \mid \bar{l}(c)c[\&_{i=1,2}(x_i).P_i]) \triangleright z_1\!:\!\overline{T_1^\circ}, .., z_n\!:\!\gamma^\bullet, E^\circ}$$

For securely typing the process in the second antecedent, we should have $s \sqsubseteq \mathsf{tamp}(z_1 : \overline{T_1^\circ}, .., z_n : \gamma^\bullet, E^\circ) = \mathsf{tamp}(\gamma^\bullet)$ by $(\mathsf{Bra}^{\downarrow_{\mathrm{A}}})$ (noting $\mathsf{tamp}(\overline{E^\circ}) = \mathsf{tamp}(\overline{T_1^\circ}) = \mathtt{H}$). Using Lemma 4.6, we can check that this condition exactly corresponds to the side condition of $[Case]$, $s \sqsubseteq \mathsf{protect}(\gamma) = \mathsf{protect}(T')$. Note the analysis clarifies information flow involved in each construct.

(6) The encoding of `bind` simply connects a channel of a variable to a channel of a behaviour which it is bound.

(7) The lifting is translated into $[\![\mathtt{lift}(M)]\!]_u \stackrel{\text{def}}{=} !\,u(c).\bar{c}(m)[\![M]\!]_m$ where m of M is transformed into an affine replicated u via additional interactions. Assuming $E \vdash M : T$, this agent is typed as $\vdash [\![\mathtt{lift}(M)]\!]_u : u : ((T^\circ)_s^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}, E^\circ$. For $[Seq]$, let $T' = [T_1...T_n\gamma]$ and $[\![M]\!]_u = !u(\vec{z}y).P$ with $\vec{z} = z_1..z_n$. Central to the secrecy typing of $[\![\mathtt{seq}\ x = N\ \mathtt{in}\ M]\!]_u$ is the affine prefixing of P by $c(x)$ in $!u(\vec{z}y).\overline{n}(c)c(x).P$, shown below.

$$(\mathsf{In}^{\downarrow_{\mathrm{A}}})\quad\frac{\vdash_{\mathsf{sec}} P \triangleright x : \overline{T^\circ},\ \vec{z} : \overline{\vec{T}^\circ},\ y : \gamma^\bullet,\ E^\circ}{\vdash_{\mathsf{sec}} c(x).P \triangleright c : (\overline{T^\circ})_s^{\downarrow_{\mathrm{A}}},\ \vec{z} : \overline{\vec{T}^\circ},\ y : \gamma^\bullet,\ E^\circ}$$

For this to be typable, $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$ in Figure 5 demands: (1) $\gamma^\bullet$ is affine, i.e. $T' = [T_1..T_n\gamma]$ is pointed, and (2) $s \sqsubseteq \mathsf{tamp}(\vec{z} : \vec{T}^\circ, y : \gamma^\bullet, \overline{E^\circ}) = \mathsf{tamp}(\gamma^\bullet) = \mathsf{protect}(\gamma) = \mathsf{protect}([T_1..T_n\gamma])$, reaching the side condition in $[Seq]$.

(8) The recursion is translated by connecting a recursive variable x and the location u by using a forwarder as: $[\![\mu x^T.M]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,x)([\![M]\!]_u \mid \mathtt{fw}\langle xu\rangle^{T^\circ})$. Since $\mathtt{fw}\langle xu\rangle$ creates a cycle between u and x, x should be typed with a $?_{\mathrm{A}}$-type (by the DCC-type of x being pointed). This corresponds to the side condition of $[Rec]$, that is T is pointed.

A few concrete examples of the encoding follow. The final example, (7), shows the encoding of `bind` where the inflation rule, (Inf), in § 3.5 is needed for justifying its well-typedness of the encoding.

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

(1) $\vdash () : \mathtt{unit}_s$ is translated into $\vdash_{\mathsf{sec}} !u(c).\overline{c} \triangleright u : ()^{\uparrow_{\mathrm{L}}}$, which is obviously secure.

(2) A DCC-term $x : \mathtt{unit}_s \vdash x : \mathtt{unit}_{s'}$ is translated into a secure π^{LA}-term $\vdash_{\mathsf{sec}} !u(c).\overline{x}\langle c\rangle \triangleright u : \mathtt{unit}^{\circ}_s \rightarrow x : \overline{\mathtt{unit}^{\circ}_{s'}}$. Since s and s' are ignored in the translation of $\mathtt{unit}_s$, this is well-typed for arbitrary s and s'.

(3) $x : \mathbb{B}_s \vdash x : \mathbb{B}_{s'}$ is translated into $\vdash_{\mathsf{sec}} !u(c).\overline{x}(c')c'[.\overline{c}\mathtt{inl} \,\&\, .\overline{c}\mathtt{inr}] \triangleright u : \mathbb{B}^{\circ}_s \rightarrow x : \overline{\mathbb{B}^{\circ}_{s'}}$. Then it is well-typed iff $s \sqsubseteq s'$ by the side condition of $(\mathsf{Bra}^{\downarrow_{\mathrm{L}}})$, since c' has level s while c has level s'.

(4) An identity function $\vdash \lambda x.x : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}}$ is translated into

$$\vdash_{\mathsf{sec}} !u(xc).\overline{x}\langle c\rangle \triangleright u : (\overline{\mathbb{B}^{\circ}_{\mathtt{L}}}\mathbb{B}^{\bullet}_{\mathtt{H}})^{!_{\mathrm{L}}}$$

applying the optimisation $(\boldsymbol{\nu}\, u')(\mathtt{fw}\langle u'x\rangle^{\mathbb{B}_s} \mid \overline{u'}\langle c\rangle) \longrightarrow \overline{x}\langle c\rangle$. This process is secure, since, after obtaining $\vdash_{\mathsf{sec}} \overline{x}\langle c\rangle \triangleright x : \overline{\mathbb{B}^{\circ}_{\mathtt{L}}},\ c : \mathbb{B}^{\bullet}_{\mathtt{L}}$, we can use (Subs) to get $\vdash_{\mathsf{sec}} \overline{x}\langle c\rangle \triangleright x : \overline{\mathbb{B}^{\circ}_{\mathtt{L}}},\ c : \mathbb{B}^{\bullet}_{\mathtt{H}}$, thus, by $(\mathrm{In}^{!_{\mathrm{L}}})$ we reach the above sequent. But the translation of $\vdash \lambda x.x : \mathbb{B}_{\mathtt{H}} \Rightarrow \mathbb{B}_{\mathtt{L}}$ is untypable; in $!u(xc).\overline{x}\langle c\rangle$, x should be assigned by $\mathtt{H}$, but then c cannot be by $\mathtt{L}$.

(5) A function $\vdash \lambda x.\mathtt{inl}(()) : \mathbb{B}_{\mathtt{H}} \Rightarrow \mathbb{B}_{\mathtt{L}}$ is translated into

$$\vdash_{\mathsf{sec}} !u(xc).\overline{c}\mathtt{inl}(m)!m(z).\overline{z} \triangleright u : (\mathbb{B}_{\mathtt{H}} \Rightarrow \mathbb{B}_{\mathtt{L}})^{\circ}$$

applying the optimisation as the above. This process is well-typed. First we type its body as $\vdash_{\mathsf{sec}} \overline{c}\mathtt{inl}(m)!m(z).\overline{z} \triangleright c : \mathbb{B}^{\bullet}_{\mathtt{L}}$. Then we can apply $(\mathsf{Rep}^{!_{\mathrm{L}}})$ to obtain the above term.

(6) $\vdash MN : \mathbb{B}_{\mathtt{L}}$ with $M \stackrel{\text{def}}{=} \lambda x.\mathtt{inl}(()) : \mathbb{B}_{\mathtt{H}} \Rightarrow \mathbb{B}_{\mathtt{L}}$ and $N \stackrel{\text{def}}{=} \mathtt{inl}(()) : \mathbb{B}_{\mathtt{H}}$ becomes:

$$!u(z).(\boldsymbol{\nu}\, mx_0)(!m(xc).\overline{c}\mathtt{inl}(m)!m(y).\overline{y} \mid !x_0(e).\overline{e}\mathtt{inl}(w)!w(v).\overline{v} \mid \overline{m}\langle x_0 z\rangle)$$

where u is typed by $\mathbb{B}^{\circ}_{\mathtt{L}}$, x_0 by $\mathbb{B}^{\circ}_{\mathtt{H}}$, and e by $\mathbb{B}^{\bullet}_{\mathtt{H}}$. This process is securely typed: although x_0 has the tampering level $\mathtt{H}$, it is not used in $[\![M]\!]_m$, so it is safe.

(7) A DCC-term $y : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}}, z : \mathbb{B}_{\mathtt{H}} \vdash \mathtt{bind}\ x = z\ \mathtt{in}\ yx : \mathbb{B}_{\mathtt{H}}$ in Example 4.3 (3) is translated into the following process (with some optimisation for simplicity).

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

Here we annotate channels by the level explicitly. Note y has type $(\overline{\mathbb{B}^{\circ}_{\mathtt{L}}}\mathbb{B}^{\bullet}_{\mathtt{H}})^{?_{\mathrm{L}}}$ while z has type $\overline{\mathbb{B}^{\circ}_{\mathtt{H}}}$, 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 $(\overline{\mathbb{B}^{\circ}_{\mathtt{H}}}\mathbb{B}^{\bullet}_{\mathtt{H}})^{?_{\mathrm{L}}}$, making e high and the process as a whole typable.

4.4 Noninterference via Embedding

Basic properties of the embedding follow. Below $\Omega^{\tau}_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(\mathtt{fw}\langle uy\rangle^{\tau} | \mathtt{fw}\langle yu\rangle^{\tau})$. Throughout the rest of the section we consider the typability in π^{LA} incorporating both subtyping and inflation.

Proposition 4.8

(*1*) (typability) *If* $E \vdash M : T$, *then* $\vdash_{\mathsf{sec}} [\![M]\!]_u \triangleright \langle T\rangle^{E}_u$ *is securely typed.*

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

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

Proof. **(1)** is straightforward induction, using (Subs) and Lemma 4.6 (3) for [*Var*] and [*UnitM*]; and (Inf) for [*App*] and [*BindM*]. For **(2)**, "only if" is by both-way simulation of reduction, following [11, § J.1]. **(3)** is standard, using (2). □

We believe the converse of (3) holds. The clause (1) may also be strengthened with its converse, assuming we use the explicitly typed version of π^{LA}.

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. Section 7. A *closing substitution* is the one which substitutes closed terms for 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 σ_1 and σ_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 σ_1 and σ_2 such that $E \vdash \sigma_1 \cong_s \sigma_2$, $M\sigma_1 \Downarrow$ iff $M\sigma_2 \Downarrow$.*

Proof. Assume $x : T \vdash M : \mathbb{O}_s$ (the reasoning trivially extends to multiple variables). Let $\vdash N_i : T$ $(i = 1, 2)$ with $\mathsf{protect}(T) \not\sqsubseteq s$. If $\mathsf{protect}(T) \sqsubseteq \mathsf{tamp}(T^\circ)$ then $\mathsf{protect}(T^\circ) \not\sqsubseteq s$ hence by Prop. 3.22 we obtain $[\![N_1]\!]_x \cong_s [\![N_2]\!]_x$ under $x : T^\circ$. Thus $x : T \vdash \{N_1/x\} \sim_s \{N_2/x\}$ implies $[\![N_1]\!]_x \cong_s [\![N_2]\!]_x$. Below (replication) indicates the use of the standard replication theorem [10, Proposition 7].

$$
\begin{array}{lll}
x : T \vdash \{N_1/x\} \sim_s \{N_2/x\} & & \\
\quad \Rightarrow & [\![N_1]\!]_x \cong_s [\![N_2]\!]_x & \text{(above)} \\
\quad \Rightarrow & (\nu x)([\![M]\!]_u | [\![N_1]\!]_x) \cong_s (\nu x)([\![M]\!]_u | [\![N_2]\!]_x) & \text{(congruency)} \\
\quad \Rightarrow & [\![M\{N_1/x\}]\!]_u \cong_s [\![M\{N_2/x\}]\!]_u & \text{(replication)} \\
\quad \Rightarrow & M\{N_1/x\} \cong_s^{\text{DCC}} M\{N_2/x\} & \text{(Prop. 4.8 (3))} \\
\quad \Rightarrow & M\{N_1/x\} \Downarrow \text{ iff } M\{N_2/x\} \Downarrow, & \text{(Def. 4.5)}
\end{array}
$$

hence done. □

Remark 4.11 (a stronger NI property through full abstraction) If the converse of Proposition 4.8 (3) (hence full abstraction) holds, which we firmly believe to be so, then we can strengthen Theorem 4.10 by replacing "$E \vdash \sigma_1 =_s \sigma_2$" with "$E \vdash \sigma_1 \cong_s \sigma_2$" where $E \vdash \sigma_1 \cong_s \sigma_2$ indicates $\sigma_1(x) \cong_s \sigma_2(x)$ for each $x \in \mathtt{dom}(\sigma_i)$. The proof is identical with the above except for the first step. This shows how a stronger embedding property can be used for establishing a stronger property for the source language by reflecting the result for the embedding.

Remark 4.12 ([*BindM*] and (Inf)) In Example 4.7 (6), we observed the need of (Inf) for justifying the encoding of [*BindM*] (which corrects our development in [32]). However the encoding of many significant usage of [*BindM*] in the original DCC are typable in the secrecy typing without (Inf) (which include [*Case*] and [*Seq*] in Figure 6). It may be worth studying in which practical situations [*BindM*] becomes indispensable in a way which necessitates (Inf) for its justification. A related discussion is also found in Abadi [1].

Remark 4.13 (extension to product) We conclude our discussion on call-by-name DCC by extending the encoding to product types. The following treatment comes from Hyland and Ong [37], which allows a clean embedding of pointed types and recursion (the encoding of call-by-value products, which is quite different, is treated in §4.6 later). We first stipulate the following equation on types (let $n \geq 1$ below).

$$[T_1..T_n(T'_1 \times T'_2)] \quad = \quad [T_1..T_nT'_1] \times [T_1..T_nT'_2]$$

By reading the equation as a rewrite rule (from the left to the right), any type can be rewritten into the following normal form:

$$\Pi_{1 \leq i \leq n}[\vec{T_i}\gamma_i] \quad \stackrel{\text{def}}{=} \quad [\vec{T_1}\gamma_1] \times ... \times [\vec{T_n}\gamma_n]$$

where, as before, each γ_i is either a unit, a sum or a lifted type, and each type in $\vec{T_i}$ is again a normal form. As seen from the above notation, we take a normal form up to associativity. We call a type of the form $[\vec{T_i}\gamma_i]$ *prime*. Prime types are ranged over by $\theta, \theta', \ldots$. We now define the encoding of types. Setting $\gamma^\bullet$ as before, T° is translated into a sequence of channel types.

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

$$(\Pi_{1 \leq i \leq n}\theta_i)^\circ \stackrel{\text{def}}{=} \theta_1^\circ \ldots \theta_n^\circ$$

Thus a product of DCC-types becomes a vector of channel types. In the above map, the dualisation is generalised to that on a vector of types in the obvious way.

The mapping of the environment and the action type needs an additional construction, which concerns how we treat a free variable. In brief, a free variable x of type $T = \Pi_{1 \leq i \leq n}\theta_i$ is decomposed into n names, say $x_1..x_n$, so that each x_i is given a prime type. We need a function which maps each free variable to a sequence of names, written ψ. We assume ψ always conforms to the given typing in the sense that, for example, if x is given a type $\mathbb{B}_{\mathtt{H}} \times \mathbb{B}_{\mathtt{H}}$, then x is mapped to a vector of length two, say x_1x_2. The notation $\vec{x} : \vec{\tau}$ stands for $\bigcup_i\{x_i : \tau_i\}$.

$$\text{(environment)} \quad \emptyset_\psi^\circ \stackrel{\text{def}}{=} \emptyset \qquad (E, x : T)_\psi^\circ \stackrel{\text{def}}{=} E_\psi^\circ, \psi x : \overline{T^\circ}$$

$$\text{(action)} \qquad \langle T \rangle_{\vec{u}}^{E,\psi} \stackrel{\text{def}}{=} \begin{cases} (\vec{u} : T^\circ \to A), B & T \text{ non-pointed, } E_\psi^\circ = ?_{\mathrm{L}}A, ?_{\mathrm{A}}B \\ \vec{u} : T^\circ, E_\psi^\circ & T \text{ pointed} \end{cases}$$

The encoding of terms follow that of types, written $[\![M : T]\!]_{\vec{u}}^\psi$ where, as above, ψ maps free variables in M. Among others, the pair $[\![\langle M_1, M_2\rangle]\!]_{\vec{u}_1\vec{u}_2}^\psi$ becomes $[\![M_1]\!]_{\vec{u}_1}^\psi \mid [\![M_2]\!]_{\vec{u}_1}^\psi$, the projection $[\![\pi_1(M)]\!]_{\vec{u}_1}^\psi$ becomes $(\boldsymbol{\nu}\,\vec{u}_2)[\![M]\!]_{\vec{u}_1\vec{u}_2}^\psi$, and the recursion $[\![\mu x^T.M : T]\!]_{\vec{u}}^\psi$ becomes $(\boldsymbol{\nu}\,\vec{x})([\![M : T]\!]_{\vec{u}}^{\psi \cdot x \mapsto \vec{x}} \mid \Pi \mathtt{fw}\langle x_i u_i\rangle^{\tau_i})$ where $T^\circ = \vec{\tau}$. Note this is typable iff all prime type in T° are affine, that is, iff T is pointed, conforming to the typing of recursion in Figure 6.

The encoding gives the semantic embedding of the full DCC in secrecy-enhanced π^{LA}, leading to the non-interference with exactly the same reasoning as we have done in the proof of Theorem 4.10,

4.5 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, developing a type-based secrecy analysis for call-by-value functional calculi by reflecting the secrecy analysis in π^{LA}. The resulting call-by-value calculus is useful when we consider integration of higher-order computation and 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 it from vDCC, the calculus is called DCCv. We use a syntax based on call-by-value PCF, which is convenient for our later applications. We first give the grammar of types, which use non-standard lifting motivated from the π^{LA}-encoding.

$$\begin{array}{llcl} \text{(common)} & T & ::= & S \mid U \\ \text{(total)} & S & ::= & \mathbb{N}_s \mid S \Rightarrow T \mid S_1 \times S_2 \mid S_1 +_s S_2 \\ \text{(partial)} & U & ::= & \llcorner S \lrcorner_s \end{array}$$

We call a type of form $S \Rightarrow U$ *pointed* (note pointed types are total). Notice we allow only total types to occur at the argument position of an arrow type. Further products and sums only use total types. These restrictions do not lead to a loss of generality, as we shall discuss in Remark 4.14 later. As before, we define the operation $(T)_s$ and the map $\mathsf{protect}(T)$.

—$(\mathbb{N}_s)_{s'} = (\mathbb{N}_{s \sqcup s'})$, $(\llcorner S \lrcorner_s)_{s'} = \llcorner S \lrcorner_{s \sqcup s'}$, $(S \Rightarrow T)_s = S \Rightarrow (T)_s$, $(S_1 \times S_2)_s = (S_1)_s \times (S_2)_s$ and $(S_1 +_s S_2)_{s'} = S_1 +_{s \sqcup s'} S_2$.

—$\mathsf{protect}(\mathbb{N}_s) = \mathsf{protect}(\llcorner S \lrcorner_s) = \mathsf{protect}(S_1 +_s S_2) = s$ and $\mathsf{protect}(S \Rightarrow T) = \mathsf{protect}(T)$.

Preterms are those of the standard PCFv extended with $\mathtt{seq}$, products and sums.

$$\begin{array}{rcl} M & ::= & n \mid \mathtt{succ}(M) \mid \mathtt{pred}(M) \mid x \mid \lambda x^S.M \mid MN \mid \langle M, N \rangle \mid \pi_i(M) \mid \\ & & \mathtt{in}_i(M) \mid \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{S_i}).M_i\} \mid \mu x^S.\lambda y^{S'}.M \mid \mathtt{seq}\ x^S{=}N\ \mathtt{in}\ M\ . \end{array}$$

$\mathtt{bind}$ is not used since it is redundant in our presentation, cf. Remark B.3. We often omit type annotations of bound variables for brevity.

The reduction $\longrightarrow$ in DCCv is the standard call-by-value one-step reduction, generated from the following rules together with closure under all contexts except λ-abstraction. V stands for either a variable, a natural number, a λ-abstraction or a recursion.

$$\begin{array}{rcl} \mathtt{succ}(n) & \longrightarrow & n+1 \\ \mathtt{pred}(n) & \longrightarrow & n-1 \\ (\lambda x.M)V & \longrightarrow & M\{V/x\} \\ \pi_1(\langle V_1, V_2 \rangle) & \longrightarrow & V_1 \\ \mathtt{case}\ \mathtt{in}_1(V)\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{S_i}).M_i\} & \longrightarrow & M_1\{V/x_1\} \\ (\mu x.\lambda y.M)V & \longrightarrow & M\{\mu x.\lambda y.M/x\}\{V/y\} \\ \mathtt{seq}\, x{=}V\, \mathtt{in}\, M & \longrightarrow & M\{V/x\} \end{array}$$

The typing uses the sequent $E \vdash M : T$, where the environment E is a finite map from variables to total types. The typing rules are given in Figure 8. $\mathtt{pred}(M)$ is typed as $\mathtt{succ}(M)$. The main difference from the typing rules for DCC (cf. Figure 6)

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

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

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

$$[\mathit{Pair}]\ \frac{E \vdash M_i : S_i \quad (i = 1,2)}{E \vdash \langle M_1, M_2\rangle : S_1 \times S_2} \qquad [\mathit{Proj}]\ \frac{E \vdash M : S_1 \times S_2}{E \vdash \pi_i(M) : S_i}$$

$$[\mathit{Inl}]\ \frac{E \vdash M : S_1}{E \vdash \mathtt{inl}(M) : S_1 +_s S_2} \qquad [\mathit{Case}]\ \frac{E \vdash M : S_1 +_s S_2 \quad E, x_i : S_i \vdash M_i : T}{E \vdash \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{S_i}).M_i\} : T}\ s \sqsubseteq \mathsf{protect}(T)$$

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

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

Fig. 8. Typing Rules of Call-by-Value DCC

is in the distinction between total types and partial types in DCCv. [*seq*] demands the resulting type to be partial, taking the secrecy level of a possibly diverging argument into account. This is the only rule in which the secrecy level of divergence is taken into consideration.

Remark 4.14 (partiality) We now discuss how two kinds of partial arrow types (cf. [17]) can be embedded in the present system.

- $U \rightharpoonup U'$, which is a total type. Intuitively, this type represents a closure which expects a possibly nonterminating argument and, therefore, produces a possibly nonterminating datum.
- $U \rightharpoonup_s U'$, which is a partial type (this notation restores the partial function space constructor in our original presentation in [33]). This is a partial version of $\rightharpoonup$, representing a divergence or, when convergent, a closure of type $\rightharpoonup$, and is convenient when we have successive partial applications.

These partial arrow types are used in the system whose environments use partial types. Two rules which use partial arrow types are:

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

These partial arrow types and their typing rules can be encoded into total arrow types by forgetting partiality at the level of types (translating $U \rightharpoonup U'$ into $S \Rightarrow U'$ and $U \rightharpoonup_s U'$ into $\llcorner U \rightharpoonup U' \lrcorner_s$, both with $U \overset{\text{def}}{=} \llcorner S \lrcorner_s$), while regaining it at the level of terms. For example, by encoding the partial application MN into $\mathtt{seq}\, x = N\ \mathtt{in}\ Mx$ and applying [*Seq*], we can justify [*AppP*] above including its side condition. Note the partiality is already used when we apply [*Seq*]. Similarly, given $U_1 = \llcorner S_1 \lrcorner_{s_1}$ and $U_2 = \llcorner S_2 \lrcorner_{s_2}$, we can encode their product as $\llcorner S_1 \times S_2 \lrcorner_{s_1 \sqcup s_2}$ (for the use of $\sqcup$, see Example 4.18 (4) for illustration).

Two simple DCCv-terms follow.

Example 4.15 (DCCv-terms) Below we write $\mathbb{N}$ for $\mathbb{N}_{\mathtt{L}}$ for brevity.

(1) Assume $E \vdash N : \llcorner\mathbb{N}\lrcorner_{\mathtt{M}}$. Then $E,\, y : \mathbb{N} \Rightarrow \llcorner\mathbb{N}\lrcorner_{\mathtt{H}} \vdash \mathtt{seq}\ x = N\ \mathtt{in}\ yx : \llcorner\mathbb{N}\lrcorner_{\mathtt{H}}$ is well-typed, with $\mathtt{M}$ being a secrecy level between $\mathtt{H}$ and $\mathtt{L}$. The use of possibly diverging N in $\mathtt{seq}$, to be observed at level $\mathtt{M}$, is justified by having a high-level partial type for the whole term.

(2) Using [*LamP*] and [*AppP*] given above, as well as the notation $\rightharpoonup_s$, the sequent $\vdash \lambda x.\lambda y.xy : (\llcorner\mathbb{N}\lrcorner_{\mathtt{L}} \rightharpoonup_{\mathtt{L}} \llcorner\mathbb{N}\lrcorner_{\mathtt{H}}) \rightharpoonup_{\mathtt{L}} \llcorner\mathbb{N}\lrcorner_{\mathtt{L}} \rightharpoonup_{\mathtt{M}} \llcorner\mathbb{N}\lrcorner_{\mathtt{H}}$ 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 $\mathtt{L}$. Next the result of application of the first argument may be observed at level $\mathtt{M}$. Finally the result of the second application may be observed at $\mathtt{H}$.

Proposition 4.16 (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 B. $\square$

Remark 4.17 (typing for call-by-value secrecy) The typing in Figure 8 simplifies our presentation in [32], while maintaining its essential features. In particular, [*LamP*] and [*AppP*] in [32] are derivable (via [*LamP*] and [*AppP*]), while [*Rec*] in [32] is the same as [*Rec*] in Figure 8 via the embedding of lifted partial arrow types to pointed types. The simplification is based on the analysis of the π^{LA}-encoding, given in the next subsection. Among others the encoding directly suggests the use of total types in environments and in contravariant positions (which is also an insight from the denotational study of partial computation [52]).

4.6 Embedding DCCv: Types

DCCv is strongly motivated by the projection of the secrecy analysis in Section 3 onto the standard process encoding of call-by-value functions. We first present the encoding of types.

$$
\begin{array}{lll}
\text{(type)} & S^{\bullet} \stackrel{\text{def}}{=} (S^{\circ})^{\uparrow_{\mathrm{L}}} & U^{\bullet}_s \stackrel{\text{def}}{=} (U^{\circ})^{\uparrow_{\mathrm{A}}}_s \ \ (s = \mathsf{protect}(U)) \\
& \mathbb{N}^{\circ}_s = ([\oplus_{i\in\mathbb{N}}\,]^{\uparrow_{\mathrm{L}}}_s)^{!_{\mathrm{L}}} & (S \Rightarrow T)^{\circ} \stackrel{\text{def}}{=} \begin{cases} (\overline{S^{\circ}}T^{\bullet})^{!_{\mathrm{L}}} & T \text{ total} \\ (\overline{S^{\circ}}T^{\bullet})^{!_{\mathrm{A}}} & T \text{ partial} \end{cases} \\
& (S_1 \times S_2)^{\circ} \stackrel{\text{def}}{=} ((S_1^{\circ}S_2^{\circ})^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}} & (S_1 +_s S_2)^{\circ} \stackrel{\text{def}}{=} ([S_1^{\circ} \oplus_s S_2^{\circ}]^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}} \\
& \llcorner S \lrcorner^{\circ}_s \stackrel{\text{def}}{=} S^{\circ} & \\
\text{(environment} & \emptyset^{\circ} \stackrel{\text{def}}{=} \emptyset \qquad (E,\, x : S)^{\circ} & \stackrel{\text{def}}{=} E^{\circ}, x : \overline{S^{\circ}} \\
\text{(action)} & \langle T \rangle^{E}_{u} \stackrel{\text{def}}{=} u : T^{\bullet}, E^{\circ} &
\end{array}
$$

In the standard process encoding of call-by-value computation, interaction starts from an output [46; 31; 18]. The encoding above reflects this idea, motivating the construction of DCCv-types. 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 the environments (as well as types in contravariant positions) uses the dual of $(\ )^\circ$, 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 an environment, 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 $?_{\mathrm{L}}$ and $?_{\mathrm{A}}$-actions, it directly emits (if ever) information at u, rather than getting invoked at it.

Example 4.18 (encoded DCCv types)

(1) $\mathbb{N} \Rightarrow \mathbb{N}$ and its (least-level) lifting $\llcorner\mathbb{N} \Rightarrow \mathbb{N}\lrcorner_{\mathtt{L}}$ are respectively translated as $((\overline{\mathbb{N}^\circ}\mathbb{N}^\bullet)^{!_{\mathrm{L}}})^{\uparrow_{\mathrm{L}}}$ and $((\overline{\mathbb{N}^\circ}\mathbb{N}^\bullet)^{!_{\mathrm{L}}})^{\uparrow_{\mathrm{A}}}_{\mathtt{L}}$. 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 [31]).

(2) The encoding of $S_1 \times S_2$ by $(\ )^\circ$ becomes $((S_1^\circ S_2^\circ)^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$, representing the behaviour which, when invoked, immediately returns a linear unary output of two data (cf. Example 3.6 (1)). Since these initial two actions have no information content, they are not secrecy annotated, illustrating the reason for the lack of a secrecy annotation in $T_1 \times T_2$.

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

(4) The encoding of a partial product $\llcorner S_1 \lrcorner_{s_1} \times \llcorner S_2 \lrcorner_{s_2} \stackrel{\text{def}}{=} \llcorner S_1 \times S_2 \lrcorner_{s_1 \sqcup s_2}$ (cf. Remark 4.14) via $(\ )^\circ$, becomes $((S_1^\circ S_2^\circ)^{\uparrow_{\mathrm{A}}}_{s_1 \sqcup s_2})^{!_{\mathrm{A}}}$. Intuitively, $\llcorner S_1 \lrcorner_{s_1} \times \llcorner S_2 \lrcorner_{s_2}$ waits for two data to converge and emits the resulting data, so that the level of the final output can only be higher than two convergence levels.

(5) $\langle \llcorner\mathbb{N}\lrcorner_{\mathtt{H}} \rangle^E_u$ with $E \stackrel{\text{def}}{=} x : \mathbb{N}$ is encoded as $u : (\mathbb{N}^\circ)^{\uparrow_{\mathrm{A}}}_{\mathtt{H}}, x : \overline{\mathbb{N}^\circ}$. 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 4.19 $\mathsf{protect}(T) = \mathsf{tamp}(T^\bullet)$ *for each* T. *Further* $\mathsf{tamp}(\langle T \rangle^E_u) = \mathsf{tamp}(T^\bullet)$ *for each* T, u *and* E.

$$\langle x\rangle_u \stackrel{\text{def}}{=} \overline{u}\langle x\rangle \qquad \langle n\rangle_u \stackrel{\text{def}}{=} \overline{u}(c)[\![n]\!]_c \qquad \langle \mathtt{succ}(y)\rangle_u \stackrel{\text{def}}{=} \overline{u}(c)!c(e).\overline{y}(c')c'[\&_i\ .\overline{e}\mathtt{in}_{i+1}]$$

$$\langle \lambda x.M\rangle_u \stackrel{\text{def}}{=} \overline{u}(c)!\, c(xm).\langle M\rangle_m$$

$$\langle MN\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle M\rangle_m \mid m(a).(\boldsymbol{\nu}\, n)(\langle N\rangle_n \mid n(b).\overline{a}\langle bu\rangle)$$

$$\langle M_1, M_2\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m_1)(\langle M_1\rangle_{m_1} \mid m_1(c_1).(\boldsymbol{\nu}\, m_2)(\langle M_2\rangle_{m_2} \mid m_2(c_2).\overline{u}(c)!c(e).\overline{e}\langle c_1c_2\rangle)$$

$$\langle \pi_1(M)\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle M\rangle_m \ \mid\ m(c).\overline{c}(e)e(m_1m_2)\overline{u}\langle m_1\rangle)$$

$$\langle \mathtt{in}_1(M)\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle M\rangle_m \mid m(c).\overline{u}\mathtt{in}_1\langle c\rangle)$$

$$\langle \mathtt{case}\ N\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{S_i}).M_i\}\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle N\rangle_m \mid m[\&_i(x_i).\langle M_i\rangle_u]$$

$$\langle \mathtt{seq}\ x = N\ \mathtt{in}\ M\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, n)(\langle N\rangle_n \mid n(x).\langle M\rangle_u)$$

$$\langle \mu x.\lambda y.M\rangle_u \stackrel{\text{def}}{=} \overline{u}(c)(\boldsymbol{\nu}\, x)(P \mid !x(yz).\overline{c}\langle yz\rangle) \qquad (\langle \lambda y.M\rangle_u \equiv \overline{u}(c)P)$$

$$\langle \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle M\rangle_m | m(c).\overline{c}(e)e[\&_i\ .\langle N_i\rangle_u]) \quad (N_j = N_2 \text{ if } i \geq 2)$$

Fig. 9. Encoding of Call-by-Value DCC

4.7 Embedding DCCv: Terms and Noninterference

Figure 9 lists the encoding of DCCv-terms. The encoding is standard [46; 31; 18].

(1) $\langle V\rangle_u$ immediately outputs at u, having the shape $\overline{u}\langle x\rangle$ if $V = x$, $\overline{u}(c)P$ 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) For the encoding of a pair, consider (as our default treatment says) that a product pairs total types. Then a pair is essentialy that of values. Let $\langle V_i\rangle_u \stackrel{\text{def}}{=} \overline{u}(m_i)P_i$ $(i = 1, 2)$. By applying simple syntactic equality to the encoding in Figure 9, we obtain $\langle V_1, V_2\rangle_u = \overline{u}(m).\overline{c}(m_1m_2)(P_1 \mid P_2)$. By noting u is linear, true information of this behaviour only gets unfolded from m_1 and m_2 (hence the level of a total product is the meet of the levels of its components).

(4) The encoding of a pair also elucidates information flow in partial pairings (cf. Remark 4.14 and Example 4.18 (4)). Assume, in the encoding of a pair in Figure 9, that M_1 and M_2 have partial types. Then m_1 and m_2 are affine; since u is outputted after inputting at m_1 and m_2, the level of u can only be the same as, or higher than, the levels of m_1 and m_2. Thus taking the join makes sense for a partial product (observationally, this says convergence of a partial pair can only be observed after both components converge, whereas one can freely extract either component and observe its behaviour in a total product).

(5) `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 (since, in DCCv, x is pointed).

The typing rules of DCCv are easily justifiable by the secrecy typing in π^{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 $E, x : S \vdash M : U$. Assuming the encoding of these terms are well-typed, our purpose is to make the following derivation secure.

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

Secrecy-wise, the only non-trivial inference is for the affine input "$n(x)$", using the secure version of $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$, which demands $s \sqsubseteq \mathsf{tamp}(U^{\bullet}) = \mathsf{protect}(U)$ (cf. Proposition 4.19), reaching the side condition in [*Seq*]. Typability of other encodings can similarly be verified, so that we obtain:

Proposition 4.20 (typability) *If* $E \vdash M : T$ *in* DCCv *then* $\vdash_{\mathsf{sec}} \langle M\rangle_u \rhd \langle T\rangle^E_u$ *is well-typed in the secrecy analysis with inflation.*

As before, the non-interference in DCCv is proved via Proposition 4.20, soundness and computational adequacy. The argument is identical except for the use of observable at $\llcorner \mathbb{N} \lrcorner_s$. We conclude:

Theorem 4.21 (non-interference) *Let* $E \vdash M : \llcorner \mathbb{N} \lrcorner_s$. *Then for any closing* σ_1 *and* σ_2 *such that* $E \vdash \sigma_1 \sim_s \sigma_2$, $M\sigma_1 \Downarrow$ *iff* $M\sigma_2 \Downarrow$.

5. STATE IN LINEAR/AFFINE π-CALCULUS

5.1 Reference Agent

The purpose of this section is to introduce a simple extension of π^{LA} 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 behaviours), we can define this agent as follows.

$$\mathsf{Ref}\langle xv\rangle = x[(c).(\mathsf{Ref}\langle xv\rangle|\overline{c}\langle v\rangle)\&(v'e).(\mathsf{Ref}\langle xv'\rangle|\overline{e})] \tag{13}$$

In $\mathsf{Ref}\langle xv\rangle$, x is its principal channel and v is (the name of) its stored value. This process waits for invocation with two branches at x, with its left branch for reading and its right branch for writing. The read branch receives from the request a single name c as a continuation. This continuation 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 acknowledges the receipt of the new value via e. Writing down these behaviours as reduction, we reach the reduction rules for $\mathsf{Ref}\langle xv\rangle$. Below we write `read` and `write` to denote read and write actions to reference agents (instead of writing `inl` and `inr`) to gain the readability.

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

This agent may as well be introduced into the sequential version of π^{LA}: if we do so, we obtain a class of sequential behaviours which combines general state and higher-order computation. In the present paper, we combine π^{LA} as given (which does not have the sequentiality constraint [10]) with references. As is well-known, combination of state and concurrency leads to a loss of Church-Rosser property via interference [39; 40; 44], as the following simple example shows.

$$R \stackrel{\text{def}}{=} \mathsf{Ref}\langle x1\rangle \mid \overline{x}\,\mathtt{write}(2c)c.\mathbf{0} \mid \overline{x}\,\mathtt{read}(c)c(y).\overline{y}(e)e[.\overline{u}\langle v\rangle \,\&\, .\Omega_u] \tag{14}$$

where, for legibility, we write $\mathsf{Ref}\langle xn\rangle$ for $(\boldsymbol{\nu}\, v)(\mathsf{Ref}\langle xv\rangle \mid [\![n]\!]_v)$, while $\overline{x}\,\mathtt{write}(nc)P$ stands for $\overline{x}\,\mathtt{write}(wc)([\![n]\!]_w \mid P)$. By the racing condition at x, this agent may or may not emit an output at u, i.e. if the write action interacts with the reference first, then R diverges, while if the read action reaches first, then it terminates as seen in the following reductions:

$$\begin{array}{llll} R \longrightarrow^+ \mathsf{Ref}\langle x2\rangle \mid \overline{x}\,\mathtt{read}(c)c(y).\overline{y}(e)e[.\overline{u}\mathtt{in}_n \,\&\, .\Omega_u] & \longrightarrow^+ & \mathsf{Ref}\langle x2\rangle \mid \Omega_u & \text{or} \\ R \longrightarrow^+ \mathsf{Ref}\langle x1\rangle \mid \overline{x}\,\mathtt{write}(2c)c.\mathbf{0} \mid \overline{u}\mathtt{in}_n & \longrightarrow^+ & \mathsf{Ref}\langle x2\rangle \mid \overline{u}\mathtt{in}_n & \end{array}$$

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 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.

$$\mathsf{Counter}\langle x\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(!x(f).\overline{y}\,\mathtt{read}(c)c(n).\overline{y}\,\mathtt{write}(n{+}1, e)e.\overline{f}\langle n\rangle|\mathsf{Ref}\langle y0\rangle) \tag{15}$$

where $\overline{y}\,\mathtt{write}(n{+}1,\, e)P \stackrel{\text{def}}{=} \overline{y}\,\mathtt{write}(me)([\![\mathsf{succ}\langle n\rangle]\!]_m|P)$, with $[\![\mathsf{succ}\langle n\rangle]\!]_m$ being 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 process first reads the value n stored in a local reference and write $n+1$ to it, and finally returns n to a channel f.

In the light of its expressiveness as well as for the sake of a clean presentation, we incorporate stateful behaviours into π^{LA} by introducing a reference agent as a constant. The grammar 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 (14) above.

As we already observed, the incorporation of references into π^{LA} results in non-determinism.

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 in π^{LA} while seamlessly integrating them with stateful behaviour. The following action modes are additionally used.

$$\mathbf{!}_{\mathrm{R}} \quad \text{Reference server} \qquad \mathbf{?}_{\mathrm{R}} \quad \text{Client requests to } \mathbf{!}_{\mathrm{R}}$$

$\mathbf{!}_{\mathrm{R}}$ and $\mathbf{?}_{\mathrm{R}}$ are mutually dual. We add $\mathbf{!}_{\mathrm{R}}$ (resp. $\mathbf{?}_{\mathrm{R}}$) to $\mathcal{M}_{\mathbf{!}}$ (resp. $\mathcal{M}_{\mathbf{?}}$).

Channel Types. The channel types are extended by the following syntax.

$$\begin{array}{lll} \tau_{\mathrm{I}} & ::= & \ldots \ \mid \ \mathsf{ref}\langle\tau\rangle \ \mid \ \mathsf{refr}\langle\tau\rangle \ \mid \ \mathsf{refw}\langle\tau\rangle \\ \tau_{\mathrm{O}} & ::= & \ldots \ \mid \ \mathsf{rw}\langle\tau\rangle \ \mid \ \mathsf{r}\langle\tau\rangle \ \mid \ \mathsf{w}\langle\tau\rangle \end{array}$$

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

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

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

Fig. 10. Typing Rules for Reference

These added types are abbreviations for branching/selection types which represent the behaviours of references and the behaviours which interact with references. First, $\mathsf{ref}\langle\tau\rangle$ is a type of a reference agent whose value has type τ.

$$\mathsf{ref}\langle\tau\rangle \stackrel{\text{def}}{=} [(\tau)^{\uparrow_{\mathrm{L}}} \& \overline{\tau}()^{\uparrow_{\mathrm{L}}}]^{!_{\mathrm{R}}}$$

where we demand $\mathsf{md}(\tau) \in \mathcal{M}_{!}$. $\mathsf{rw}\langle\tau\rangle$, with $\mathsf{md}(\tau) \in \mathcal{M}_{?}$, is its dual:

$$\mathsf{rw}\langle\tau\rangle \stackrel{\text{def}}{=} [(\tau)^{\downarrow_{\mathrm{L}}} \oplus \overline{\tau}()^{\downarrow_{\mathrm{L}}}]^{?_{\mathrm{R}}}.$$

The remaining types are subtypes of these types. First, $\mathsf{refr}\langle\tau\rangle$ (resp. $\mathsf{refw}\langle\tau\rangle$) with $\mathsf{md}(\tau) \in \mathcal{M}_{!}$ is the types for a read-only (resp. write-only) reference, so that:

$$\mathsf{refr}\langle\tau\rangle \stackrel{\text{def}}{=} [(\tau)^{\uparrow_{\mathrm{L}}} \& \_]^{!_{\mathrm{R}}}, \qquad \mathsf{refw}\langle\tau\rangle \stackrel{\text{def}}{=} [\_ \& \overline{\tau}()^{\uparrow_{\mathrm{L}}}]^{!_{\mathrm{R}}}.$$

Finally, $\mathsf{r}\langle\tau\rangle$ (resp. $\mathsf{w}\langle\tau\rangle$) with $\mathsf{md}(\tau) \in \mathcal{M}_{?}$ is the dual of $\mathsf{refr}\langle\tau\rangle$ (resp. $\mathsf{refw}\langle\tau\rangle$).

$$\mathsf{r}\langle\tau\rangle \stackrel{\text{def}}{=} [(\tau)^{\downarrow_{\mathrm{L}}} \oplus \_]^{?_{\mathrm{R}}}, \qquad \mathsf{w}\langle\tau\rangle \stackrel{\text{def}}{=} [\_ \oplus \overline{\tau}()^{\downarrow_{\mathrm{L}}}]^{?_{\mathrm{R}}}.$$

Observe a reference contains 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. The definitions of $\asymp$ and $\odot$ follow precisely those given for general replicated types.

Typing. The additional typing rules for stateful actions are given by Figure 10 (which may be understood in the light of the reduction rules in (14) of § 5.1). These rules are combined with those in Figure 3, where $?A$ now indicates A may include $?_{\mathrm{R}}$-types (so that $(\mathsf{In}^{!_A})$ and $(\mathsf{Bra}^{!_A})$ prefix free write actions).

We also use the subsumption, in which we conclude $\vdash P \triangleright A$ from $\vdash P \triangleright B$ if $B \leq A$, where $\leq$ is induced by the identity on non-reference types together with:

$$\frac{\tau \leq \tau'}{\mathsf{r}\langle\tau\rangle \leq \mathsf{r}\langle\tau'\rangle} \quad \frac{\tau' \leq \tau}{\mathsf{w}\langle\tau\rangle \leq \mathsf{w}\langle\tau'\rangle} \quad \frac{\tau \leq \tau'}{\mathsf{r}\langle\tau\rangle \leq \mathsf{rw}\langle\tau'\rangle} \quad \frac{\tau' \leq \tau}{\mathsf{w}\langle\tau\rangle \leq \mathsf{rw}\langle\tau'\rangle} \quad \frac{}{\mathsf{rw}\langle\tau\rangle \leq \mathsf{rw}\langle\tau\rangle}$$

In these rules, the value type τ appears in the *covariant* position in the read type, and in the *contravariant* position in the write type. Hence the ordering of the value is covariant in the read type, while contravariant in the write type. In the last rule, we cannot vary τ since τ in $\mathsf{rw}\langle\tau\rangle \stackrel{\text{def}}{=} [(\tau)^{\downarrow_{\mathrm{L}}} \& \overline{\tau}()^{\downarrow_{\mathrm{L}}}]^{?_{\mathrm{R}}}$ occurs both as itself (covariant position) and as its dual (contravariant position), cf. [59].

The subtyping based on distinction between read and write capabilities will be used for the fine-grained secrecy analysis in Section 6, integrated with the secrecy subtyping discussed in § 3.4.

Remark 5.1 (read/write subtyping) The subtyping based on read/write capabilities is a special case of a more general subtyping relation generated from $[\vec{\tau}_1 \oplus .. \oplus \vec{\tau}_n] \leq [\vec{\tau}'_1 \oplus .. \oplus \vec{\tau}'_{n+m}]$ with $\tau_{ij} \leq \tau'_{ij}$ $(1 \leq j \leq n)$, and dually, which is covariant in carried types (this subtyping has close connection to the subtyping in functional calculi). The present study however only needs its restriction to reference types.

Remark 5.2 (linearity and reference) By definition, "$?_{\mathrm{L}}A, ?_{\mathrm{A}}B$" in the premise of $(\mathsf{In}^{!_{\mathrm{L}}})$ and $(\mathsf{Bra}^{!_{\mathrm{L}}})$ do *not* include read/write actions. This is for ensuring linearity. Take $P \stackrel{\text{def}}{=} !u(z).\overline{x}\,\mathtt{read}(c)c(y).\overline{y}(w)w.\overline{z}$. Then $P|\mathsf{Ref}\langle xu\rangle|\overline{u}\langle z\rangle$ diverges, so we should not type u with $(()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$ and x with $\mathsf{ref}\langle(()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}\rangle$. However if we were to allow $?_{\mathrm{L}}A$ to include a read action, we could (wrongly) assign these types to u and x of $P|\mathsf{Ref}\langle xu\rangle$. We can enlarge the typablity through the use of effect typing [9], where non-linearity smuggled via free references (as in P above) gets exposed as a recursion in types.

A few examples of stateful processes follow.

Example 5.3 (reference)

(1) (newref) In ML, $\mathsf{ref}\,M$ 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

$$[\![\lambda m.\mathtt{new}\ y \mapsto m\ \mathtt{in}\ y]\!]_u \stackrel{\text{def}}{=} !u(mz).\overline{z}(y)\mathsf{Ref}\langle ym\rangle,$$

which is well-typed under $u\!:\!(\overline{\tau}(\mathsf{ref}\langle\tau\rangle)^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$ assuming the type of m is τ. The process, when invoked, receives a value v and a return channel z, and finally sends, via z, a pointer to a new reference with value v.

(2) (read) In ML, $\mathbf{!}M$ indicates the result of evaluating M into a reference label and reading from it. To represent this operation as a process, we again first decompose it into $\lambda m.(\mathtt{let}\ x = \mathbf{!}\ m\ \mathtt{in}\ x)$, using finer operations. We can then represent this expression as

$$[\![\lambda m.(\mathtt{let}\ x = \mathbf{!}\ m\ \mathtt{in}\ x)]\!]_u \stackrel{\text{def}}{=} !u(mz).\overline{m}\,\mathtt{read}(c)c(x).\overline{z}\langle x\rangle.$$

The name u of this process has the typing $u\!:\!(\mathsf{rw}\langle\overline{\tau}\rangle(\tau)^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}$, assuming the type of x is τ. The process receives a value v and a continuation z upon invocation, reads variable m, and finally sends, via z, a value stored in m.

(3) (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 $\langle\lambda y.\mathbf{!}x,\ \lambda v.x := v\rangle$. We can represent this behaviour as the following process:

$$Var_{rwx} \stackrel{\text{def}}{=} !r(c).\overline{x}\,\mathtt{read}(c')c'(v).\overline{c}\langle v\rangle \mid !w(vc).\overline{x}\,\mathtt{write}\langle vc\rangle$$

using the encoding of the λ-abstraction and a pair. We can then type this agent as $\vdash Var_{rwx} \triangleright r : ((\tau)^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}},\ w : (\overline{\tau}()^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}},\ x : \mathsf{rw}\langle\overline{\tau}\rangle$.

The stateful extension of π^{LA}, denoted π^{LAR}, satisfies the subject reduction (which is established for its secure version in the next section) and allows faithful embedding of languages with imperative features. Using π^{LAR}, the next section develops a theory of secrecy analysis for imperative, concurrent computation.

6. SECRECY WITH STATE

6.1 Secrecy Annotation on Channel Types

The presence of state changes how information flows among processes and, therefore, how we may guarantee safety of flows. There are two main aspects of information flow with state which are worth noting.

(1) *An action of writing can transmit information.* By writing to a reference, the behaviour of another agent which reads from that reference is affected, so we should take this action into account when we consider the tamper level.
(2) *An output to a replicated action can transmit information.* A replicated process can directly or indirectly write to a reference, thus transmitting information. Note a reference to be written can be hidden.

These observations suggest the following annotations for stateful types.

(1) A reference type will be annotated by a secrecy level, indicating the level at which it receives a write action, and dually.
(2) A replicated type will be annotated by a secrecy level, indicating the level at which it receives an invocation, and dually.

We thus arrive at the following refined grammar of channel types.

$$\begin{array}{lcl}\tau_{\mathrm{I}} & ::= & \ldots \mid (\vec{\tau})^{!_{\mathrm{L}}}_{s} \mid [\&_{i\in I}\,\vec{\tau}_i]^{!_{\mathrm{L}}}_{s} \mid (\vec{\tau})^{!_{\mathrm{A}}}_{s} \mid [\&_{i\in I}\vec{\tau}_i]^{!_{\mathrm{A}}}_{s} \mid \mathsf{ref}_s\langle\tau\rangle \mid \mathsf{refr}_s\langle\tau\rangle \mid \mathsf{refw}_s\langle\tau\rangle \\ \tau_{\mathrm{O}} & ::= & \ldots \mid (\vec{\tau})^{?_{\mathrm{L}}}_{s} \mid [\oplus_{i\in I}\,\vec{\tau}_i]^{?_{\mathrm{L}}}_{s} \mid (\vec{\tau})^{?_{\mathrm{A}}}_{s} \mid [\oplus_{i\in I}\,\vec{\tau}_i]^{?_{\mathrm{A}}}_{s} \mid \mathsf{rw}_s\langle\tau\rangle \mid \mathsf{r}_s\langle\tau\rangle \mid \mathsf{w}_s\langle\tau\rangle\end{array}$$

We annotate read-only types with secrecy annotations, which is needed for subtyping (to be discussed soon). The carried types obey the same conditions as their non-secrecy counterpart (for example, $\mathsf{md}(\tau) \in \mathcal{M}_!$ for $\mathsf{ref}_s\langle\tau\rangle$). As before, we may regard reference types and their duals as replicated branching types, e.g. $\mathsf{ref}_s\langle\tau\rangle \stackrel{\mathrm{def}}{=} [(\tau)^{\uparrow_{\mathrm{L}}}\&\overline{\tau}()^{\uparrow_{\mathrm{L}}}]^{!_{\mathrm{R}}}_{s}$.

We define the subtyping relation on these channel types, which integrates secrecy subtyping in § 3.4 and read/write subtyping in § 5.2. Thus the subtyping rules on reference types and their duals have now become:

$$\frac{\tau \le \tau' \quad s \sqsubseteq s'}{\mathsf{r}_s\langle\tau\rangle \le \mathsf{r}_{s'}\langle\tau'\rangle} \qquad \frac{\tau' \le \tau \quad s \sqsubseteq s'}{\mathsf{w}_s\langle\tau\rangle \le \mathsf{w}_{s'}\langle\tau'\rangle} \qquad \frac{\tau \le \tau' \quad s \sqsubseteq s'}{\mathsf{r}_s\langle\tau\rangle \le \mathsf{rw}_{s'}\langle\tau'\rangle} \qquad \frac{\tau' \le \tau \quad s \sqsubseteq s'}{\mathsf{w}_s\langle\tau\rangle \le \mathsf{rw}_{s'}\langle\tau'\rangle} \qquad \frac{s \sqsubseteq s'}{\mathsf{rw}_s\langle\tau\rangle \le \mathsf{rw}_{s'}\langle\tau\rangle}$$

$(\vec{\tau})^{?_{\mathrm{A}}}_{s}$ and $[\oplus_i\vec{\tau}_i]^{?_{\mathrm{A}}}_{s}$ are treated as $(\vec{\tau})^{\uparrow_{\mathrm{A}}}_{s}$ and $[\oplus_i\vec{\tau}_i]^{\uparrow_{\mathrm{A}}}_{s}$ in § 3.4 (and dually by the duality rule). Hence the subtyping rules in § 3.4 are replaced by:

$$\frac{\tau_i \le \tau_i'}{(\vec{\tau})^{\uparrow_{\mathrm{L}}} \le (\vec{\tau}')^{\uparrow_{\mathrm{L}}}} \qquad \frac{p_{\mathrm{O}} \ne \uparrow_{\mathrm{L}} \quad \tau_i \le \tau_i' \quad s \sqsubseteq s'}{(\vec{\tau})^{p_{\mathrm{O}}}_{s} \le (\vec{\tau}')^{p_{\mathrm{O}}}_{s'}} \qquad \frac{\tau_{ij} \le \tau_{ij}' \quad s \sqsubseteq s'}{[\oplus_i\vec{\tau}_i]^{p_{\mathrm{O}}}_{s} \le [\oplus_i\vec{\tau}_i']^{p_{\mathrm{O}}}_{s'}} \qquad \frac{\overline{\tau_{\mathrm{I}}'} \le \overline{\tau_{\mathrm{I}}}}{\tau_{\mathrm{I}} \le \tau_{\mathrm{I}}'}$$

6.2 Tampering Level of Stateful Types

For the calculation of tampering levels, we use distinction between read actions and write actions, which is important for capturing flow of information accurately: for example if x is typed as $\mathsf{r}_s\langle\mathbb{N}_{s'}\rangle$ in some process, then the process cannot affect the environment through x, simply because x can only be used for reading. A further discussion on this point will be given later. Incorporating the read/write distinction, the tampering level is now given as follows.

Definition 6.1 (tampering levels) $\mathsf{tamp}(\tau)$ is defined by the same clauses as Definition 3.4, except:

(1) $(\vec{\tau})^{?_{\mathrm{L}}}_{s}$, $[\oplus_i \vec{\tau}_i]^{?_{\mathrm{L}}}_{s}$, $(\vec{\tau})^{?_{\mathrm{A}}}_{s}$, $[\oplus_i \vec{\tau}_i]^{?_{\mathrm{A}}}_{s}$, $\mathsf{rw}_s\langle\tau\rangle$ and $\mathsf{w}_s\langle\tau\rangle$ are immediately tampering;
(2) $\mathsf{tamp}(\mathsf{r}_s\langle\tau\rangle) = \mathsf{tamp}(\tau)$.

$\mathsf{tamp}(A)$ is given as Definition 3.4.

τ with $\mathbf{?}$-mode is immediately tampering because the invoked process $!x(\vec{y}).P$ in the environment may, after invocation, be engaged in free write actions, affecting the environment. However the read-only type $\mathsf{r}_s\langle\tau\rangle$ is *not* immediately tampering and its level coincides with that of τ. The following property is easily proved by the rule induction on $\leq$.

Proposition 6.2 *$\leq$ is a partial order and $\tau_1 \leq \tau_2$ implies $\mathsf{tamp}(\tau_1) \leq \mathsf{tamp}(\tau_2)$.*

Remark 6.3 (read/write distinction) We illustrate the significance of read/write distinction in secrecy analysis through examples. Consider the following imperative command (where x is an imperative variable of a boolean type; $\mathbf{!}x$ reads its content).

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

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

$$\vdash \overline{x}\,\mathtt{read}(c)c(b).\overline{b}(g)g[.[\![C_1]\!]_f\&.[\![C_2]\!]_f] \triangleright x : \mathsf{r}_s\langle\overline{\mathbb{B}^{\circ}_{s}}\rangle, A \tag{16}$$

where A indicates the unknown part of the action type. Note the first action of this process *reads* from x, which does not change the state of x.

For example, even if $s = \mathtt{L}$, if C_1 and C_2 tamper only at the high level, the command C as a whole should be regarded as a high-level command. In fact, the tampering level of x is calculated as $\mathsf{tamp}(\mathsf{r}_{\mathtt{L}}\langle\overline{\mathbb{B}^{\circ}_{\mathtt{L}}}\rangle) = \mathsf{tamp}(\overline{\mathbb{B}^{\circ}_{\mathtt{L}}}) = \mathtt{H}$.

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

$$[\![y := n]\!]_f \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, c)(\overline{y}\,\mathtt{write}\langle cf\rangle \mid [\![n]\!]_c) \tag{17}$$

where f is a channel for acknowledgement. The typing of the process in (16) can now be elaborated as follows.

$$x : \mathsf{r}_s\langle\overline{\mathbb{B}^{\circ}_{s}}\rangle, y : \mathsf{w}_{s'}\langle\overline{\mathbb{N}^{\circ}_{s'}}\rangle, f : ()^{\uparrow_{\mathrm{L}}} \tag{18}$$

(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 (18) to be typable, s' should also be high (cf. $(\mathsf{Bra}^{\downarrow_{\mathrm{L}}})$ in Figure 5), conforming to the treatment of conditionals in [66; 70].

6.3 Structural Security

A key element in secrecy typing for stateful behaviours is an additional well-formedness condition for reference types. It reflects a different way in which information leaks in stateful computing. A similar idea is found in SLam-Calculus in [21, Appendix], and is in fact implicit in many existing secrecy analyses for imperative languages [70]. We first state the condition, then illustrate the idea.

Definition 6.4 (structural security) τ is *structurally secure* if for each type occurring in τ, say τ', the following two conditions hold: (1) $\mathsf{sec}(\tau') \sqsubseteq \mathsf{tamp}(\tau')$ when $\mathsf{md}(\tau') = \mathbf{!}_{\mathrm{R}}$, and (2) $\mathsf{sec}(\tau') \sqsubseteq \mathsf{tamp}(\overline{\tau'})$ when $\mathsf{md}(\tau') = \mathbf{?}_{\mathrm{R}}$.

Convention 6.5 *Henceforth we assume all channel types we treat are structurally secure.*

As a simple example, $\mathsf{ref}_{\mathtt{L}}\langle \mathbb{B}_{\mathtt{H}} \rangle$ is structurally secure while $\mathsf{ref}_{\mathtt{H}}\langle \mathbb{B}_{\mathtt{L}} \rangle$ is not. The definition says a mutable type should have higher tampering levels in carried types than enclosing types. This is because a reference transmits information by:

(1) Receiving information when a datum is written; and

(2) Emitting information when a datum is read and used.

In (2), it suffices to measure the level only when the datum is used, since reading itself is a semantically innocuous operation. With this understanding, we need to make the level of (1) lower than, or the same as, the level of information in (2).

We illustrate the need of structural security using a simple example (from (14) in § 5.1). We explicitly annotate channels and values with secrecy levels.

$$R \stackrel{\text{def}}{=} \mathsf{Ref}\langle x1 \rangle \mid \overline{x}\,\mathtt{write}(2c)c.\mathbf{0} \mid \overline{x}\,\mathtt{read}(c)c(y).\overline{y}(e)e[.\overline{u}\langle v \rangle \,\&\, .\Omega_u]$$

As we already observed in § 5.1, the write action at the channel x by $\overline{x}\,\mathtt{write}(2c)c.\mathbf{0}$ may affect termination at a channel u. Now assume x has reference type $\mathsf{ref}_{\mathtt{H}}\langle \mathbb{N}_{\mathtt{L}} \rangle$ violating structural security in Definition 6.4. Then e in the above process has the low level. Suppose u in the above process has the low level. Since e has the low level too, $e[.\overline{u}\langle v \rangle \,\&\, .\Omega_u]$ is typable by $(\mathsf{Bra}^{\downarrow \mathrm{A}})$. For readability, we annotate the whole term explicitly by the levels.

$$\mathsf{Ref}\langle x^{\mathtt{H}} 1^{\mathtt{L}} \rangle \mid \overline{x}\,\mathtt{write}(2^{\mathtt{L}}c)c.\mathbf{0} \mid \overline{x}\,\mathtt{read}(c)c(y).\overline{y}(e)e^{\mathtt{L}}[.\overline{u}^{\mathtt{L}}\langle v \rangle \,\&\, .\Omega^{\mathtt{L}}_u]$$

Thus the high-level channel x affects an action at the low-level channel u. A similar example is easily constructed for sequential processes.

Remark 6.6 (an alternative to structural security) The above discussion indicates that, if we consider the reading action as a disclosure of information, then we can dispense with structural security. However this alternative method is sometimes too restrictive. An example is when a high-level datum is stored in a low-level store, which a low-level principal forwards to a high-level principal (which is surely safe). Further those cases when the alternative method allows storing a low-level datum in a high-level store are also directly treatable by the present method through the use of the subtyping. We shall illustrate these aspects using a concrete programming language in the next section, Remark 7.6.

$$(\mathsf{In}^{!_{\mathrm{L}}})\quad \frac{s \sqsubseteq \mathsf{tamp}(A,B) \qquad \vdash P \rhd \vec{y} : \vec{\tau}, ?_{\mathrm{L}}A^{\text{-}x}, ?_{\mathrm{A}}B^{\text{-}x}}{\vdash !\, x(\vec{y}).P \rhd (x : (\vec{\tau})^{!_{\mathrm{L}}}_{s} \!\rightarrow A), B}
\qquad
(\mathsf{Bra}^{!_{\mathrm{L}}})\quad \frac{s \sqsubseteq \mathsf{tamp}(A,B) \qquad \vdash P \rhd \vec{y}_i : \vec{\tau}_i, ?_{\mathrm{L}}A^{\text{-}x}, ?_{\mathrm{A}}B^{\text{-}x}}{\vdash !\, x[\&_i(\vec{y}_i).P_i] \rhd (x : [\&_i\vec{\tau}_i]^{!_{\mathrm{L}}}_{s} \!\rightarrow A), B}$$

$$(\mathsf{In}^{!_{\mathrm{A}}})\quad \frac{s \sqsubseteq \mathsf{tamp}(A) \qquad \vdash_{\mathsf{sec}} P \rhd \vec{y} : \vec{\tau}, ?A^{\text{-}x}}{\vdash_{\mathsf{sec}} !\, x(\vec{y}).P \rhd x : (\vec{\tau})^{!_{\mathrm{A}}}_{s}, A}
\qquad
(\mathsf{Bra}^{!_{\mathrm{A}}})\quad \frac{s \sqsubseteq \mathsf{tamp}(A) \qquad \vdash P_i \rhd \vec{y}_i : \vec{\tau}_i, ?A^{\text{-}x}}{\vdash !\, x[\&_i(\vec{y}_i).P_i] \rhd x : [\&_i\vec{\tau}_i]^{!_{\mathrm{A}}}_{s}, A}$$

Fig. 11. Secrecy Typing for State

6.4 Secrecy Typing with State

The secrecy typing for stateful processes are given in Figure 11 (which refines Figure 10). Other rules remain the same as before by replacing channel types by secrecy-annotated ones. For reference, we present the summary of all secrecy typing rules for π^{LAR} is given in Figure 19 in Appendix, the last page. $(\mathsf{In}^{!_{\mathrm{L}}})$ and $(\mathsf{Bra}^{!_{\mathrm{L}}})$ require the secrecy level s to be lower than the tampering level of body P, since each input directly receives information. The same holds for $(\mathsf{In}^{!_{\mathrm{A}}})$ and $(\mathsf{Bra}^{!_{\mathrm{A}}})$.

Some examples follow. Below and henceforth we write e.g. $(\vec{\tau})^{!_{\mathrm{L}}}$ (omitting the secrecy level) to indicate the omitted level is $\mathtt{H}$.

Example 6.7 (secrecy typing for state)

(1) Recall a process $[\![y := n]\!]_f$ in (17) in § 6.2. Then $\vdash_{\mathsf{sec}} [\![y := n]\!]_f \rhd y : \mathsf{w}_s\langle\overline{\mathbb{N}^{\circ}_{s'}}\rangle, f : ()^{\uparrow_{\mathrm{L}}}$ if $s \sqsubseteq s'$. The tampering level of this process is s.

(2) Recall a process $[\![\mathtt{if}\ \mathtt{!}\,x\ \mathtt{then}\ y := 1\ \mathtt{else}\ y := 2]\!]_f$ in (16) in § 6.2. Then we have: $\vdash_{\mathsf{sec}} [\![\mathtt{if}\ \mathtt{!}\,x\ \mathtt{then}\ y := 1\ \mathtt{else}\ y := 2]\!]_f \rhd x : \mathsf{r}_s\langle\overline{\mathbb{B}^{\circ}_{s_b}}\rangle, y : \mathsf{w}_{s'}\langle\overline{\mathbb{N}^{\circ}_{s_n}}\rangle, f : ()^{\uparrow_{\mathrm{L}}}$ if $s \sqsubseteq s_b \sqsubseteq s' \sqsubseteq s_n$. The tampering level of this process is s'.

(3) Let $[\![\mathtt{new}\ y \mapsto x\ \mathtt{in}\ y]\!]_m \stackrel{\text{def}}{=} \overline{m}(y)\mathsf{Ref}\langle yx\rangle$ (cf. Example 5.3 (1)). Then we have $\vdash_{\mathsf{sec}} [\![\mathtt{new}\ y \mapsto x\ \mathtt{in}\ y]\!]_m \rhd x : \overline{\tau}, m : (\mathsf{ref}_s\langle\tau\rangle)^{\uparrow_{\mathrm{L}}}$ with $s \sqsubseteq \mathsf{tamp}(\mathsf{ref}_s\langle\tau\rangle) = \mathsf{tamp}(\tau) \sqcap \mathsf{tamp}(\overline{\tau})$. The tampering level of this process is $\mathsf{tamp}(\tau) \sqcap \mathsf{tamp}(\overline{\tau})$.

(4) Recall the process representing $[\![\lambda x.\mathtt{new}\ y \mapsto x\ \mathtt{in}\ y]\!]_u$ from Example 5.3 (1). Then it is typable by type $u : (\overline{\tau}(\mathsf{ref}_s\langle\tau\rangle)^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$ with $s \sqsubseteq \mathsf{tamp}(\tau) \sqcap \mathsf{tamp}(\overline{\tau})$ (since this process does not write on any free reference), and its tampering level is the same as above, i.e. $\mathsf{tamp}(\tau) \sqcap \mathsf{tamp}(\overline{\tau})$.

(5) Let $[\![\mathtt{let}\ x = \mathtt{!}y\ \mathtt{in}\ x]\!]_z \stackrel{\text{def}}{=} \overline{y}\,\mathtt{read}(c)c(x).\overline{z}\langle x\rangle$ (cf. Example 5.3 (2)). Then $\vdash_{\mathsf{sec}} [\![\mathtt{let}\ x = \mathtt{!}y\ \mathtt{in}\ x]\!]_z \rhd z : (\mathsf{ref}_s\langle\tau\rangle)^{\uparrow_{\mathrm{L}}}, y : \mathsf{r}_s\langle\overline{\tau}\rangle$ is well-typed if $s \sqsubseteq \mathsf{tamp}(\tau) \sqcap \mathsf{tamp}(\overline{\tau})$. The tampering level of this process is $\mathsf{tamp}(\tau) \sqcap \mathsf{tamp}(\overline{\tau})$.

(6) Recall $\mathsf{Counter}\langle x\rangle$ in (15) in § 5.1. Then we have: $\vdash_{\mathsf{sec}} \mathsf{Counter}\langle x\rangle \rhd x : ((\mathbb{N}^{\circ}_{s_n})^{\uparrow_{\mathrm{A}}}_{s})^{!_{\mathrm{A}}}_{s'}$ if $s' \sqsubseteq s \sqsubseteq s_n$. The tampering level of this process is s.

6.5 Basic Properties of Secrecy Typing in π^{LAR}

In the following we summarise the key properties of the secure π^{LA}. We start from the subject reduction theorem.

Proposition 6.8 (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 3.7, we first prove the substitution lemma. The remaining interesting case is references, which is similarly proved as in [73, Proposition 3]. See Appendix A for the full proof. □

There are several ways for defining a secrecy-sensitive contextual congruence for π^{LAR}, which differ in the ways of treating branching structures of nondeterministic state change. Here we use the clause identical with the one given in Definition 3.9, Section 3 (which we reproduce below), by which we obtain a version of May-equivalence.

Definition 6.9 (secrecy-sensitive contextual congruence) Fix some s. Then *s-sensitive contextual congruence*, denoted $\cong_s$, is the maximum typed congruence which satisfies the following condition: whenever $\vdash_{\mathsf{sec}} P_1 \cong_s P_2 \triangleright x : ()^{\uparrow_{\mathrm{A}}}_{s'}$ such that $s' \sqsubseteq s$, we have $P_1 \Downarrow_x$ iff $P_2 \Downarrow_x$.

Proposition 6.10 (non-interference) *Let* $\vdash_{\mathsf{sec}} P_i \triangleright A$ $(i = 1, 2)$ *such that* $\mathsf{tamp}(A) = s$. *Then* $s \not\sqsubseteq s'$ *implies* $\vdash_{\mathsf{sec}} P_1 \cong_{s'} P_2 \triangleright A$.

The statement is literally the same as the noninterference theorem for stateless processes (Proposition 3.12). The proof of Proposition 6.10 is given in [33], which uses, following the proof of noninterference of secure π^{LA}-processes, an inductive causality analysis of stateful processes.

Remark 6.11 (alternative formulations of noninterference)

(1) In the presence of nondeterminism, it is often necessary to use equivalences which capture branching structure due to nondeterministic state change, such as failure/testing equivalences and bisimulations. In the present setting, one may use the equality based on reduction-closure [30], for which we simply add the following clause to Definition 6.9: *whenever* $P \cong_s Q$ *and* $P \longrightarrow P'$, *we have* $Q \twoheadrightarrow Q'$ *such that* $P' \cong_s Q'$ *for some* Q'. We conjecture that the non-interference property as stated in Proposition 6.10 also holds for this refined equality (which implies Proposition 6.10 since the reduction-closed congruence is strictly smaller than $\cong_s$ as we are presently using): the proof may use the bisimulation-based method in [75].

(2) Another possible extension is the incorporation of inflation. We believe some form of the inflation following the basic framework of §3.5 can be incorporated into the present system, even though a precise understanding and sound incorporation of the notion of inflation in the concurrent, stateful setting is left as an open issue.

7. CONCURRENCY, REFERENCE AND PROCEDURE

In this section we use the secrecy analysis in π^{LAR} 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, extended with higher-order procedures and general references. The typing rules are directly suggested from the secrecy typing in π^{LAR}. We discuss the significance of fine-grained secrecy typing on imperative features coming from π^{LAR}, and establish the noninterference properties of the language through its embedding in π^{LAR}.

7.1 A Volpano-Smith Language

We first 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.

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

The syntax of commands is from [66], extended with general references, local variables and higher-order procedures. We use `let` and `seq` for clearer presentation of typing rules, though in examples we often write e.g. "$x := \mathtt{!}y$" (which stands for "$\mathtt{let}\ z = \mathtt{!}y\ \mathtt{in}\ x := z$"). `let` and `seq` may as well be considered as the result of type-oriented preprocessing rather than those directly written by programmers.

Expressions are from DCCv, incorporating dereference and incorporation of commands (the restricted position of "return" in the latter does not lose generality: the lack of `new` in expressions does not lose generality either).

The reduction rules are given in Figure 12. We use the following reduction context:

$$\begin{array}{ll}
E ::= [\,] \mid & \mathtt{succ}(E) \mid \mathtt{pred}(E) \mid Ee \mid vE \mid E\ \mathtt{return}\ e \\
\quad \mid & E; c_2 \mid \mathtt{while}\ E\ \mathtt{do}\ c \mid \mathtt{let}\ x = E\ \mathtt{in}\ c \mid \mathtt{seq}\ x = E\ \mathtt{in}\ c
\end{array}$$

The reduction takes the form $(\boldsymbol{\nu}\vec{x})(p,\sigma) \longrightarrow (\boldsymbol{\nu}\vec{x}')(p',\sigma')$ where p and p' are either expressions, commands or threads. $(\boldsymbol{\nu}\vec{x})(p,\sigma)$ is called a *configuration.* In a configuration $(\boldsymbol{\nu}\vec{x})(p,\sigma)$, σ denotes a *store*, which is a finite map from reference names to values, and we demand $x_i \in \mathtt{dom}(\sigma)$. x_i in $(\boldsymbol{\nu}\vec{x})(p,\sigma)$ indicates local references. We regard $(\boldsymbol{\nu}\,\vec{x})$ as a set and x_i is bound in $(\boldsymbol{\nu}\vec{x})(p,\sigma)$, for which we assume the standard α-equality.

The reduction on expressions is the standard single-step call-by-value reduction, using a store as necessary. Other rules come from DCCv, cf. §4.5, with appropriate state change. A dereference is treated just as in Figure 12.

Among the reduction rules for commands given in Figure 12, observe how the "new" command adds a binding to the hidden names.

(Expression)

$$(\texttt{skip return } e, \sigma) \longrightarrow (e, \sigma)$$

(Command)

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

$$(\texttt{skip}; c,\ \sigma) \longrightarrow (c,\ \sigma) \qquad \frac{(c_1, \sigma) \longrightarrow (\texttt{skip}, \sigma')}{(c_1; c_2, \sigma) \longrightarrow (c_2, \sigma')}$$

$$(\texttt{if } n \texttt{ then } c_1 \texttt{ else } c_2, \sigma) \longrightarrow (c_1, \sigma) \quad (n \neq 0)$$

$$(\texttt{if } 0 \texttt{ then } c_1 \texttt{ else } c_2, \sigma) \longrightarrow (c_2, \sigma)$$

$$(\texttt{while } n \texttt{ do } c, \sigma) \longrightarrow (c; \texttt{while } y \texttt{ do } c, \sigma) \quad (n \neq 0)$$

$$(\texttt{while } 0 \texttt{ do } c, \sigma) \longrightarrow (\texttt{skip}, \sigma)$$

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

$$(\texttt{let } x = \texttt{!}y \texttt{ in } c, \sigma) \longrightarrow (c\{v/x\},\ \sigma) \quad (\sigma(y) = v)$$

$$(\texttt{new } x \mapsto v \texttt{ in } c,\ \ \sigma) \longrightarrow (\boldsymbol{\nu} x)(c,\ \ \sigma[x \mapsto v])$$

(Thread)

$$\frac{(\boldsymbol{\nu}\vec{x}_i)(c_i,\ \sigma) \longrightarrow (\boldsymbol{\nu}\vec{x}'_i)(c'_i,\ \sigma') \quad \vec{x} = \uplus_j \vec{x}_j \quad \vec{x}' = \uplus_j \vec{x}'_j}{(\boldsymbol{\nu}\vec{x})(c_i \mid \Pi_{j \neq i} c_j,\ \sigma) \longrightarrow (\boldsymbol{\nu}\vec{x}')(c'_i \mid \Pi_{j \neq i} c_j,\ \sigma')}$$

(Common)

$$\frac{(\boldsymbol{\nu}\vec{x})(p, \sigma) \longrightarrow (\boldsymbol{\nu}\vec{x}')(p', \sigma')}{(\boldsymbol{\nu} x\vec{x})(p, \sigma) \longrightarrow (\boldsymbol{\nu} x\vec{x}')(p', \sigma')}$$

$$\frac{(\boldsymbol{\nu}\vec{x})(p, \sigma) \longrightarrow (\boldsymbol{\nu}\vec{x}')(p', \sigma')}{(\boldsymbol{\nu}\vec{x})(E[p], \sigma) \longrightarrow (\boldsymbol{\nu}\vec{x}')(E[p'], \sigma')} \quad (\vec{x} \cap \mathsf{fv}(E[\ ]) = \emptyset)$$

Fig. 12. Reduction of Extended VS-Calculus

Finally the reduction rule for threads is given in Figure 12. In the rule, $\uplus$ denotes a disjoint union. The rule allows component commands to run in an interleaved fashion, sharing a same store.

Remark 7.1 (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.

7.2 Secrecy with Reference and Procedure

We illustrate the subtlety in secrecy with local references and procedure by examples. In § 7.7, we shall see how the secrecy properties of each of the following expressions are exactly analysed through its encoding into the π-calculus. *For brevity we assume u, v, w are low-level imperative variables, x, y, z high-level ones, and a, a_i, a' those which can be either.*

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.

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

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:

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

The command writes at w and w', which are both local; however in fact it writes 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, higher-order procedures add different kinds of subtlety.

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

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

  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

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

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

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

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

Hidden Shared State. A leakage can take place through a hidden store. Let $e_6 \stackrel{\text{def}}{=} \lambda y^{\mathtt{H}}.x := y^{\mathtt{H}}$ and $e_7 \stackrel{\text{def}}{=} \lambda w.\texttt{if}\ x\ \texttt{then}\ u^{\mathtt{L}} := 0\ \texttt{else}\ u^{\mathtt{L}} := 1$ (where we omit $\texttt{return}$ when we do not care the returned value) and consider:

$$c_6 \stackrel{\text{def}}{=} \texttt{new}\ x \mapsto 0\ \texttt{in}\ ((e_6)1\ ;\ (e_7)2)$$

Here e_6 writes a high-level datum to a hidden x, while e_7 uses the value of x to write to a low variable. Note the danger of this leakage cannot be observed just by looking at the visible interface: only by measuring the level of x, we can detect such unsafe flow.

As another example of shared local state, define three expressions, $e_8 \stackrel{\text{def}}{=} \lambda y^{\mathtt{H}}.w := 0; \mathtt{return}\, y$, $e_9 \stackrel{\text{def}}{=} \lambda y^{\mathtt{H}}.w := 1; \mathtt{return}\, y$ and $e_{10} \stackrel{\text{def}}{=} \lambda z.\mathtt{return}\,!w$ and consider the following expression:

$$c_7 \stackrel{\text{def}}{=} \mathtt{new}\; w^{\mathtt{L}} \mapsto 0 \;\mathtt{in}\; (a_1 := e_8 \;;\; a_2 := e_9 \;;\; a_3 := e_{10})$$
$$c_8 \stackrel{\text{def}}{=} \mathtt{if}\; x^{\mathtt{H}} \,\mathtt{then}\; a' := !a_1 \;\mathtt{else}\; a' := !a_2$$
$$c_9 \stackrel{\text{def}}{=} v^{\mathtt{L}} := (!a_3)0$$

Now we consider whether the command "$c_7; c_8; c_9$" is secure or not. If it is, then $e_{8,9}$ should be high-level data, so that a' and $a_{1,2}$ are high-level. We observe three points.

—The functional behaviours of $e_{8,9}$ (abstracted as $\mathbb{N}_{\mathtt{H}} \Rightarrow \mathbb{N}_{\mathtt{H}}$) are at the high-level.

—The side effects of these procedures are at a hidden low-level variable w.

—The content of w, a low-level datum, can get revealed when, after $c_7; c_8$ is executed.

Thus $c_7; c_8$ is not safe because of unsafe implicit flow in the conditional in c_8: depending on the value of a high-level variable x, a' stores either e_8 or e_9, so that the effect of invoking $!a'$ has different effects on a local variable, which is revealed as different low-level data when $!a_3$ is invoked. However, if we should use $\mathbb{N}_{\mathtt{H}} \Rightarrow \mathbb{N}_{\mathtt{H}}$ as the types of $e_{8,9}$ (which may be a natural idea given w is hidden), we can never compositionally judge that c_8 can induce a dangerous flow.

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. The latter extends DCCv (§ 4.5), adding reference types and mutable arrow types. A base is a finite function from variables to total value types. Following the syntax of types of DCCv, we use S for total types and U for partial types.

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

$\mathtt{ref}_s(S)$ is a reference type of value typed by S with side effect at level s. $\mathtt{refr}_s(S)$ (resp. $\mathtt{refw}_s(S)$) is a read (resp. write) reference type with side effect at the level s. $S \stackrel{s}{\Rightarrow} T$ is a function space from a total type, with side effect at the level s. We write $S \Rightarrow T$ (a pure function space) for $S \stackrel{\mathtt{H}}{\Rightarrow} T$. The restriction of argument types to total types is as in DCCv (cf. §4.5), and does not lead to a loss of expressiveness. In $\mathrm{cmd}\, \tau_s$, $\tau = \Downarrow$ (resp. $\tau = \Uparrow$) indicates convergence (resp. potential

(Expression)

$$\frac{s \sqsubseteq s'}{\mathbb{N}_s \leq \mathbb{N}_{s'}} \qquad \frac{S' \leq S \quad T \leq T' \quad s' \sqsubseteq s}{S \overset{s}{\Rightarrow} T \leq S' \overset{s'}{\Rightarrow} T'} \qquad \frac{S \leq S' \quad s \sqsubseteq s'}{\llcorner S \lrcorner_s \leq \llcorner S' \lrcorner_{s'}}$$

$$\frac{s' \sqsubseteq s}{\mathtt{ref}_s(S) \leq \mathtt{ref}_{s'}(S)} \qquad \frac{s' \sqsubseteq s \quad S \leq S'}{\mathtt{refr}_s(S) \leq \mathtt{refr}_{s'}(S')} \qquad \frac{s' \sqsubseteq s \quad S' \leq S}{\mathtt{refw}_s(S) \leq \mathtt{refw}_{s'}(S')}$$

$$\frac{s' \sqsubseteq s \quad S \leq S'}{\mathtt{ref}_s(S) \leq \mathtt{refr}_{s'}(S')} \qquad \frac{s' \sqsubseteq s \quad S' \leq S}{\mathtt{ref}_s(S) \leq \mathtt{refw}_{s'}(S')}$$

(Environment)

$$\frac{\forall x \in \mathtt{dom}(E_1).\ E_1(x) \leq E_2(x)}{E_1 \leq E_2}$$

(Command)

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

Fig. 13. Subtyping of the Extended VS-Calculus

divergence), while s is a lower bound at which the termination may be observed (or, as Smith [65] puts it, at which level of variables a termination depends upon).

The following notion will be used for preserving totality in total function types in the presence of general state (cf. Remark 5.2).

Definition 7.2 The set of *non-reference types* are those types which are of the forms $\mathbb{N}_s$, $S \overset{s}{\Rightarrow} T$ and $\llcorner S \lrcorner_s$.

Another crucial elements in the present type discipline is the subtyping on value types and command types, generated from the rules in Figure 13. The subtyping is based on the secrecy subtyping for π^{LAR} studied in Section 6, inheriting, at the same time, from the preceding work on imperative secrecy [66; 29; 65]. In Figure 13, S in $\mathtt{refr}_s(S)$ is covariant, while S in $\mathtt{refw}_s(S)$ is contravariant. Hence in $\mathtt{ref}_s(S)$, S is invariant [55]. The subtyping on reference types, $\mathtt{ref}_s(S)$, $\mathtt{refr}_s(S)$ and $\mathtt{ref}_s(S)$, is dual to the subtyping on $\mathsf{rw}\langle\tau\rangle$, $\mathsf{r}\langle\tau\rangle$ and $\mathsf{w}\langle\tau\rangle$ in π^{LAR} as given in §6.1. The subtyping of the environment is defined pointwise. In converging command types, secrecy levels are irrelevant, while non-converging ones, they are covariant. For this reason we sometimes omit s from $\Downarrow_s$ without loss of precision.

We can easily observe that $\leq$ is a partial order. The subtyping on expression types, environments and command types will be later illustrated regarding its interplay with structural security at the end of the next subsection and in its connection to π^{LAR} in Proposition 7.13.

7.4 Tampering Level and Structural Security

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 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. To motivate their introduction, we start from stateless interactions in a DCCv-term $E \vdash M : T$. In its π-calculus translation, this becomes $\vdash_{\mathsf{sec}} \langle M \rangle_u \triangleright u : T^\bullet, E^\circ$. The tampering level of this behaviour is calculated from the action type $u : T^\bullet, E^\circ$. Since all channel type in E° has mode $?_{\mathrm{L}}$ or $?_{\mathrm{A}}$, we can completely neglect E° from the calculation, and consider only $T^\bullet$.

The situation is quite different in the present imperative setting. In both $E \vdash c : \rho$ and $E \vdash e : T$, the command and 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)
- $\mathsf{protect}(\mathbb{N}_s) = \mathsf{protect}(\llcorner S \lrcorner_s) = s$
- $\mathsf{protect}(S \overset{s}{\Rightarrow} T) = \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(T)$
- $\mathsf{protect}(\mathtt{ref}_s(S)) = \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(S)$, $\mathsf{protect}(\mathtt{refr}_s(S)) = \mathsf{protect}(S)$ and $\mathsf{protect}(\mathtt{refw}_s(S)) = \mathsf{protect}^{\mathcal{E}}(S)$

(2)
- $\mathsf{protect}^{\mathcal{E}}(\mathbb{N}_s) = \mathsf{protect}^{\mathcal{E}}(S \Rightarrow T) = \mathtt{H}$
- $\mathsf{protect}^{\mathcal{E}}(S \overset{s}{\Rightarrow} T) = \mathsf{protect}^{\mathcal{E}}(\mathtt{refw}_s(S)) = s$
- $\mathsf{protect}^{\mathcal{E}}(\mathtt{refr}_s(S)) = \mathsf{protect}^{\mathcal{E}}(S)$

The tampering level of E, denoted by $\mathsf{tamp}(E)$ (cf. Definition 3.4), is defined as:

$$\mathsf{tamp}(E) \overset{\text{def}}{=} \sqcap\{\mathsf{protect}^{\mathcal{E}}(S) \mid x : S \in E\}$$

The illustration of the above clauses is best given after we present the embedding of these types into process types, where we show that the above definition precisely corresponds to those of the tamper level in π^{LAR}. The following result confirms that the subtyping relation and the protection levels interact coherently.

Proposition 7.3 (1) $T \leq T'$ *implies* $\mathsf{protect}(T) \leq \mathsf{protect}(T')$ *and* $\mathsf{protect}^{\mathcal{E}}(T') \leq \mathsf{protect}^{\mathcal{E}}(T)$*; and* (2) $E \leq E'$ *implies* $\mathsf{tamp}(E') \leq \mathsf{tamp}(E)$.

Proof. (1) By induction of the size of T. We only prove the case $T = \mathtt{refr}_s(S)$ and $T' = \mathtt{refr}_{s'}(S')$ for $\mathsf{protect}^{\mathcal{E}}(T') \leq \mathsf{protect}^{\mathcal{E}}(T)$. Assume $\mathtt{refr}_s(S) \leq \mathtt{refr}_{s'}(S')$ with $s' \sqsubseteq s$ and $S \leq S'$. Then by the induction, $\mathsf{protect}^{\mathcal{E}}(S') \leq \mathsf{protect}^{\mathcal{E}}(S)$. Since $\mathsf{protect}^{\mathcal{E}}(S) = \mathtt{refr}_s(S)$ for any s, we have $\mathsf{protect}^{\mathcal{E}}(\mathtt{refr}_{s'}(S')) \leq \mathsf{protect}^{\mathcal{E}}(\mathtt{refr}_s(S))$, as required. (2) By induction of the size of E. W.o.l.g. we can set: $E_1 = \{x : T_1\}$ and $E_2 = \{x : T_2, y : T\}$ and assume $E_1 \leq E_2$ with $T_1 \leq T_2$. Then by (1), we know $\mathsf{protect}^{\mathcal{E}}(T_2) \leq \mathsf{protect}^{\mathcal{E}}(T_1)$. Then $\mathsf{tamp}(E_2) = \mathsf{protect}^{\mathcal{E}}(T_2) \sqcap \mathsf{protect}^{\mathcal{E}}(T) \sqsubseteq \mathsf{protect}^{\mathcal{E}}(T_1) = \mathsf{tamp}(E_1)$, as desired. □

Remark 7.4 We can also prove the above result via the translation into the π^{LAR}. See Propositions 6.2 and 7.13.

Using protection levels, we introduce a basic condition on value types, which plays a key rôle for harnessing aliases.

Definition 7.5 (structural security) The set of *structurally secure types* are generated by:

- $\mathbb{N}_s$ is structurally secure for any s.

- If S and T are structurally secure then $S \overset{s}{\Rightarrow} U$ is structurally secure.
- If S is structurally secure and $s \sqsubseteq \mathsf{protect}(\mathtt{ref}_s(S))$ then $\mathtt{ref}_s(S)$ is structurally secure. Similarly for $\mathtt{refr}_s(S)$ and $\mathtt{refw}_s(S)$.
- If S is structurally secure then $\llcorner S \lrcorner_s$ is structurally secure for any s.

The condition is directly suggested by structural security of the π^{LAR}-calculus, see Proposition 7.14 later. When combined with the subtyping relation discussed in the previous subsection, this condition can ensure secure flow with flexibility and generality (we suggested this point already in Remark 6.6). The following remark illustrates this point in some detail.

Remark 7.6 (structural security, subtyping and imperative secrecy) The structural security for imperative secrecy is based on the following two principles.

P1. The measurement of the level of a received datum is done not when that datum is read from a reference, but when that datum is actually used.

P2. If the above principle is to be maintained without violating safe information flow, then a stored datum in a reference should always have a tampering level higher than the annotating secrecy level of that reference.

In the following, we illustrate how these two principles lead to general secrecy typing for imperative computation, using two examples.

P1: Under the typing $x\,{:}\,\mathsf{ref}_{\mathtt{H}}\langle \mathbb{N}_{\mathtt{H}}\rangle$ and $u\,{:}\,\mathsf{ref}_{\mathtt{L}}\langle \mathbb{N}_{\mathtt{H}}\rangle$, consider the following assignment.

$$u \ := \ !x. \tag{19}$$

This command reads from a high-level datum from a high-level variable x, and writes the datum to a low-level variable. Is (19) safe? From **P1**, we do not regard $!x$ (reading of x) as reception of information: in fact, since if another principal wishes to use this datum by reading u, that principal should act at a high-level, since the datum is typed as $\mathbb{N}_{\mathtt{H}}$. Further such lowering is useful when, for example, it is combined with the following command, with $S \overset{\text{def}}{=} \mathsf{ref}_{\mathtt{L}}\langle \mathbb{N}_{\mathtt{H}}\rangle$. The typing follows secrecy typing rules presented later.

$$\mathsf{swap} \overset{\text{def}}{=} \lambda v_1^S.\, \lambda v_2^S.\, \mathtt{new}\ w^S \mapsto !v_1\ \mathtt{in}\ (v_1 := !v_2\ ;\ v_2 := !w; \mathtt{return}\ 1) \tag{20}$$

This procedure swaps the content of two low-level references without touching the high-level data (except generically swapping them), so can act entirely at a low-level (or as a low-level principal). Note (19) followed by (20) does not incur any insecure flow, because difference in the high-level value never influences a low-level behaviour.

P2: We now show the principle **P2** is not as restrictive as it may first look. If we are without the structural security, the only way to maintain secrecy may be, in contrast to **P1**, to regard the dereference of an imperative variable at level s is the reception of information. As far as we stick to this discipline, we can have a type $\mathsf{ref}_{\mathtt{L}}\langle \mathsf{ref}_{\mathtt{H}}\langle \mathbb{N}_{\mathtt{L}}\rangle\rangle$ which stores a low-level reference in a high-level reference. Such a reference can be meaningful since, for example, a high-level principal can look at the value stored in a shared low-level reference updated by a low-level principal.

For example, assuming x is of this type and u is of type $\mathsf{ref}_{\mathtt{L}}\langle\mathbb{N}_{\mathtt{L}}\rangle$, we may consider a high-level command such as:

$$x := u\ ;\ \mathtt{if}\ \ !!x\ \mathtt{then}\ y^{\mathtt{H}} := 1\ \mathtt{else}\ y^{\mathtt{H}} := 0 \tag{21}$$

which is surely safe. This and other similar examples however are always well-typed in the present type discipline, because of the use of subtyping. Observe, in the alternative method, the principal can only read from a low-level reference, since it has to clear a high-level reading beforehand. So let us first assign the following type to x:

$$\mathsf{ref}_{\mathtt{H}}\langle\mathsf{refr}_{\mathtt{H}}\langle\mathbb{N}_{\mathtt{H}}\rangle\rangle \tag{22}$$

But noting $\mathsf{refr}_{\mathtt{L}}\langle\mathbb{N}_{\mathtt{L}}\rangle \le \mathsf{refr}_{\mathtt{H}}\langle\mathbb{N}_{\mathtt{H}}\rangle$ (by the second rule in the second line of Figure 13) and using the subtyping between a read/write reference type and a write reference type (the second rule in the third line of Figure 13), we obtain:

$$\mathsf{ref}_{\mathtt{H}}\langle\mathsf{refr}_{\mathtt{H}}\langle\mathbb{N}_{\mathtt{H}}\rangle\rangle\ \le\ \mathsf{refw}_{\mathtt{H}}\langle\mathsf{refr}_{\mathtt{L}}\langle\mathbb{N}_{\mathtt{L}}\rangle\rangle \tag{23}$$

so that $x := u$ does make sense (and is well-typed in the type discipline later). Whenever a low-level reference is stored in a high-level reference, the latter should be read-only, so the same argument applies. This observation suggests that the approach based on structural security is strictly more general than the alternative approach. The latter would however be useful as a run-time policy, being more amenable to run-time monitoring.

7.5 Secrecy Typing

We use the following three kinds of typing judgements, each derived from the associated typing rules.

(expression)	$E \vdash e : T$	(the rules are given in Figure 14)
(command)	$E \vdash c : \mathrm{cmd}\,\tau_s$	(the rules are given in Figure 15)
(thread)	$E \vdash o : \mathrm{cmd}\,\tau_s$	(the rule is given in Figure 15)

Further Figure 15 gives weakening and subsumption rules common to expressions and commands (letting p range over their union, and α over the union of value and command types).

As we shall show later, the system is a conservative extension of the possibilistic part of Smith's system [65]. 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 read and write reference types for capturing the level of writing, which is crucial for controlling aliasing effects. In the following we illustrate typing rules one by one.

(Expressions)

- *Var, Num, Succ.* Standard. Constants such as `succ` and `pred` are typed just as in DCCv.
- *Lam.* The first condition in (†) guarantees that invoking the closure does not lead to the effects from its body that are lower than the receiving level. The second condition demands that, when the target type is total, the body never has free references. This is necessary for ensuring totality, corresponding to a constraint in $(\mathsf{In}^{!_{\mathtt{L}}})$ (a basic example which violates this condition is $(x := \lambda y^{\mathbb{N}}.(!x)y)\,;\,(!x)3$,

$$[\mathit{Var}]\ \frac{}{E, x : S \vdash x : S} \qquad [\mathit{Num}]\ E \vdash n : \mathbb{N}_s \qquad [\mathit{Succ}]\ \frac{E \vdash M : \mathbb{N}_s}{E \vdash \mathtt{succ}(M) : \mathbb{N}_s}$$

$$[\mathit{Lam}]\ \frac{E \cdot x : S \vdash e : T}{E \vdash \lambda x.e : S \stackrel{s}{\Rightarrow} T}\ (\dagger) \qquad [\mathit{App}]\ \frac{E \vdash e : S \stackrel{s}{\Rightarrow} T \quad E \vdash e' : S}{E \vdash ee' : T}$$

$$[\mathit{Lift}]\ \frac{E \vdash e : S}{E \vdash e : \llcorner S \lrcorner_s} \qquad [\mathit{Seq}]\ \frac{E_1 \vdash e' : \llcorner S \lrcorner_s \quad E_2, x : S \vdash e : U}{E \vdash \mathtt{seq}\ x = e'\ \mathtt{in}\ e : U}\ (\dagger\dagger)$$

$$[\mathit{Ret}]\ \frac{E \vdash c : \mathrm{cmd} \Downarrow_s \quad E \vdash e : T}{E \vdash c\ \mathtt{return}\ e : T} \qquad [\mathit{RetP}]\ \frac{E_1 \vdash c : \mathrm{cmd} \Uparrow_s \quad E_2 \vdash e : U}{E \vdash c\ \mathtt{return}\ e : U}\ (\dagger\dagger)$$

($\dagger$) $s \sqsubseteq \mathsf{tamp}(E)$; if T is total then $\mathtt{cod}(E)$ are non-reference types.

($\dagger\dagger$) $s \sqsubseteq \mathsf{protect}(U) \sqcap \mathsf{tamp}(E_2)$; $E_i \leq E$.

Fig. 14. Secrecy Typing Rules for the Extended VS-Calculus: expressions

which diverges). As discussed in Remark 5.2, a refined treatment of totality is possible using a version of the effect typing [9].

- *App.* *[App]* does not mention secrecy levels since we assume the arguments are applied after they are evaluated (the case when they do not is taken care of by *Seq*). As in DCCv, we can derive the following partial version of *[App]* from the rules in Figure 14.

$$[\mathit{AppP}]\ \frac{E \vdash e : U_1 \stackrel{s}{\Rightarrow} U_2 \quad E \vdash e' : U_1}{E \vdash ee' : U_2} \qquad s \sqcup \mathsf{protect}(U_1) \sqsubseteq \mathsf{tamp}(E) \sqcap \mathsf{protect}(U_2)$$

  [AppP] 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 environment, which should be the same as, or higher than, the level of termination of N, which affects the actions at the environment (note that having distinct environments allows us to type more terms since E_i's tamper level is higher than E by Proposition 7.3).
- *Ret, RetP.* The total higher-order procedure *[Ret]* does not need to consider secrecy levels. On the other hand, if the command c is partial, then its termination at level s is transmitted to e, so that information which e emits — its write effects and its termination — can only be higher than, or the same as, s of e; hence we need the side condition on secrecy.

(Command and Thread)

- *Skip.* `skip` terminates immediately, so it has the $\Downarrow$-type.
- *Assignment.* The rule crucially relies on the structural security (Definition 7.5). For example, $x := \mathtt{!}u$ with x and u typed as $\mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{L}})$ and (structurally insecure) $\mathtt{refr}_{\mathtt{H}}(\mathbb{N}_{\mathtt{L}})$, respectively, becomes typable without the structurally secure condition, which is clearly insecure. The rule records a write effect of "x" in E.

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

$$[Ass] \qquad \frac{E \vdash v : S \quad E \vdash x : \mathtt{refw}_s(S)}{E \vdash x := v \ : \ \mathrm{cmd}\Downarrow}$$

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

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

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

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

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

$$[Deref] \qquad \frac{E \vdash z : \mathtt{refr}_s(S) \quad E, x : S \vdash c : \mathrm{cmd}\, \tau_{s_0}}{E \vdash \mathtt{let}\ x = \mathtt{!}z\ \mathtt{in}\ c : \mathrm{cmd}\, \tau_{s_0}}$$

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

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

$$[Sub] \qquad \frac{E \vdash t : \alpha \quad \alpha \leq \alpha' \quad E \leq E'}{E' \vdash t : \alpha'}$$

Fig. 15. Secrecy Typing Rules for the Extended VS-Calculus: command and thread

- *Com* and *If*. *[Com]*'s side condition is equivalent to [65], which enhances [29; 66]. 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 environments in these two commands adds typability since, if we subsumption E_i so that they coincide as E, the resulting tampering level is in general lower than each $\mathsf{tamp}(E_i)$. The condition does not specify the case when the preceding command does terminate, since if so no information flow downs to the subsequent command. *[If]* is standard, requiring the conditional variable cannot influence later behaviour at lower levels.
- *While*. The side condition is due to Smith [65] who enlarges the typability of the while command on the basis of [66; 29] (the condition is somewhat simplified, but is equivalent to the one given in [65], see Proposition 7.10 (2) later). We offer intuitive illustration following [65]. 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 e is initially evaluated to be true. Then we unfold the while loop into $c;\mathtt{while}\ e\ \mathtt{do}\ c$, 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 π^{LAR}.

- *Let, Seq* In *[Let]*, e is total, so both tampering and secrecy levels are ignored. *[Seq]* in commands precisely corresponds to *[Seq]* in expressions, considering the tampering of write actions of c in addition.
- *Deref.* Note that the effect of S in z's type $\mathtt{refr}_s(S)$ is recorded if S is mutable (by definition of $\mathsf{protect}^{\mathcal{E}}(\mathtt{refr}_s(S))$), even though z is only read in this command. To see its necessity, we consider:

$$\mathtt{let}\ x = \mathtt{!}z\ \mathtt{in}\ x := 3. \tag{24}$$

  Here x looks local, but may be aliased to a free name referred by z. By calculating the level of z by $\mathsf{protect}^{\mathcal{E}}(\mathtt{refr}_s(S)) = \mathsf{protect}^{\mathcal{E}}(S)$ in the environment, we can safely capture the level of x.
- *New.* Similar to *[Deref]*, if v has a mutable type, then it is recorded in E. To understand its necessity, consider:

$$\mathtt{new}\ z \mapsto y\ \mathtt{in}\ \mathtt{let}\ \ x = \mathtt{!}z\ \mathtt{in}\ x := 3. \tag{25}$$

  Note y should have a reference type. Hence when $z \mapsto y$ is inferred, y has a write reference type in E, which subsumes the writing at x since x is higher than y by structural security.
- *Par.* The collection of threads is typable when each thread is typable. It is possible and meaningful to have different secrecy levels for different threads, though we do not explore the possibility in the present study.
- *Sub.* Standard. This rule correspond to (Sub) and (Weak) in π^{LAR}.

Remark 7.7 (refinement on write effects) The secrecy typing as given above can be refined if we associate writing effects with each expression and command (references and closures which are written), so that the level of the environment needs only be considered for those names (these names are cleared when we abstract an expression, just as the standard effect discipline [9]). The refinement is especially useful when we have commands which contain abstractions. The refinement is technically innocuous (though useful) and maintains the same noninterference results which we state later, with the identical arguments.

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 7.8 (typing examples in the extended Smith-Volpano calculus) Commands and expressions are from § 7.2 (some at the end are from Remark 7.6). We assume x, y, z are high while u, v, w are low unless otherwise stated.

(1) Let $E = u : \mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}}), x : \mathtt{ref}_{\mathtt{H}}(\mathbb{N}_{\mathtt{H}}), y : \mathtt{refr}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}})$. By *[Ass]* and *[Deref]*:

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

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

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

We also have $E \vdash 0 : \mathbb{N}_{\mathtt{H}}$. Finally by *[New]*,

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

for arbitrary s (we omit such s from now on). The tampering level of this command is $\mathsf{tamp}(E/u) = H$.

(2) Let $E = w : \mathtt{refr}_{\mathtt{L}}(\mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}}))$. Then we have

$$E, w_0 : \mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}}) \vdash w_0 := 3 \ : \ \mathrm{cmd} \Downarrow$$

By *[Deref]*, we obtain:

$$E \vdash \mathtt{let}\ w_0 = \mathtt{!}w\ \mathtt{in}\ w_0 := 3 \ : \ \mathrm{cmd} \Downarrow \tag{26}$$

Note the tampering level of the above command is $\mathtt{L}$ since $\mathsf{protect}^{\mathcal{E}}(\mathtt{refr}_{\mathtt{L}}(\mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}}))) = \mathsf{protect}^{\mathcal{E}}(\mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}})) = \mathtt{L}$.

(3) Let $E' = w : \mathtt{ref}_{\mathtt{L}}(\mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}})), u : \mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}}), v : \mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}})$. First we have $E' \vdash w : \mathtt{refw}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}})$ by *[Var]* and *[Sub]*. Then by *[Ass]* we obtain:

$$E' \vdash w := u \ : \ \mathrm{cmd} \Downarrow \tag{27}$$

We now apply *[Sub]* and *[Com]* to (26) and (27). Also by *[Var]*, we have $E' \vdash v : \mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}})$. Finally we obtain, by applying *[New]*:

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

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

(4) Recall $c_3 \stackrel{\mathrm{def}}{=} u{:=}1; (\mathtt{if}\ z\ \mathtt{then}\ x{:=}e_1\ \mathtt{else}\ x{:=}e_2); z'{:=}(\mathtt{!}x)0; u{:=}0$, with

$$e_1 \stackrel{\mathrm{def}}{=} \lambda y.(\mathtt{!}x)y \ (\stackrel{\mathrm{def}}{=}\ \mathtt{let}\ w = \mathtt{!}x\ \mathtt{in}\ \mathtt{let}\ k = w0\ \mathtt{in}\ z' := k)$$

and $e_2 \stackrel{\mathrm{def}}{=} \lambda y.y$. To analyse this command, we note that e_1 contains a mutable variable x, so the use of *[Lam]* demands the target is partial. Secondly, to type $\mathtt{if}$-command, since z is a high-variable, x should have type $\mathtt{ref}_{\mathtt{H}}(T)$. By the structural security, this means T has type $T = \llcorner S \lrcorner_{\mathtt{H}}$. Hence by the side condition of *[Seq]*, $\mathtt{seq}\ k = w0\ \mathtt{in}\ z' := k$ has type cmd $\Uparrow_{\mathtt{H}}$, so that $\mathtt{if}$-command and $z' := (\mathtt{!}x)0$ should have type cmd $\Uparrow_{\mathtt{H}}$ by the side condition of *[Seq]*. To compose $z'{:=}(\mathtt{!}x)0$ and $u := 0$, we need to satisfy the side condition of *[Seq]* again, which means $u := 0$ has type cmd $\Uparrow_{\mathtt{H}}$, too. However it is impossible because by *[Ass]*, we should have $E_2 \vdash u : \mathtt{refw}_{\mathtt{L}}(\mathbb{N}_s)$ for some E_2 which implies $\mathsf{protect}^{\mathcal{E}}(E_2(u)) = \mathtt{L}$, violating the side condition *[Seq]* such that $\mathtt{H} \sqsubseteq \mathsf{tamp}(E_2)$.

(5) Let $e_3 \stackrel{\mathrm{def}}{=} \lambda x.\ u := x\ \mathtt{return}\ 0$ and $E = z : \mathbb{N}_{\mathtt{H}}, u : \mathtt{ref}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}})$. We can check e_3 is typable as $E \vdash e_3 \ : \ \llcorner \mathbb{N}_{\mathtt{L}} \stackrel{\mathtt{H}}{\Rightarrow} \mathbb{N}_{\mathtt{H}} \lrcorner_{\mathtt{H}}$. We now analyse the command $c_4 \stackrel{\mathrm{def}}{=} \mathtt{if}\ z\ \mathtt{then}\ \mathtt{let}\ y = (e_3)0\ \mathtt{in}\ \mathtt{skip}$. We observe that the tampering level of the internal $\mathtt{let}$-command is $\mathsf{tamp}(E) = \mathtt{L}$. 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 $c'_4 \stackrel{\text{def}}{=} \mathtt{if}\ z\ \mathtt{then}\ \mathtt{let}\ y = (e_4)0\ \mathtt{in}\ \mathtt{skip}$, with $e_4 \stackrel{\text{def}}{=} \lambda x.\ \mathtt{skip}\ \mathtt{return}\ !u$, and $E' = z : \mathbb{N}_{\mathtt{H}}, u : \mathtt{refr}_{\mathtt{L}}(\mathbb{N}_{\mathtt{H}})$. Then we have:

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

(note that u's tampering level is $\mathtt{H}$ in E' since $\mathbb{N}_{\mathtt{H}}$ immutable). Hence c'_4 is typable because $\mathtt{H} \sqsubseteq \mathsf{tamp}(E') = \mathtt{H}$.

(6) Similarly we can check $c_5 \stackrel{\text{def}}{=} \mathtt{if}\, z\, \mathtt{then}\, \mathtt{new}\, v \mapsto w\, \mathtt{in}\, \mathtt{let}\, x = (e_5)v\, \mathtt{in}\, \mathtt{skip}$ with $e_5 \stackrel{\text{def}}{=} \lambda u.\ !!u := 1\ \mathtt{return}\ 0$ in § 7.2 is untypable. This is because the write mode of w is recorded by *[New]* and its tampering level is $\mathtt{L}$, which violates the condition for $\mathtt{if}$-command. However if we change c_5 into:

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

Then the command is typable since the body of the $\mathtt{if}$-command is a high-level (by the lack of free variable in the $\mathtt{if}$-branch).

(7) Given $e_6 \stackrel{\text{def}}{=} \lambda y^{\mathtt{H}}.\, x := y$ and $e_7 \stackrel{\text{def}}{=} \lambda w^{\mathtt{L}}.\mathtt{if}\, x\, \mathtt{then}\, u := 0\, \mathtt{else}\, u := 0$, the command $c_6 \stackrel{\text{def}}{=} \mathtt{new}\ x \mapsto 0\ \mathtt{in}\ ((e_6)1\ ;\ (e_7)2)$ is untypable because, when we type e_6, x should be high, while when we type e_7, it should be low.

(8) Recall $e_8 \stackrel{\text{def}}{=} \lambda y^{\mathtt{H}}.w := 0; \mathtt{return}\, y$, $e_9 \stackrel{\text{def}}{=} \lambda y^{\mathtt{H}}.w := 1; \mathtt{return}\, y$ and $e_{10} \stackrel{\text{def}}{=} \lambda z.\mathtt{return}\, !w$, as well as $c_7 \stackrel{\text{def}}{=} \mathtt{new}\ w \mapsto 0\ \mathtt{in}\ (a_1 := e_8\, ;\, a_2 := e_9\, ;\, a_3 := e_{10})$ and $c_8 \stackrel{\text{def}}{=} \mathtt{if}\, x^{\mathtt{H}}\, \mathtt{then}\, a' := !a_1\, \mathtt{else}\, a' := !a_2$, $c_9 \stackrel{\text{def}}{=} v^{\mathtt{L}} := (!a_3)0$, the composition $c_7; c_8; c_9$ is untypable because the type of e_8 is $\mathbb{N}_{\mathtt{H}} \stackrel{\mathtt{L}}{\Rightarrow} \mathbb{N}_{\mathtt{H}}$ whose tamper level is low.

(9) Let $E \stackrel{\text{def}}{=} x : \mathsf{ref}_{\mathtt{H}}\langle \mathbb{N}_{\mathtt{H}} \rangle,\ u : \mathsf{ref}_{\mathtt{L}}\langle \mathbb{N}_{\mathtt{H}} \rangle$, Then the following assignment (cf. Remark 7.6 (1)) is secure.

$$E \vdash_{\mathsf{sec}} u\ :=\ !x \triangleright \mathrm{cmd} \Downarrow_{\mathtt{L}}$$

Note $!x$ is *not* regarded as disclosure of a high-level datum, even if x is high.

(10) Let $E \stackrel{\text{def}}{=} x : \mathsf{ref}_{\mathtt{H}}\langle \mathsf{refr}_{\mathtt{H}}\langle \mathbb{N}_{\mathtt{H}} \rangle\rangle, u : \mathsf{ref}_{\mathbb{N}_{\mathtt{L}}}\langle \mathtt{L} \rangle$. Then the following command is securely typable (cf. Remark 7.6 (2)).

$$x := u\ ;\ \mathtt{if}\ \ !!x\ \mathtt{then}\ y^{\mathtt{H}} := 1\ \mathtt{else}\ y^{\mathtt{H}} := 0$$

The conditional is safe; $x := u$ is typable since, as we illustrated in Remark 7.6 (2), we have $\mathsf{ref}_{\mathtt{H}}\langle \mathsf{refr}_{\mathtt{H}}\langle \mathbb{N}_{\mathtt{H}} \rangle\rangle \sqsubseteq \mathsf{ref}_{\mathtt{H}}\langle \mathsf{refr}_{\mathtt{L}}\langle \mathbb{N}_{\mathtt{L}} \rangle\rangle$, so that $E \vdash x : \mathsf{ref}_{\mathtt{H}}\langle \mathsf{refr}_{\mathtt{L}}\langle \mathbb{N}_{\mathtt{L}} \rangle\rangle$.

Basic syntactic properties of the secrecy typing follow, after a definition. In Proposition 7.10 (2) below, the *possibilistic Smith-calculus* is the calculus by Smith [65] which neglects (Protect) (i.e. we ignore the part of the system in [65] which "counts execution steps").

Definition 7.9 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 15 and (ii) the rules in Figure 14 except for *[Var]*, *[Num]* and *[Succ]*.

We also write $E \vdash \sigma$ when σ is well-typed with respect to E in the obvious sense.

Proposition 7.10

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

(*2*) (conservativity) *Assume* $\mathcal{L} \stackrel{def}{=} \{\mathtt{L}, \mathtt{H}\}$ *and* c *is first-order under* E. *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* $E' \leq E$ *and* $s \sqsubseteq \mathsf{tamp}(E')$.

Proof. For (1) we use the both-way correspondence in typability between the explicit versions of secure π^{LAR} 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 [65] as $E' \vdash c : \mathtt{cmd} \Uparrow_{s'}$, with E' has read-write subtyping 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 [65] and $\mathsf{tamp}(E)$, as well as the side conditions in `while`-rules. See Appendix C. □

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.

$$\begin{array}{llll}
\text{(value)} & S^{\bullet} \stackrel{\text{def}}{=} (S^{\circ})^{\uparrow_{\mathrm{L}}} & & U^{\bullet} \stackrel{\text{def}}{=} (U^{\circ})^{\uparrow_{\mathrm{A}}}_{s} \quad (\mathsf{protect}(U) = s) \\
 & \mathbb{N}^{\circ}_{s} = ([\oplus_{i\in\mathbb{N}}\ ]^{\uparrow_{\mathrm{L}}}_{s})^{!_{\mathrm{L}}} & & (S \stackrel{s}{\Rightarrow} U)^{\circ} \stackrel{\text{def}}{=} (\overline{S^{\circ}} U^{\bullet})^{!_{\mathrm{A}}}_{s} \\
 & \llcorner S \lrcorner^{\circ}_{s} \stackrel{\text{def}}{=} S^{\circ} & & \mathtt{ref}_s(S)^{\circ} \stackrel{\text{def}}{=} \mathsf{ref}_s\langle S^{\circ}\rangle \\
 & \mathtt{refr}_s(S)^{\circ} \stackrel{\text{def}}{=} \overline{\mathsf{r}_s\langle \overline{S^{\circ}}\rangle} & & \mathtt{refw}_s(S)^{\circ} \stackrel{\text{def}}{=} \overline{\mathsf{w}_s\langle \overline{S^{\circ}}\rangle} \\
\text{(base)} & (\emptyset)^{\circ} = \emptyset \qquad (E, x : S)^{\circ} = E^{\circ}, x : \overline{S^{\circ}} & & \\
\text{(action)} & \Downarrow^{\bullet}_{s} \stackrel{\text{def}}{=} ()^{\uparrow_{\mathrm{L}}} \qquad \Uparrow^{\bullet}_{s} \stackrel{\text{def}}{=} ()^{\uparrow_{\mathrm{A}}}_{s} & &
\end{array}$$

Remark 7.11 (subtyping in the VS-calculus through process encoding) By examining the embedding, 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 subtyping of reference types is elucidated by observing the content type now occurs covariantly in the read and contravariantly in the write, hence invariantly in the reference [5]. In fact the subtyping on value types in § 7.3 precisely corresponds to the secrecy subtyping in the π^{LAR}-types.

Proposition 7.12 (subtyping) $T_1 \leq T_2$ *iff* $T_1^{\circ} \leq T_2^{\circ}$ *iff* $T_1^{\bullet} \leq T_2^{\bullet}$.

Proof. See Appendix D.1. □

The protection levels in the VS-calculus and the tampering levels in the π^{LAR}-calculus coincide via encoding. For the proof see Appendix D.2.

(Expression)

$\langle c \;\texttt{return}\; e\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)([\![c]\!]_m \mid m.\langle e\rangle_u)$

Others are the same as DCCv defined in Figure 9.

(Command)

$$[\![\texttt{skip}]\!]_u \stackrel{\text{def}}{=} \overline{u}$$

$$[\![x := v]\!]_u \stackrel{\text{def}}{=} \begin{cases} (\boldsymbol{\nu}\, y)(\overline{x}\texttt{inr}\langle yu\rangle \mid P) & (\langle v\rangle_m \stackrel{\text{def}}{=} \overline{m}(y)P) \\ \overline{x}\texttt{inr}\langle yu\rangle & (\langle v\rangle_m \stackrel{\text{def}}{=} \overline{m}\langle y\rangle) \end{cases}$$

$$[\![c_1; c_2]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, e)([\![c_1]\!]_e \mid e.[\![c_2]\!]_u)$$

$$[\![\texttt{if}\, v\, \texttt{then}\, c_1\, \texttt{else}\, c_2]\!]_u \stackrel{\text{def}}{=} \begin{cases} (\boldsymbol{\nu}\, y)(P \mid \texttt{if}\; y\; \texttt{then}\; [\![c_1]\!]_u\; \texttt{else}\; [\![c_2]\!]_u) & (\langle v\rangle_m = \overline{m}(y)P) \\ \texttt{if}\; y\; \texttt{then}\; [\![c_1]\!]_u\; \texttt{else}\; [\![c_2]\!]_u & (\langle v\rangle_m = \overline{m}\langle y\rangle) \end{cases}$$

$$\begin{aligned}[\![\texttt{while}\; e\; \texttt{do}\; c]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, fgh)(\,&\overline{g}\langle u\rangle \mid\, !f(k).\overline{g}\langle k\rangle \mid\, !h(y).\langle e\rangle_y \mid \\ &!g(k).\overline{h}(y)y(z).\texttt{if}\; z\; \texttt{then}\; (\boldsymbol{\nu}\, l)([\![c]\!]_l \mid l.\overline{f}\langle k\rangle)\; \texttt{else}\; \overline{k}\,)\end{aligned}$$

$$[\![\texttt{let}\; x = e\; \texttt{in}\; c]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle e\rangle_m \mid m(x).[\![c]\!]_u)$$

$$[\![\texttt{new}\; x \mapsto v\; \texttt{in}\; c]\!]_u \stackrel{\text{def}}{=} \begin{cases} (\boldsymbol{\nu}\, xy)(\mathsf{Ref}\langle xy\rangle \mid P \mid [\![c]\!]_u) & (\langle v\rangle_m \stackrel{\text{def}}{=} \overline{m}(y)P) \\ (\boldsymbol{\nu}\, x)(\mathsf{Ref}\langle xy\rangle \mid [\![c]\!]_u) & (\langle v\rangle_m \stackrel{\text{def}}{=} \overline{m}\langle y\rangle) \end{cases}$$

$$[\![\texttt{let}\; x = \texttt{!}y\; \texttt{in}\; c]\!]_u \stackrel{\text{def}}{=} \overline{y}\texttt{inl}(e)e(x).[\![c]\!]_u$$

$$[\![\Pi_i c_i]\!]_{\vec{u}} \stackrel{\text{def}}{=} \Pi_i [\![c_i]\!]_{u_i}$$

$$\texttt{if}\; x\; \texttt{then}\; P\; \texttt{else}\; Q \stackrel{\text{def}}{=} \overline{x}(z)z[\&_i\, .R_i] \qquad (R_i = P,\text{ if } i \geq 1 \quad R_0 = Q)$$

Fig. 16. Encoding of the Extended VS-Calculus

Proposition 7.13 (protection levels and tampering levels)

(*1*) $\mathsf{tamp}(S^\bullet) = \mathsf{tamp}(S^\circ) = \mathsf{tamp}(\llcorner S \lrcorner_s^\circ)$.
(*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 with the structural security on mutable types of the translation.

Proposition 7.14 (structural security) *T is structurally secure iff T° is structurally secure iff $T^\bullet$ is structurally secure.*

Proof. We only prove if T is structurally secure then T° is structurally secure. We prove by the size of T. The only interesting case is either $T = \texttt{ref}_s(S)$, $\texttt{refr}_s(S)$ or $\texttt{refw}_s(S)$. Suppose $\texttt{ref}_s(S)$ is structurally secure. Then by definition and Proposition 7.13 (1), we have $\mathsf{sec}(\texttt{ref}_s(S)^\circ) = s \sqsubseteq \mathsf{protect}(\texttt{ref}_s(S)) = \mathsf{protect}(S) \sqcap \mathsf{protect}^{\mathcal{E}}(S) = \mathsf{tamp}(S^\circ) \sqcap \mathsf{tamp}(\overline{S^\circ}) = \mathsf{tamp}(\texttt{ref}_s(S)^\circ)$, as required. The other cases are just similar. □

The encoding of commands and expressions is given in Figure 16. Expressions use call-by-value encoding [46; 31]. `while` is translated using tail recursion. We shall later establish the encoding of each typable expression/command/thread is indeed securely typable under the encoding of the corresponding type.

7.7 Analysis of Imperative Secrecy via Embedding

The embedding of commands and expressions offers an in-depth analysis of imperative secrecy via the fine-grained representation in name passing processes, both in types and operations. In the following we present such an analysis using concrete examples from § 7.2 and § 7.5. We also present an analysis and a derivation of the secrecy conditions for while loop due to Smith [65].

(1) (`if`) $\mathtt{if}\, v\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2$ 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 $\mathbb{N}_s$) should be lower than the tampering level of $[\![c_1]\!]_e$ and $[\![c_2]\!]_e$ by the side condition of $(\mathsf{Bra}^{\downarrow_{\mathrm{L}}})$. This corresponds to the side condition of *[If]* of the VS-calculus given in Figure 15.

(2) (`deref`) First we show the encoding of command $\mathtt{let}\ w_0 = \mathtt{!}w\ \mathtt{in}\ w_0 := 3$ from (26) in Example 7.8, § 7.5 (cf. Example 6.7 (1)).

$$\vdash_{\mathsf{sec}} \overline{w}\,\mathtt{read}(e)e(w_0).\overline{w_0}\,\mathtt{write}\langle 3f\rangle \triangleright w : \mathsf{r}_{\mathtt{L}}\langle \mathsf{rw}_{\mathtt{L}}\langle \overline{\mathbb{N}_{\mathtt{H}}}\rangle\rangle, f : ()^{\uparrow_{\mathrm{L}}}$$

Since f (the linear unary output) has no tampering level, the tampering level of the above command is that of w, i.e. $\mathsf{tamp}(\mathsf{r}_{\mathtt{L}}\langle \mathsf{rw}_{\mathtt{L}}\langle \mathbb{N}_{\mathtt{H}}\rangle\rangle) = \mathsf{tamp}(\mathsf{rw}_{\mathtt{L}}\langle \mathbb{N}_{\mathtt{H}}\rangle) = \mathtt{L}$.

(3) (`new`) A process representation of $\mathtt{new}\ w \mapsto v\ \ \mathtt{in}\ \mathtt{let}\ w_0 = \mathtt{!}w\ \mathtt{in}\ w_0 := 3$ appeared in (25) in § 7.5, is typed as follows.

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

In the above encoding, we observe that w_0 is assigned v; hence the tampering level of the above command is that of v, i.e. $\mathtt{L}$. Note also if we set the type of v to be $\mathsf{rw}_{\mathtt{H}}\langle \overline{\mathbb{N}_{\mathtt{L}}}\rangle$, then this command is untypable (and in fact unsafe), due to a violation of structural security. Similarly we can analyse the correspondence between tampering level of c_1 and c_2 in § 7.2 and that of their encodings.

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

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

First because of `if`-statement, x should have the high reference type. Hence w and k should have $\mathtt{H}$ by structural security. Also the reply at c may not terminate, hence for $c(k).\overline{z}\,\mathtt{write}\langle kc'\rangle \mid c'.\overline{f}$ to be typable, by $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$, z should be $\mathtt{H}$ and $\overline{f}$ has a type $()^{\uparrow_{\mathrm{A}}}_{\mathtt{H}}$. Now, the sequential composition is given as:

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

To type $f.[\![u := 0]\!]_{f'}$, by $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$, it is necessary for $\mathtt{H}$ to be lower than the tampering level of $[\![u := 0]\!]_{f'}$, but we have $\mathsf{tamp}(u : \mathsf{rw}_{\mathtt{L}}\langle \mathbb{N}_s\rangle) = \mathtt{L}$, 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 [65]. We show this condition is precisely what can be derived from the typability of the encoding. We consider a simplified case $\mathtt{while}\ !x\ \mathtt{do}\ c$ with x being a boolean, which easily extends to the general case.

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

Note e and f have mode $!_{\mathrm{A}}$ 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.

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

From this we can immediately derive the following conditions:
(a) By $(\mathsf{In}^{!_{\mathrm{A}}})$ at the e-replications, we require $s' \sqsubseteq \mathsf{tamp}(E)$.
(b) By $(\mathsf{Bra}^{\downarrow_{\mathrm{A}}})$ at the z-input, we require $s \sqsubseteq \mathsf{tamp}(E)$ and $s \sqsubseteq s'_0$.
(c) By $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$ 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 [65].

7.8 Noninterference via Embedding

We can now prove the noninterference for the extended VS-calculus following essentially the same technical development as in Section 4. We first introduce the contextual congruence for the extended VS-calculus, after some preliminaries.

- $E \vdash \sigma$ if a store σ conforms to E in typing in the obvious sense and, moreover, $\mathtt{dom}(\sigma)$ covers all reference variables in E.
- $E \vdash o : \mathrm{cmd}\,\tau_s$ is *semi-closed* if $\mathtt{cod}(E)$ contains only reference types. A typed context $C[\ ]^{E,\rho}$ is *semi-closing* if the resulting thread is semi-closed ($C[\ ]^{E,\rho}$ denotes a typed context whose resulting thread has type ρ and environment E).
- $(o,\sigma)\Downarrow$ denotes $(o,\sigma) \twoheadrightarrow (\boldsymbol{\nu}\,\vec{x})(o',\sigma') \not\longrightarrow$.

Definition 7.15 We write $E \vdash e_1 \cong^{\mathrm{vs}}_s e_2 : T$ when, for each semi-closing context $C[\ ]^{E,\rho}$ such that $\rho = \mathrm{cmd}\,\tau_{s_0}$, $s_0 \sqsubseteq s$, and for each σ such that $E \vdash \sigma$, we have $(C[e_1],\sigma)\Downarrow$ iff $(C[e_2],\sigma)\Downarrow$. Similarly we define $E \vdash c_1 \cong^{\mathrm{vs}}_s c_2 : \rho$ and $E \vdash o_1 \cong^{\mathrm{vs}}_s o_2 : \rho$. We often write e.g. $e_1 \cong^{\mathrm{vs}}_s e_2$, omitting the type information.

Below in (2), let $[\![\sigma]\!] \stackrel{\mathrm{def}}{=} \Pi_i \mathsf{Ref}\langle x_i v_i\rangle^\circ$ where we set:

$$\mathsf{Ref}\langle x_i v_i\rangle^\circ \stackrel{\mathrm{def}}{=} \begin{cases} (\boldsymbol{\nu}\, c)(\mathsf{Ref}\langle x_i c\rangle | P) & (\text{if } \langle v_i\rangle_u \equiv \overline{u}(c)P) \\ \mathsf{Ref}\langle x_i y_i\rangle & (\text{if } \langle v_i\rangle_u \equiv \overline{u}\langle y_i\rangle) \end{cases}$$

Proposition 7.16

(*1*) (typability) (a) *If* $E \vdash e : T$ *then* $\vdash_{\mathsf{sec}} [\![e]\!]_u \triangleright u : T^{\bullet}, E^{\circ}$. (b) *If* $E \vdash c : \rho$ *then* $\vdash_{\mathsf{sec}} [\![c]\!]_u \triangleright u : \rho^{\bullet}, E^{\circ}$. (c) *If* $E \vdash o : \rho$ *then* $\vdash_{\mathsf{sec}} [\![o]\!]_{\vec{u}} \triangleright u_1 : \rho^{\bullet}, .., u_n : \rho^{\bullet}, E^{\circ}$.

(*2*) (computational adequacy) *Assume* $E \vdash o : cmd\,\tau_s$ *is semi-closed and* $E \vdash \sigma$. *Then* $(o, \sigma) \Downarrow$ *iff* $([\![o]\!]_{\vec{u}} \,|\, [\![\sigma]\!]) \longrightarrow^{*} \Pi_i \overline{u_i} \,|\, R \not\longrightarrow$.

(*3*) (soundness) (a) $[\![e_1]\!]_u \cong_s [\![e_2]\!]_u$ *implies* $e_1 \cong_s^{\text{VS}} e_2$. (b) $[\![c_1]\!]_u \cong_s [\![c_2]\!]_u$ *implies* $c_1 \cong_s^{\text{VS}} c_2$. (c) $[\![o_1]\!]_u \cong_s [\![o_2]\!]_u$ *implies* $o_1 \cong_s^{\text{VS}} o_2$.

PROOF. (1) is straightforward induction, using Propositions 7.12, 7.13 and 7.14. The while command is treated in §7.7. For other cases, see Appendix D.3 (1). (2) uses a both-way correspondence in reduction modulo a strongly syntactic equivalence, see Appendix D.3 (2). (3) is standard from (1) and (2) above, noting all contexts in the VS-calculus are realisable by a π-term preserving typability. □

Definition 7.17 Let $E \vdash \sigma_{1,2}$ and define $E \vdash \sigma_1 \sim_s \sigma_2$ as $\sigma_1(x) = \sigma_2(x)$ for each $x \in \mathtt{dom}(E)$ such that $\mathsf{protect}(E(x)) \sqsubseteq s$.

Theorem 7.18 (non-interference) *If* $E \vdash o : cmd\,\tau_s$ *and* $E \vdash \sigma_1 \sim_s \sigma_2$ *then* $(o, \sigma_1) \Downarrow$ *iff* $(o, \sigma_2) \Downarrow$.

PROOF. Let $\vdash e_i : T$ $(i = 1, 2)$ with $\mathsf{protect}(T) \not\sqsubseteq s$. Suppose T is not a reference type. By Proposition 7.13 (1) and Proposition 7.16 (1). we know $\mathsf{protect}(T) \sqsubseteq \mathsf{tamp}(T^{\bullet})$. By Proposition 6.10, we know $[\![x \mapsto v_1]\!] \cong_s [\![x \mapsto v_2]\!]$ under $x : \mathsf{ref}\langle T^{\circ}\rangle$. If v_i is a reference type, $[\![x \mapsto v_1]\!] \cong_s [\![x \mapsto v_2]\!]$ immediately holds. Thus we know:

Claim A. $\sigma_1 \sim_s \sigma_2$ implies $[\![\sigma_1]\!] \cong_s [\![\sigma_2]\!]$.

We also need a process context which translates n affine outputs to a single affine output. Leaving the construction to Appendix D.3 (5), we observe:

Claim B. Given $\vdash P \triangleright A$ such that $A \stackrel{\text{def}}{=} u_1 : ()^{\uparrow_{\mathrm{A}}}, .., u_n : ()^{\uparrow_{\mathrm{A}}}, !_{\mathrm{L}}B$, there exists a context $C[\,\cdot\,]_A^{x:()^{\uparrow_{\mathrm{A}}}}$ such that $P \twoheadrightarrow (\Pi u_i)|R$ iff $C[P] \Downarrow_x$.

Now assume $E \vdash o : \mathrm{cmd}\,\tau_s$, $E \vdash \sigma_{1,2}$ and $E \vdash \sigma_1 \sim_s \sigma_2$. We now reason as follows. Below we let $\mathtt{dom}(\sigma) = \mathtt{dom}(E) = \{\vec{y}\}$.

$$
\begin{array}{llll}
\sigma_1 \sim_s \sigma_2 & \Rightarrow & [\![\sigma_1]\!] \cong_s [\![\sigma_2]\!] & \text{(Claim A)} \\
& \Rightarrow & [\![c]\!]_{\vec{u}} \,|\, [\![\sigma_1]\!] \cong_s [\![c]\!]_{\vec{u}} \,|\, [\![\sigma_2]\!] & \text{(congruency)} \\
& \Rightarrow & \exists C[\,\cdot\,].\ (C[[\![c]\!]_{\vec{u}} \,|\, [\![\sigma_i]\!]] \Downarrow_x \Leftrightarrow\ [\![c]\!]_{\vec{u}} \,|\, [\![\sigma_i]\!] \twoheadrightarrow (\Pi\overline{u_j})|R_i) & \text{(Claim B)} \\
& \Rightarrow & [\![c]\!]_{\vec{u}} \,|\, [\![\sigma_1]\!] \twoheadrightarrow (\Pi\overline{u_j})|R_1 \ \Leftrightarrow\ [\![c]\!]_{\vec{u}} \,|\, [\![\sigma_2]\!] \twoheadrightarrow (\Pi\overline{u_j})|R_2 & \text{(Def. 6.9)} \\
& \Rightarrow & (c,\ \sigma_1) \Downarrow \ \Leftrightarrow\ (c,\ \sigma_2) \Downarrow & \text{(Pro. 7.16 (2))}
\end{array}
$$

hence done. □

Theorem 7.18 only mentions convergence, without discussing how the resulting states of stores are to be related. In the following we show two results which do talk about the content of stores.

We first generalise the relation $\sim_s$. A *generalised store* is a pair of a finite set of (typed) names and a store, written $(\boldsymbol{\nu}\,\vec{x})\sigma$, such that $x_i \in \mathtt{dom}(\sigma)$. We write $E \vdash (\boldsymbol{\nu}\,\vec{x})\sigma$ iff $E \cdot E' \vdash \sigma$ for some E' such that $\mathtt{dom}(E) = \vec{x}$. The encoding is defined as $[\![(\boldsymbol{\nu}\,\vec{x})\sigma]\!] \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,\vec{x})[\![\sigma]\!]$. Then $\cong^{\text{vs}}_s$ over generalised stores is given as: $E \vdash (\boldsymbol{\nu}\,\vec{x}_1)\sigma_1 \cong^{\text{vs}}_s (\boldsymbol{\nu}\,\vec{x}_2)\sigma_2$ iff, for any o such that $E \vdash o : \text{cmd}\, s_\tau$, we have:

$$(\boldsymbol{\nu}\,\vec{x}_1)(o, \sigma_1) \Downarrow \quad \Leftrightarrow \quad (\boldsymbol{\nu}\,\vec{x}_2)(o, \sigma_2) \Downarrow$$

where we write $(o, \sigma) \Downarrow (\boldsymbol{\nu}\,\vec{x})\sigma'$ for $(o, \sigma) \Downarrow (\boldsymbol{\nu}\,\vec{x})(\Pi \mathtt{skip}, \sigma')$.

The following result generalises Theorem 7.18 by incorporating the first-order store. Below, assuming $E \vdash o$ is semi-closed and $E(x_i) = \mathbb{N}_{s_i}$ for $1 \leq i \leq n$, we write $(o, \sigma) \Downarrow_{x_1:n_1,..,x_m:n_m}$ when $(o, \sigma) \Downarrow (\boldsymbol{\nu}\,\vec{x})\sigma'$ such that $\sigma'(x_i) = n_i$ for each i.

Theorem 7.19 (non-interference for first-order store) *If $E \vdash o : cmd\, \tau_s$, $E \vdash \sigma_1 \cong^{\text{vs}}_s \sigma_2$ and $E(x_i) = \mathbb{N}_{s_i}$ for $x_i \in \{\vec{x}\}$ such that $s' \sqsubseteq s$, then $(o, \sigma_1) \Downarrow_{\vec{x}:\vec{n}}$ iff $(o, \sigma_2) \Downarrow_{\vec{x}:\vec{n}}$.*

Proof. We have essentially the same reasoning as Theorem 7.18, using the strengthend versions of key lemmas, including computational adequacy and what correspond to Claim A/B. See Appendix D.6. □

For generalisation of the non-interference further to higher-order stores, we restrict the language to the sequential one. Below *the sequential VS-calculus* is the imperative language we have been treating so far except we only allow a single thread in the configuration (hence a thread and a command are one and the same thing: except this, the typing etc. remains identical). $\cong^{\text{vs}}_s$ on commands and stores is defined using the restricted set of contexts.

Theorem 7.20 (sequential non-interference for generalised store) *In the sequential VS-calculus, if $E \vdash c : cmd\, \tau_s$ and $E \vdash \sigma_1 \cong^{\text{vs}}_s \sigma_2$ then $(c, \sigma_1) \Downarrow (\boldsymbol{\nu}\,\vec{x}_1)\sigma'_1$ implies $(c, \sigma_2) \Downarrow (\boldsymbol{\nu}\,\vec{x}_2)\sigma'_2$ such that $(\boldsymbol{\nu}\,\vec{x}_1)\sigma'_1 \cong^{\text{vs}}_s (\boldsymbol{\nu}\,\vec{x}_2)\sigma'_2$.*

Proof. Again we have essentially the same reasoning as Theorem 7.18, though with different versions of the key lemmas. See Appendix D.6. □

We observe that, by apprioriately varying the affine termination level, the stated noninterference property encompasses both the strong and weak noninterference. For more discussions, see the next section.

In the concurrent language, we believe that the analogue of Theorem 7.20 holds with a stronger non-interference property in which intermediate states are also s-equated (cf. Remark 6.11, see also [14]). Such a result may be proved through the noninterference in π^{LAR} based on reduction-closed congruence as well as using a close operational correspondence between the original program phrases and their embedding. See Section 8 for more discussion.

8. DISCUSSIONS

8.1 Further Study on Imperative Secrecy (1)

In Section 7, we studied a new secrecy typing for imperative, higher-order and concurrent programming, extending Volpano-Smith multi-threaded imperative language with higher-order procedures and general references. In this subsection and

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

$$[Succ]\ \ \frac{E \vdash M : \mathbb{N}_s}{E \vdash \mathtt{succ}(M) : \mathbb{N}_s} \qquad\qquad [If]\ \ \frac{E_1 \vdash M : \mathbb{N}_s \quad E_2 \vdash N_i : T}{E \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : T}\ (\ddagger)$$

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

$$[Lift]\ \ \frac{E \vdash M : S}{E \vdash M : \llcorner S \lrcorner_s} \qquad\qquad [Seq]\ \ \frac{E_1 \vdash N : \llcorner S \lrcorner_s \quad E_2, x{:}S \vdash M : T \quad T \text{ partial}}{E \vdash \mathtt{seq}\ x = N\ \mathtt{in}\ M : T}\ (\ddagger)$$

$$[Rec]\ \ \frac{E, x{:}S \vdash \lambda y.M : S \quad S \text{ pointed}}{E \vdash \mu x^S.\lambda y.M : S} \qquad\qquad [Subs]\ \ \frac{E' \vdash M : T' \quad E \leq E' \quad T' \leq T}{E \vdash M : T}$$

$$[Ass]\ \ \frac{E \vdash x : \mathtt{refw}_s(S) \quad E \vdash V : S}{E \vdash x := V : \mathtt{COM}} \qquad\qquad [Deref]\ \ \frac{E \vdash z{:}\mathtt{refr}_s(S) \quad E, x{:}S \vdash M : T}{E \vdash \mathtt{let}\ x = \mathtt{!}z\ \mathtt{in}\ M : T}$$

$$[Skip]\ \ \frac{-}{E \vdash \mathtt{skip} : \mathtt{COM}} \qquad\qquad [New]\ \ \frac{E \vdash V : S \quad E, x : \mathtt{ref}_s(S) \vdash M : T}{E \vdash \mathtt{new}\ x \mapsto V\ \mathtt{in}\ M : T}$$

($\ddagger$) $E_{1,2} \leq E,\ s \sqsubseteq \mathsf{protect}(T) \sqcap \mathsf{tamp}(E_2)$
($\dagger\dagger$) $s \sqsubseteq \mathsf{tamp}(E)$; if T is total then $\mathtt{cod}(E)$ are non-reference types.

Fig. 17. Typing Rules of Imperative Call-by-Value DCC

the next, we outline how we can directly apply the π^{LAM}-calculus to extend DCCv in § 4.5 with those imperative constructs, while still remaining inside sequential programs. The syntax is extended as:

$$\begin{array}{rcl} M & ::= & \ldots \mid x := V \mid \mathtt{let}\ x = !y\ \mathtt{in}\ M \mid \mathtt{new}\ x \mapsto V\ \mathtt{in}\ M \mid \mathtt{skip} \\ V & ::= & x \mid \lambda x.M \mid \mu\lambda z.\lambda y.M \mid \mathtt{skip} \end{array}$$

where V denotes a value (values are variables, natural numbers, $\mathtt{skip}$, abstractions and recursions). For types we only extend total types:

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

Unlike the language in Section 7, we use $\mathtt{COM}$ as part of the types for expressions. $(T)_s$, as well as the three forms of tamper levels, $\mathsf{protect}(T)$, $\mathsf{protect}^{\mathcal{E}}(T)$ and $\mathsf{tamp}(E)$, are defined precisely as in §7.4. Further we demand the structural security on types which is defined by the identical clause as in Definition 7.5. The typing rules are given in Figure 17.[5] The rules closely follow the secrecy typing of commands/expressions of the extended VS-calculus, so may not need illustration. We can easily encode the imperative DCCv into π^{LAR} following Figure 9 and Figure 16, based on which the noninterference is proved following the proofs of the corresponding results in Section 7.

[5]We do not include the secrecy level enhancement in (App), unlike in §4.5 since the corresponding operation was not considered in π^{LAR}. At this point we leave it open how this operation can be incorporated into the imperative DCCv.

The imperative DCCv is an expressive calculus which 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 way: for example, "`while` e `do` c" is encoded as follows (assuming e and c are translated into N and M, respectively, and using the shorthand ;):

$$(\mu z.\, \lambda y.\, \mathtt{if}\ N\ \mathtt{then}\ M; (z\,\mathtt{skip})\ \mathtt{else}\ y)\ \mathtt{skip}$$

We can verify that the encoding yields precisely the same side condition as given in Figure 15.

There is another significant way to do the secrecy analysis for imperative higher-order computation. We observe that the π-calculus-based secrecy analyses in the preceding work (including our previous work), cf. [22; 29; 56], are based on the idea of recording a secrecy level explicitly (whose origin can be found in the initial work on static secrecy analysis by Denning and Denning [16]), where the sequent may be written:

$$\vdash P \triangleright A, s$$

which means P tampers the environment at a secrecy level at most s during its run. Since we explicitly record the tampering level of P independently from A, we may call this approach an *explicit approach*. In contrast, the approach in the present paper does not use such s but derive this level from A, which we may call *implicit approach*. In principle, π^{LA} and π^{LAR} can adopt both kinds of approaches, of which the present work has focussed on the implicit one. Since comparisons between these two approaches offer basic insight on secrecy in stateful computation, we outline an explicit approach for the secrecy typing of π^{LAR} in the following, showing how this secrecy analysis leads to an alternative formulation of imperative DCCv.

In the explicit approach, the secrecy typing for π^{LAR} takes the shape we mentioned already, $\vdash P \triangleright A, s$, where s is called an *explicit tampering level*. The approach adds this explicit tampering level to indicate the level of tampering by writing, including the indirect one through ?-actions. Channel/action types stay precisely the same (including the structural secrecy). For secrecy typing rules, the only changes are:

—An explicit tampering s is manipulated so that: (1) for each write action at a reference type of level s, as well as for each $?_{\mathrm{L}}$-action of form $(\vec{\rho})^{?_{\mathrm{L}}}_{s}$, we place s or lower; (2) in parallel composition, we take the meet of two levels; and (3) in replication, we record the explicit tampering level in the replication type, and clear that level so that we start from any level (in particular one can start from the high level).

—In the secrecy condition in $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$, $(\mathsf{Bra}^{\downarrow_{\mathrm{L}}})$ and $(\mathsf{Bra}^{\downarrow_{\mathrm{A}}})$, the receiving level of input should be the same as, or lower than, the explicit tampering level plus the tampering level of free linear/affine actions to be suppressed, instead of using only implicit tampering. Similarly, in replicated inputs (cf. Figure 11), we demand a receiving level to be the same as, or lower than, the given explicit tampering level, instead of using implicit tampering.

This typing can be shown to guarantee the noninterference as far as sequential processes in the sense of [10] go. The calculus allows lowering of the explicit tampering

$[Var]\ \ E, x : S \vdash x : S$

$[Num]\ \ E \vdash n : \mathbb{N}_s$

$[Succ]\ \ \dfrac{E \vdash V : \mathbb{N}_s}{E \vdash \mathtt{succ}(V) : \mathbb{N}_s}$

$[If]\ \ \dfrac{E_1 \vdash V : \mathbb{N}_{s'} \quad E_2 \vdash N_i : T, s}{E \vdash \mathtt{if}\ V\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : T, s}\ (\ddagger)$

$[Lam]\ \ \dfrac{E, x : S \vdash M : T, s}{E \vdash \lambda x.M : S \stackrel{s}{\Rightarrow} T}\ (\dagger\dagger)$

$[App]\ \ \dfrac{E \vdash V : S \stackrel{s}{\Rightarrow} T \quad E \vdash W : S}{E \vdash VW : T, s}$

$[Lift]\ \ \dfrac{E \vdash V : S}{E \vdash V : \llcorner S \lrcorner_s}$

$[Seq]\ \ \dfrac{E \vdash N : \llcorner S \lrcorner_{s'}, s'' \quad E, x{:}S \vdash M : T, s \quad T \text{ partial}}{E \vdash \mathtt{seq}\ x = N\ \mathtt{in}\ M : T, s \sqcap s''}\ (\ddagger)$

$[Promote]\ \ \dfrac{E \vdash V : T}{E \vdash V : T, s}$

$[Let]\ \ \dfrac{E \vdash N : S, s'' \quad E, x{:}S \vdash M : T, s}{E \vdash \mathtt{seq}\ x = N\ \mathtt{in}\ M : T, s \sqcap s''}\ (\ddagger)$

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

$[Subs]\ \ \dfrac{E' \vdash M : T', s' \quad E \leq E' \quad T' \leq T \quad s \sqsubseteq s'}{E \vdash M : T, s}$

$[Ass]\ \ \dfrac{E \vdash x : \mathtt{refw}_s(S) \quad E \vdash V : S}{E \vdash x := V : \mathtt{COM}, s}$

$[Deref]\ \ \dfrac{E \vdash z{:}\mathtt{refr}_{s'}(S) \quad E, x{:}S \vdash M : T, s}{E \vdash \mathtt{let}\ x = \mathtt{!}z\ \mathtt{in}\ M : T}$

$[Skip]\ \ \dfrac{-}{E \vdash \mathtt{skip} : \mathtt{COM}}$

$[New]\ \ \dfrac{E \vdash V : S \quad E, x : \mathtt{ref}_s(S) \vdash M : T, s}{E \vdash \mathtt{new}\ x \mapsto V\ \mathtt{in}\ M : T, s}$

($\ddagger$) $s' \sqsubseteq \mathsf{protect}(T) \sqcap s$
($\dagger\dagger$) if T is total then $\mathtt{cod}(E)$ are non-reference types.

Fig. 18. Typing Rules of Imperative Call-by-Value DCC (with explicit tampering)

level in each sequent, intuitively because the termination level (if any) is distinct from the explicit tampering level.

From this secrecy typing, we can immediately obtain an alternative secrecy typing for DCCv, with the identical syntax of programs. The typing sequents have the forms $\Gamma \vdash M : \alpha, s$ and $\Gamma \vdash V : \alpha$ (the latter corresponding to a replication with a cleared tampering level: s is called "pc" in the literature). The typing rules are given in Figure 18. All altered secrecy conditions directly come from the explicit secrecy typing for π^{LAR} discussed above. Note these two approaches yield quite close secrecy typing, with a difference in the way to measure the level of **?**, one based on interface types and one based on accumulation of the levels of each performed action.

In the next subsection, further illustration on these two approaches will be given in the context of comparisons with related work.

8.2 Related Work

The general theme in the present study is to use the typed π-calculi as a language-independent basis for describing, reasoning about, and analysing diverse typed language constructs; and to explore, as a possible application, a type-based secrecy analysis of programming languages with distinct type disciplines and operational behaviours through the secrecy typing of typed π-calculi. While the present work is only a preliminary experiment to examine the potential of the π-calculus for this and other related purposes, the case study may demonstrate one of the starting points towards the overall goal we have set out at the beginning. In the following we discuss several related studies and point out some of the remaining topics.

Integrated Function Types. The foregoing study of types in programming languages has concentrated on types for functional programming languages. It is thus no surprise that a few prominent examples of integrated type disciplines are found among function-based type disciplines, which often use monads. Basic examples include pointed types [35; 51] and the incorporation of imperative constructs in Haskell [36]. 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 π-calculus, and shows its significance in secrecy analysis.

Security Analyses in Process Calculi. Secrecy and other security issues in processes are widely studied recently, cf. [1; 62; 19; 22; 56]. [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 programming languages. In this context, one of the challenging topics is the integration of the technologies as experimented in the present paper with secrecy and security concerns in distributed computing. See also [33] for more comparison with the control flow analysis of the π-calculus [7; 8].

For type-based secrecy analysis proper, as we already observed in the previous subsection, the π-calculus-based secrecy analyses in the preceding work (including our own), cf. [22; 29; 56], have been based on the explicit recording of a secrecy level, in contrast to the implicit approach we have taken in the preceding sections.

Type-Based Secrecy Analysis for Programming Languages. 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 [54; 57; 3; 21], the dependency core calculus [3] is a powerful functional metalanguage for secrecy, using pointed types [35; 51]. 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 π-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 Section 7 and §8.1 above).

Smith and Volpano (cf.[65; 66; 70]) studied various aspects of secrecy in imperative languages. Sequential procedures are studied in [70]. Multi-threading is studied in [66], whose typability was enlarged by our work with Vasconcelos [29] using the π-calculus, based on which a further enhancement was done in [65]. As we have seen in Proposition 7.10, our calculus is a conservative extension of the possibilistic part of the calculus in [65], 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 π^{LA} on the other. The incorporation of execution steps into secrecy typing [65] into the present framework is one of the remaining topics.

Boudol and Castellani [14] also studied a language similar to [65]. One of the significant features is the use of bisimulation for formulating and proving noninterference, leading to a stronger property. In this context, the use of secrecy bisimulation in π^{LA} [75] for proving similar results would be worth exploring. Another topic is the incorporation of *scheduler* into the π^{LAM}-calculus to enrich language constructs, following [14].

There are two prominent recent work which presented secrecy analysis for the combination of higher-order procedures and imperative features, one by Myers, Zdancewic and others [49; 76] (using Java) and another by Pottier and Simonet [58] (using ML). These work are based on the explicit typing approach, while the present work adopted the implicit approach which centres on the linear/affine type discipline for the π-calculus and its basic secrecy refinement. The implicit approach offers a basic framework where the effects of each action are represented in the interface types of a given process (hence of a program through an embedding). We have also shown an alternative secrecy typing for π^{LAR} and the induced secrecy typing for the imperative extension of DCCv in §8.1. These two paradigms of secrecy typing, based on the same linear/affine π-calculus, gives a clear technical understanding of the basic elements of secrecy for imperative higher-order computation (on concurrency which we shall discuss later). These elements include:

(1) A choice in secrecy typing between implicit typing and explicit typing.
(2) The use of distinction between linearity (totality) and affinity (partiality) for fine-grained secrecy analysis. Relatedly, the treatment of termination as a distinct observable in addition to writing effects and returned values, both for implicit and explicit approaches.
(3) The use of structural security (i.e. a reference type should have a lower-level than its content type) for consistent treatment of general references.

In the following, we compare the present work with those by Myers, Zdancewic and others [49; 76] by Pottier and Simonet [58] in the light of these elements as well as other aspects, mainly using the two versions of imperative DCCv presented in 8.1 as a tool for illustration.

Implicit vs. Explicit. We first illustrate the difference between the implicit approach and the explicit approach (the latter taken in the work by Myers, Zdancewic and others [49; 76] by Pottier and Simonet [58]). We use the syntax of imperative DCCv. Below we write `unit` for `COM` and () for `skip` for brevity. $\lambda().M$ stands for $\lambda x^{\texttt{unit}}.M$ with $x \notin \mathsf{fv}(M)$. First, let:

$$M \stackrel{\text{def}}{=} \texttt{new}\ u^{\texttt{L}} \mapsto 0\ \texttt{in}\ \lambda().(y^{\texttt{H}} = 0\,;\, \lambda().u := !u + 1;\ !u) \tag{28}$$

which is typed as, using the implicit approach in Figure 17:

$$\vdash^{\mathsf{i}} M : \texttt{unit} \stackrel{\texttt{L}}{\Rightarrow} (\texttt{unit} \stackrel{\texttt{L}}{\Rightarrow} \mathbb{N}_{\texttt{L}}). \tag{29}$$

and, in the explicit one in Figure 18:

$$\vdash^{\mathsf{e}} M : \texttt{unit} \stackrel{\texttt{H}}{\Rightarrow} (\texttt{unit} \stackrel{\texttt{L}}{\Rightarrow} \mathbb{N}_{\texttt{L}}). \tag{30}$$

Assume $\mathtt{let}\ x = M\ \mathtt{in}\ x()()$. Then assigning the type in (29) to x shows $x()()$ is a low-level behaviour, which is correct. This is the same if we assign the type in (30) to x. However giving the type in (29) to x says that $x()$ is also low; while the explicit approach says $x()$ is high, giving a better analysis. Note such a difference only comes out when this expression is partially used.

As another example which shows a difference in a different direction, if we consider $\mathtt{new}\ w \mapsto 0\ \mathtt{in}\ (\lambda y^{\mathtt{L}}.y := 1)w$, where a name is abstracted/hidden, an effect is given a better analysis in the implicit approach, since the expression becomes a high-level expression immediately from its type $\mathtt{unit}$; while, in the explicit approach, it is a low-level expression. However, if we start from w and y without secrecy assignment, we may as well infer the level $\mathtt{H}$ for these names so that it in effect becomes a high-level expression even with the explicit approach.

Overall, the implicit approach seems to offer a cleaner analysis for hiding, in the sense that a hiding leads to cancellation of the effect of tampering; at the same time, as the first example show, the implicit approach may lead to a less accurate analyses in measuring the effects to an action in the environment.

Linearity (or totality). The origin of the use of the distinction between totality and partiality in secrecy analysis can be found in both DCC [3] and imperative secrecy calculi studied by Smith and Volpano [66]. The above cited work [49; 58] do not directly use this distinction in type structures, even though JFlow [49] partly uses the distinction (the work by Zdancewic and Myers [76] studies linearity in detail, on which we discuss later). By using imperative DCCv, we can illustrate the significance of linearity clearly. Consider the following program.

$$\text{let } y = V() \text{ in let } z = W() \text{ in } () \tag{31}$$

where we set $V \stackrel{\text{def}}{=} \lambda().x^{\mathtt{H}} := 0$ and $W \stackrel{\text{def}}{=} \lambda().u^{\mathtt{L}} := 1$. If we replace $\mathtt{let}$ with $\mathtt{seq}$, and stipulate that the assignment should always be considered to be affine, we can*not* lower the level of the assignment $x^{\mathtt{H}} := 0$, hence (regardless of use of the explicit and implicit approaches) we cannot type (31). The use of linearity easily analyse the lack of unsafe flow in this example, demonstrating the use of linearity leads to strictly more general secrecy typing (observe the argument holds both for the explicit and implicit approaches).

The restriction that linear (total) arrow types should not mention references, as given in our VS language and the imperative DCCv, is quite strict: however, as we noted in Remark 5.2, the use of a variant of effect typing can enlarge the typable linear (terminating) behaviour, which we believe to be enough for many practical purposes.

Strong/Weak Noninterference. The distinction between linearity and affinity, and the idea to give an explicit secrecy level to the latter, also leads to a comprehensive treatment of both strong and weak noninterference in a single framework. These approaches differ in that whether we should take the termination as an observable when considering noninterference in sequential programs (this point becomes subtle because, if there is only a single thread of control, no other programs can detect its termination). In the present context, if we make the termination level strictly higher than what we wish to observe, then we lose (can ignore) the termination observable, but we still care the difference in low-level states when the program terminates (this case is obtained as a simple refinement of Theorem 7.20, with essentially the same proof). If on the other hand we set the termination level sufficiently low, we obtain the strong noninterference. This aspect may not be found in the above cited work [49; 58], both of which counts the level of termination observables.

Structural Security. As we already discussed in Remark 7.6, structural security is not restrictive when it is combined with read/write subtyping. The above cited work [49; 58; 76] imposes an essentially the same constraint, though in a different format.

Richer Language Constructs. Both of the work by Myers [49] and the work by Pottier and Simonet [58] study secrecy analysis for a larger class of language constructs than treated in the present paper, except for concurrency, including exception, polymorphism and objects. One of the merits of the inquiries as given in the present note would be elucidation of each technical element of secrecy analysis in a distilled setting. It is an interesting topic for further study to examine whether, and how, secrecy of these constructs may be analysed through the embedding into typed processes. Another challenging topic is incorporation of such dynamic aspects as explicit communication of secrecy labels and declassification in [49] into processes, which would require a non-trivial enrichment of the presented framework.

Further Discussions. One of the suggestions from the comparative discussions above would be the merit of incorporating the direct use of linearity into the framework in [49] and [58]. Not only the totality of commands such as assignment, but also some of the significant instances of the totality of methods/procedures may be easily analysable using static typing. We also observe that treatment of linearity in the presence of recursive types is the topic which has been studied in the past, cf. [35], even though a clean extension of these studies to the imperative setting is an open issue (we may be able to use effect typing as suggested in Remark 5.2).

Concurrency and Low-level Computation. Secrecy for concurrent computation has many aspects, including the countermeasure against timing attack, treatment of nondeterminism, observability of behaviours of other threads/processes, primitives for parallelism, and incorporation of synchronisation constructs. The present work studies secrecy in concurrency in its most basic setting. Depending on applications, we may as well consider different kinds of noninterference properties (for example, if we only do not want a high security datum to be leaked to a public channel, the possibilistic secrecy as we have treated would be enough, while in an environment where hostile threads can detect shared state as well as termination/resource usage, a more stringent notion of noninterference would be necessary). We in particular believe that the π-calculus may offer a useful setting where the interplay between secrecy analysis and diverse concurrency constructs can be studied on a uniform basis.

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 by Zdancewic and Myers [76] 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, [76] does not treat multi-threading, and is not (intended as) an extension of the language in [66; 65]. The incorporation of the dynamics and types in [76] into the π-calculus is an interesting topic for further study, cf. [26].

8.3 Further Topics

In addition to those which are mentioned through the above comparative discussions, we believe the following topics are worth studying.

(1) The investigation into the use of explicit secrecy for the full π^{LAR} (with concurrency) and its extensions.

(2) Generalisation of Theorem 7.20 to the full π^{LAR}.

(3) Treatment of language constructs such as exceptions and jumps, as well as a technical framework to coherently integrate such run-time mechanisms as degradation [49] in the typed π-calculi.

(4) Polymorphic extension of the secrecy typing, cf. [12]. Similarly incorporation of recursive types while still making the effective use of linearity.

(5) Embedding results of major programming languages and their secrecy analysis in securely typed processes, including variants of languages treated in [49; 58]. As we noted, clean incorporation of dynamic notion of secrecy control would be an interesting technical challenge. Does the process representation offers an additional insight for dynamic constructs as treated in [49]?

(6) An integrated type inference which can infer both linearity and secrecy in a richer class of behaviours, for example those which are found in real-world programming languages. Ideally the analysis would first infer linearity with little, or no, annotation in programs; then the type inference for secrecy is done on its basis. The latter can employ methods cultivated from the accumulated preceding studies [66; 65; 49; 58].

(7) Precise understanding of different notions of safe information flow and their practical consequences in each class of behaviours, including functional, imperative and concurrent ones.

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] Bodei, C, Degano, P., Nielson, F., and Nielson, H. Control Flow Analysis of π-Calculus. *CONCUR'98*, LNCS 1466, 84–98, Springer, 1998.

[8] C. Bodei, P. Degano, F. Nielson, and H.R.Nielson. Static Analysis of Processes for No Read-Up and No-Write-Down. In Proc. FoSSaCS'99, volume 1578 of Lecture Notes in Computer Science, pages 120134, Springer-Verlag, 1999.

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

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

[11] The full version of [10]. Available at www.dcs.qmw.ac.uk/˜kohei.

[12] Berger, M., Honda, K. and Yoshida, N., Genericity and the π-Calculus, *FoSSaCs03*, LNCS 2620, 103–119, Springer, 2003. A long version, 70 pages, April 2004, is available from www.doc.ic.ac.uk/yoshdia.

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

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

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

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

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

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

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

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

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

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

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

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

[25] Honda, K., Process Logic and Duality, 2004, Available from: `http://www.dcs.qmw.ac.uk/~kohei/logics/`.

[26] Honda, K., Yoshida, N. and Berger, M., Control in the π-Calculus, *Proc. CW'04*, ACM Press, 2004.

[27] 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.

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

[29] 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.doc.ic.ac.uk/˜yoshida.

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

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

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

[33] Honda, K. and Yoshida, N., Noninterference Proofs from Flow Analysis. Department of Computing Technical Report, Imperial College, July, 2002. Updated, April, 2004. 48 pages. Available at `http://www.doc.ic.ac.uk/˜yoshida`.

[34] Honda, K. and Yoshida, N., *Addendum to "Uniform Type Structure for Secure Information Flow": Subject Reduction with Inflation.* 5pp. August, 2003. Available at `http://www.doc.ic.ac.uk/˜yoshida`.

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

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

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

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

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

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

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

[42] Lampson, B. W., A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.

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

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

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

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

[47] Milner, R., Polyadic π-Calculus: a tutorial. *Proceedings of the International Summer School on Logic Algebra of Specification*, Marktoberdorf, 1992.

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

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

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

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

[52] Eugenio Moggi. Notions of computation and monads. *Inf. & Comp.*, 93(1):55–92, 1991.

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

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

[55] Pierce, B.C., *Types and Programming Languages*, MIT, 2002.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[72] 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.

[73] 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.doc.ic.ac.uk/˜yoshida.

[74] Yoshida, N., Berger, M. and Honda, K., Strong Normalisation in the π-Calculus, *LICS'01*, 311–322, IEEE, 2001. The full version as MCS technical report, 2001-09, University of Leicester, 2001. Available at www.doc.ic.ac.uk/˜yoshida.

[75] 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.doc.ic.ac.uk/˜yoshida.

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

A. SUBJECT REDUCTION

In this section, we prove the following result for the secrecy typing for π^{LAR} in Section 6 (subsuming the same property for π^{LA} in Section 2, π^{LA} with the secrecy typing in Section 3 and π^{LAR} in Section 5). The case for the inflated extension is discussed in [34].

(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 of the secrecy typing for π^{LAR} in Section 6 is satisfied as above, then the subject reduction of the π^{LA}, π^{LAR} and π^{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 [74]. We also follow the same routine as given in the proof of Proposition 3 in [73] for references. What corresponds to Lemma A.1 of [74] (well-definedness of operators) is easy. A basic lemma follows, which corresponds to Lemma A.2 in [74].

Lemma A.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* $A/\vec{x} = A_0$, $x_i : \tau_i \in |A|$, $\mathsf{md}(\tau_i) \in \mathcal{M}_{!,\downarrow} \cup \{\updownarrow\}$, *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 $!_{\mathrm{R}}$ and $?_{\mathrm{R}}$ as $!_{\mathrm{A}}$ and $?_{\mathrm{A}}$, respectively, we can essentially follow the reasoning given in [11; 74]. (i,ii) are immediate from the definitions. (iii) is obvious since there is no edge to input and $\updownarrow$ nodes. (iv) is because we can write $A = (x_i : \tau_i \rightarrow A_i), A'$ by the side condition (note A_i may be $\emptyset$). Then by $A \asymp B$, obviously $A_0 = (A_i, A') \asymp B$. Hence we have $(A \odot B)/\vec{x} = (A/\vec{x} \odot B/\vec{x}) = A_0 \odot B$. Similarly for (v). (vi) is by $\tau \odot \tau = \tau$ with $\mathsf{md}(\tau) \in \mathcal{M}_{?}$. (vii) uses (v) and (vi). $\square$

Lemma A.2 (substitution lemma)

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

(*2*) (client type) *Suppose* $\vdash_{\mathsf{sec}} P \triangleright x : \tau, A$ *with* $\mathsf{md}(\tau) \in \mathcal{M}_{?}$ *and* $A(y) = \tau$. *Then we have* (a) $\vdash_{\mathsf{sec}} P\{y/x\} \triangleright A$ *and* (b) $\mathsf{tamp}(x : \tau, A) = \mathsf{tamp}(A)$.

PROOF. (b) of (1,2) is by definition. (a) is proved by induction on terms.

The only interesting case is $x \in \mathsf{fn}(P)$ and either a parallel composition, an input or a branching input. We prove (1) in the case of (Par) and (3) in the case of the replicated input. Suppose $\vdash_{\mathsf{sec}} P_1 | P_2 \triangleright x : \tau, A$. Since we can always permute (Weak) and (Sub) with (Par), we only consider the last applied rule is (Par).

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

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

$$\vdash_{\mathsf{sec}} (P_1 \mid P_2)\{y/x\} \triangleright A_1 \odot A_2, y : \tau \tag{32}$$

By inductive hypothesis, we have $\vdash_{\mathsf{sec}} P_1\{y/x\} \triangleright A_1, y : \tau$. Since $x, y \notin \mathsf{fn}(A_2)$, we have $(A_1, y : \tau) \asymp A_2$ and $(A_1, y : \tau) \odot A_2 = A, y : \tau$. Hence by $(P_1 \mid P_2)\{y/x\} = (P_1\{y/x\} \mid P_2)$, we have (32) as required.

Next suppose $\vdash_{\mathsf{sec}} !a(\vec{z}).Q \triangleright A, x : \tau$ with $A(y) = \tau$. Then the only interesting case is the last applied rule is $(\mathsf{In}^{!_{\mathrm{A}}})$ as follows.

$$\frac{s \sqsubseteq \mathsf{tamp}(A, x : \tau) \qquad \vdash_{\mathsf{sec}} Q \triangleright \vec{z} : \vec{\tau}\,,\, ?A^{\text{-}a}, x : \tau}{\vdash_{\mathsf{sec}} !a(\vec{z}).Q \triangleright a : (\vec{\tau})^{!_{\mathrm{A}}}_{s}, A, x : \tau}$$

Then we prove

$$\vdash_{\mathsf{sec}} !a(\vec{z}).Q\{y/x\} \triangleright a : (\vec{\tau})^{!_{\mathrm{A}}}_{s}, A \tag{33}$$

By inductive hypothesis, we have

$$\vdash_{\mathsf{sec}} Q\{y/x\} \triangleright \vec{z} : \vec{\tau}\,,\, A \tag{34}$$

By (b), we know $s \sqsubseteq \mathsf{tamp}(A, x : \tau) = \mathsf{tamp}(A)$. Hence by applying $(\mathsf{In}^{!_{\mathrm{A}}})$ to the above again, we conclude (33) as desired. $\square$

Lemma A.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) $\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{x}\,\mathtt{read}\langle c\rangle \triangleright A$ *implies* $\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{c}\langle y\rangle \triangleright A$
(v) $\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{x}\,\mathtt{write}\langle wc\rangle \triangleright A$ *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 [74]; 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 $\mathcal{M}_{!} \cup \{\updownarrow\}$ because of the definition of (Res). Hence we can use (iv) and (v) of Lemma A.1 as in [74]. For (ii) and (iii), we can directly use (v,vi,vii) of Lemma A.1 together with Substitution Lemma to prove (ii) and (iii) as in [74]. Hence we only have to prove (iv) and (v).

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

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{x}\,\mathtt{read}\langle c\rangle \triangleright x : \mathsf{ref}_s\langle\tau\rangle, y : \overline{\tau}_1, c : (\tau_2)^{\uparrow_{\mathrm{L}}} \tag{35}$$

and the last rule is (Par). Then we prove:

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{c}\langle y\rangle \triangleright x : \mathsf{ref}_s\langle\tau\rangle, y : \overline{\tau}_1, c : (\tau_2)^{\uparrow_{\mathrm{L}}} \tag{36}$$

Since (35) is inferred by (Par), we have:

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \triangleright x : \mathsf{ref}_s\langle\tau\rangle, y : \overline{\tau}_1 \quad \text{and} \quad \vdash_{\mathsf{sec}} \overline{x}\,\mathtt{read}\langle c\rangle \triangleright x : \mathsf{rw}_s\langle\overline{\tau}\rangle, c : (\tau_2)^{\uparrow_{\mathrm{L}}}$$

with $\overline{\tau} \leq \overline{\tau}_1$. The left-hand side read agent is inferred from

$$\vdash_{\mathsf{sec}} \overline{x}\,\mathtt{read}\langle c\rangle \triangleright c : (\tau')^{\uparrow_{\mathrm{L}}}, x : \mathsf{r}_{s'}\langle\overline{\tau}'\rangle$$

for some $s' \sqsubseteq s$, $\overline{\tau}' \leq \overline{\tau}$ and $\tau' \leq \tau_2$. By (Out), we have:

$$\vdash_{\mathsf{sec}} \overline{c}\langle y\rangle \triangleright c : (\tau')^{\uparrow_{\mathrm{L}}}, y : \overline{\tau}'$$

Then noting $\overline{\tau}' \leq \overline{\tau}$ and $\overline{\tau} \leq \overline{\tau}_1$ implies $\overline{\tau}' \leq \overline{\tau}_1$, by (Sub), we have:

$$\vdash_{\mathsf{sec}} \overline{c}\langle y\rangle \triangleright c : (\tau_2)^{\uparrow_{\mathrm{L}}}, y : \overline{\tau}_1 \tag{37}$$

By (iv) of Lemma A.1, we know $y : \overline{\tau}_1 \odot y : \overline{\tau}_1 = y : \overline{\tau}_1$. An application of (Par) to $\mathsf{Ref}\langle xy\rangle$ and (37) again obtains (36), as required.

(v, write): Assume

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \mid \overline{x}\,\mathtt{write}\langle wc\rangle \triangleright x : \mathsf{ref}_s\langle\tau\rangle, w : \overline{\tau}_1, c : ()^{\uparrow_{\mathrm{L}}}, y : \overline{\tau}_2 \tag{38}$$

We shall prove:

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xw\rangle \mid \overline{c} \triangleright x : \mathsf{ref}_s\langle\tau\rangle, w : \overline{\tau}_1, c : ()^{\uparrow_{\mathrm{L}}}, y : \overline{\tau}_2 \tag{39}$$

We assume (38) is inferred by (Par), so we have:

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xy\rangle \triangleright x : \mathsf{ref}_s\langle\tau\rangle, y : \overline{\tau}_2 \quad \text{and} \quad \vdash_{\mathsf{sec}} \overline{x}\,\mathtt{write}\langle wc\rangle \triangleright x : \mathsf{rw}_s\langle\overline{\tau}\rangle, w : \overline{\tau}_1, c : ()^{\uparrow_{\mathrm{L}}}$$

with $\overline{\tau} \leq \overline{\tau}_2$. The write agent in the right-hand side is inferred from:

$$\vdash_{\mathsf{sec}} \overline{x}\,\mathtt{write}\langle wc\rangle \triangleright x : \mathsf{w}_{s'}\langle\overline{\tau}'\rangle, w : \overline{\tau}', c : ()^{\uparrow_{\mathrm{L}}}$$

for some $s' \sqsubseteq s$ and $\overline{\tau} \leq \overline{\tau}'$ and $\overline{\tau}' \leq \overline{\tau}_1$ (hence $\overline{\tau} \leq \overline{\tau}_1$). By (Ref), we have:

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xw\rangle \triangleright x : \mathsf{ref}_s\langle\tau\rangle, w : \overline{\tau}$$

Noting $\overline{\tau} \leq \overline{\tau}_1$, we have:

$$\vdash_{\mathsf{sec}} \mathsf{Ref}\langle xw\rangle \triangleright x : \mathsf{ref}_s\langle\tau\rangle, w : \overline{\tau}_1 \quad \text{and} \quad \vdash_{\mathsf{sec}} \overline{c} \triangleright c : ()^{\uparrow \mathrm{L}} \tag{40}$$

Now by applying (Par) to above, and then using (Weak) for y, we have (39), as desired.

B. FURTHER DISCUSSIONS ON DCC

B.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 B.2 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 Γ 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 argument 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 $y : \mathbb{B}_{\mathtt{H}}, x : \mathbb{B}_{\mathtt{L}} \vdash \mathtt{case}\ x^{\mathtt{L}}\ \mathtt{of}\ \mathtt{inl}().\mathtt{inl}()\ \mathtt{or}\ \mathtt{inr}().\mathtt{inr}() : \mathbb{B}_{\mathtt{H}}$, we can infer $y : \mathbb{B}_{\mathtt{H}} \vdash \mathtt{bind}\ x = y\ \mathtt{in}\ \mathtt{case}\ x^{\mathtt{L}}\ \mathtt{of}\ \mathtt{inl}().\mathtt{inl}()\ \mathtt{or}\ \mathtt{inr}().\mathtt{inr}() : \mathbb{B}_{\mathtt{H}}$. 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 $w : \mathbb{B}_{\mathtt{M}} \Rightarrow \mathbb{B}_{\mathtt{H}}, y : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{M}}, x : \mathbb{B}_{\mathtt{L}}, z : \mathbb{B}_{\mathtt{H}} \vdash w(yx) : \mathbb{B}_{\mathtt{H}}$, we infer $w : \mathbb{B}_{\mathtt{M}} \Rightarrow \mathbb{B}_{\mathtt{H}}, y : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}}, z : \mathbb{B}_{\mathtt{H}} \vdash \mathtt{bind}\ x = z\ \mathtt{in}\ w(yx) : \mathbb{B}_{\mathtt{H}}$ (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 flow from

the argument to the result), what y would produce by yx can only be $\mathtt{H}$-level information, from which w will again produce $\mathtt{H}$-level information, so, at the end of the day, it is safe.

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

—[*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.

Remark B.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.

Remark B.2 (subject reduction in DCC) For reference we give instances of violation of subject reduction in DCC in the original typing rules [3] (cf. [32]). We show examples in both explicitly and implicitly typed systems. For the explicitly typed system as in the original DCC, a violation comes via the coercion $\eta_l M$: for example we have $x : \mathtt{unit} \vdash \eta_{\mathtt{H}} x : (\mathtt{unit})_{\mathtt{H}}$ and $\eta_{\mathtt{H}}\ x \longrightarrow x$, but $x : \mathtt{unit} \not\vdash x : (\mathtt{unit})_{\mathtt{H}}$. The same remedy as we presented in Figure 6 can be used for the explicit typing. In implicitly typed systems, the issue arises indirectly: for an example, we can take $y : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}}, z : \mathbb{B}_{\mathtt{H}} \vdash \mathtt{bind}\ x = z\ \mathtt{in}\ yx : \mathbb{B}_{\mathtt{H}}$. Then we have $\mathtt{bind}\ x = z\ \mathtt{in}\ yx \longrightarrow yz$. However $y : \mathbb{B}_{\mathtt{L}} \Rightarrow \mathbb{B}_{\mathtt{H}}, z : \mathbb{B}_{\mathtt{H}} \vdash yz : \mathbb{B}_{\mathtt{H}}$ is *not* well-typed if we are to use the standard application rule (which does not inflate the argument type, cf. [*App*], Figure 6).

Remark B.3 (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 admissibility of [*BindM*] is immediate. For [*UnitM*], we simultaneously establish the following two statements:

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

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

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

B.2 Subject Reduction in DCC/DCCv

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

Lemma B.4 *(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 $\Gamma \cdot x : T \vdash M_1M_2 : T'$ 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 $\Gamma \vdash (M_1M_2)\{N/x\} : T'$.
- *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 B.4 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 B.4 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 B.4 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 [32]). 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.

C. CONSERVATIVITY RESULT FOR THE SMITH CALCULUS

We first observe we have only to use *[Var]* and *[Const]* from Figure 14 by the restriction to the first-order value types in the Smith Calculus. Thus $x := v$ always tampers at s whenever $E(x) = \mathsf{ref}_s\langle \mathbb{N}_s \rangle$. By the rule induction, we can show s in $E \vdash c : s\mathtt{cmd} \Uparrow_{s'}$ in [65] 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 [65]. 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.

D. PROOFS FOR SECTION 7

D.1 Proposition 7.12 (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 cases of $T_i = S_i \stackrel{s_i}{\Rightarrow} U_i,\ (i = 1, 2)$, $T_1 = \mathtt{ref}_s(S)$ and $T_2 = \mathtt{refr}_{s'}(S')$ and $T_1 = \mathtt{refw}_s(S)$ and $T_2 = \mathtt{refw}_{s'}(S')$. Others are similar.

Case $S_1 \stackrel{s_1}{\Rightarrow} U_1 \leq S_2 \stackrel{s_2}{\Rightarrow} U_2$. Assume $S_2 \leq S_1$, $U_1 \leq U_2$ and $s_2 \sqsubseteq s_1$. Then by induction, and $S_2^\circ \leq S_1^\circ$ and $U_1^\bullet \leq U_2^\bullet$. Note for the input type, the security level is contravariant, while the carried types are covariant. Since $S_2^\circ \leq S_1^\circ$ implies $\overline{S_1^\circ} \leq \overline{S_2^\circ}$, we have: $(S_1 \stackrel{s_1}{\Rightarrow} U_1)^\circ \stackrel{\text{def}}{=} (\overline{S_1^\circ} U_1^\bullet)^{!_{\mathrm{A}}}_{s_1} \leq (\overline{S_2^\circ} U_2^\bullet)^{!_{\mathrm{A}}}_{s_2} \stackrel{\text{def}}{=} (S_2 \stackrel{s_2}{\Rightarrow} U_2)^\circ$, as desired.

Case $\mathtt{ref}_s(S) \leq \mathtt{refr}_{s'}(S')$. Assume $s' \sqsubseteq s \quad S \leq S'$. Then by IH, $S^\circ \leq S'^\circ$, which implies $\overline{S'^\circ} \leq \overline{S^\circ}$. Then $\mathsf{r}_{s'}\langle \overline{S'^\circ} \rangle \leq \mathsf{rw}_s\langle \overline{S^\circ} \rangle$. Noting $\overline{\mathtt{ref}_s(S)^\circ} = \mathsf{rw}_s\langle \overline{S^\circ} \rangle$ and $\overline{\mathtt{refr}_{s'}(S')^\circ} = \mathsf{r}_{s'}\langle \overline{S'^\circ} \rangle$, we have $\mathtt{ref}_s(S)^\circ \leq \mathtt{refr}_{s'}(S')^\circ$, as required.

Case $\mathtt{refw}_s(S) \leq \mathtt{refw}_{s'}(S')$. Assume $s' \sqsubseteq s$ and $S' \leq S$. Then by IH, $S'^\circ \leq S^\circ$, which implies $\overline{S^\circ} \leq \overline{S'^\circ}$. Then $\mathsf{w}_{s'}\langle \overline{S'^\circ} \rangle \leq \mathsf{w}_s\langle \overline{S^\circ} \rangle$, which implies $\mathtt{refw}_s(S)^\circ \leq \mathtt{refw}_{s'}(S')^\circ$.

D.2 Proposition 7.13 (coincidence of protection/tampering levels)

By induction on the size of type T. For **(1)**, we have $\mathsf{tamp}(S^\bullet) = \mathsf{tamp}((S^\circ)^{\uparrow_{\mathrm{L}}}) = \mathsf{tamp}(S^\circ) = \mathsf{tamp}(\llcorner S \lrcorner_s^\circ)$ (Note $\llcorner S \lrcorner_s^\circ \stackrel{\text{def}}{=} S^\circ$). **(2)** is by simultaneous induction.

We use (1). we first prove the first part of the statement. Suppose $T = \mathbb{N}_s$. Then $\mathsf{protect}(\mathbb{N}_s) = s$. Also by (1), $\mathsf{tamp}(T^\bullet) = \mathsf{tamp}(\mathbb{N}_s^\circ) = \mathsf{tamp}([\oplus_\omega\,]_s^{\uparrow_L}) = s$. Similarly for the case $T = \llcorner S \lrcorner_s$. Next suppose $T = S \overset{s}{\Rightarrow} U$. Then we have:

$$
\begin{array}{rcll}
\mathsf{protect}(S \overset{s}{\Rightarrow} U) & = & \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(U) & \text{(by definition in Section 7.4)} \\
& = & \mathsf{tamp}(\overline{S^\circ}) \sqcap \mathsf{tamp}(U^\bullet) & \text{(by inductive hypothesis)} \\
& = & \mathsf{tamp}((\overline{S^\circ} U^\bullet)_s^{!_A}) & \text{(by Definition 6.1)} \\
& = & \mathsf{tamp}((S \overset{s}{\Rightarrow} U)^\circ) & \text{(by definition of }{}^\circ\text{)} \\
& = & \mathsf{tamp}((S \overset{s}{\Rightarrow} U)^\bullet) & \text{(by (1))}
\end{array}
$$

The cases $T = S \Rightarrow T$ is similar. If $T = \mathtt{ref}_s(S)$, we have:

$$
\begin{array}{rcll}
\mathsf{protect}(\mathtt{ref}_s(S)) & = & \mathsf{protect}^{\mathcal{E}}(S) \sqcap \mathsf{protect}(S) & \text{(by definition in Section 7.4)} \\
& = & \mathsf{tamp}(\overline{S^\circ}) \sqcap \mathsf{tamp}(S^\bullet) & \text{(by inductive hypothesis)} \\
& = & \mathsf{tamp}(\overline{S^\circ}) \sqcap \mathsf{tamp}(S^\circ) & \text{(by (1))} \\
& = & \mathsf{tamp}([(S^\circ)^{\uparrow_L} \& \overline{S^\circ}()^{\uparrow_L}]_s^{!_R}) & \text{(by definition of } \circ\text{)} \\
& = & \mathsf{tamp}(\mathtt{ref}_s(S)^\circ) & \\
& = & \mathsf{tamp}(\mathtt{ref}_s(S)^\bullet) & \text{(by (1))}
\end{array}
$$

Similarly for $T = \mathtt{refr}_s(S)$ and $T = \mathtt{refw}_s(S)$. For the proof the second part of (2), we only show the case that T is a reference type. The cases of $T = \mathtt{refw}_s(S)$ and $T = \mathtt{ref}_s(S)$ are obvious since $\mathsf{protect}^{\mathcal{E}}(\mathtt{ref}_s(S)) = \mathsf{protect}^{\mathcal{E}}(\mathtt{refw}_s(S)) = s = \mathsf{tamp}(\mathsf{rw}_s\langle S\rangle) = \mathsf{tamp}(\mathsf{w}_s\langle S\rangle)$. Suppose $T = \mathtt{refr}_s(S)$. Then we have:

$$
\begin{array}{rcll}
\mathsf{protect}^{\mathcal{E}}(\mathtt{refr}_s(S)) & = & \mathsf{protect}^{\mathcal{E}}(S) & \text{(by definition in Section 7.4)} \\
& = & \mathsf{tamp}(\overline{S^\circ}) & \text{(by inductive hypothesis)} \\
& = & \mathsf{tamp}(\mathsf{r}_s\langle\overline{S^\circ}\rangle) & \text{(by definition of the tampering level)} \\
& = & \mathsf{tamp}(\overline{\mathtt{refr}_s(S)^\circ}) & \text{(by definition of } \circ\text{)}
\end{array}
$$

(3) is by (2).

D.3 Proposition 7.16 (1) (well-typedness of the encoding)

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 $E \cdot x : S \vdash e : U$. By inductive hypothesis, we have $\vdash_{\mathsf{sec}} \langle e \rangle_u \triangleright E^\circ, x : S^\circ, u : U^\bullet$. By the condition $s \sqsubseteq \mathsf{tamp}(E) = \mathsf{tamp}(E^\circ)$ we know $s \sqsubseteq \mathsf{tamp}(E^\circ)$, hence by $(\mathsf{In}^{!_A})$:

$$\vdash_{\mathsf{sec}} !c(xm).\langle e\rangle_m \triangleright E^\circ, c : (\overline{S^\circ} U^\bullet)_s^{!_A} \tag{41}$$

Noting $(\overline{S^\circ} U^\bullet)_s^{!_A} = ((\overline{S^\circ} U^\bullet)_s^{!_A})^{\uparrow_L}$, an application of (Out) obtains:

$$\vdash_{\mathsf{sec}} \overline{u}(c)!c(xm).\langle e\rangle_m \triangleright E^\circ, u : (S \overset{s}{\Rightarrow} U)^\bullet,$$

as required.

Case *[LamT]*. Suppose in this rule, M has a total type T'. If $\langle e\rangle_m$ contains mutable free variables except x, then we cannot type $!c(xm).\langle e\rangle_m$ because in $(\mathsf{In}^{!_L})$ we only allow $?_L A$, $?_A B$ and $\mathsf{tamp}(B) = \mathtt{H}$ appears as free in its body. But this is guaranteed by the side condition of *[LamT]*, which says that for all y, $E(y) = S$ immutable (note then $\mathsf{md}(E(y)^\circ) \in \{?_L, ?_A\}$ and $\mathsf{tamp}(E(y)^\circ) = \mathtt{H}$).

Case *[Seq]*. By assumption, we know $\vdash_{\mathsf{sec}} [\![c_i]\!]_{u_i} \triangleright u_i : ()^{p_i}, A_i$ such that $A_i = E_i^\circ$ and $A_1 \odot A_2 = E_1^\circ \odot E_2^\circ$ 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:

$$\vdash_{\mathsf{sec}} (\boldsymbol{\nu}\, u_1)([\![c_1]\!]_{u_1} \mid u_1.[\![c_2]\!]_{u_2}) \triangleright u_2 : ()^{p_2}_{s_2}, E_1^\circ \odot E_2^\circ$$

Then by (Sub) and (Weak), we have $\vdash_{\mathsf{sec}} [\![c_1; c_2]\!]_u \triangleright u_2 : ()^{p_2}_{s_2}, E^\circ$, as required.

If on the other hand $\tau = \Uparrow$, then $p_{1,2} = \uparrow_{\mathrm{A}}$. By the side condition and by noting $\mathsf{tamp}(E_2) = \mathsf{tamp}(A_2)$, we have $s_1 \sqsubseteq s_2 \sqcap \mathsf{tamp}(E_2) = \mathsf{tamp}(u_2 : ()^{\uparrow_{\mathrm{A}}}_{s_2}, A_2)$. This satisfies the constraint of $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$. Thus the following is well-typed:

$$\vdash_{\mathsf{sec}} u_1.[\![c_2]\!]_{u_2} \triangleright u_2 : ()^{\uparrow_{\mathrm{A}}}_{s_2}, u_1 : ()^{\downarrow_{\mathrm{A}}}_{s_1}, A_2$$

Hence by (Par), (Sub) and (Res) we are done.

Case *[Sub]*. We first note that each sequent $E \vdash t : \alpha$ is translated into $\vdash [\![t]\!]_u : \alpha^\bullet, E^\circ$. Since E° only contains types with ?-modes, $E \le E'$ is directly proved by (Weak) and (Sub) by Proposition 7.12.

For $\alpha^\bullet \le \alpha'^\bullet$, the only non-trivial case is when we replace $\Downarrow_s$ with $\Uparrow_s$. We show this case by induction. Others are straightforward by (Weak), (Sub) and Proposition 7.12 for T again. The base cases ($c \stackrel{\text{def}}{=} \mathtt{skip}$ and $c \stackrel{\text{def}}{=} x := v$) 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]*. Assume $E \vdash z : \mathtt{refr}_s(T)$ and $E \cdot x : T \vdash c : \mathrm{cmd}\,\tau_{s_0}$. By induction hypothesis, we have, with $p \in \{\uparrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}\}$,

$$\vdash_{\mathsf{sec}} [\![c]\!]_u \triangleright E^\circ, x : \overline{T^\circ}, u : ()^p_{s_0}$$

Note $\overline{\mathtt{refr}_s(T)^\circ} = \mathsf{r}_s\langle\overline{T^\circ}\rangle$ and $\mathsf{r}_s\langle\overline{T^\circ}\rangle \le E^\circ(x)$. Hence we have:

$$\vdash_{\mathsf{sec}} \overline{z}\,\mathtt{read}(c)c(x).[\![c]\!]_u \triangleright E^\circ, u : ()^p_{s_0}$$

Finally noting $\mathrm{cmd}\,\tau_{s_0}{}^\bullet = ()^p_{s_0}$, we conclude the proof.

D.4 Proposition 7.16 (2) (computational adequacy)

Proposition D.1 (operational correspondence)

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

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

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

Proof. For (1-a), we first show $(\boldsymbol{\nu}\vec{x})(o,\sigma) \longrightarrow (\boldsymbol{\nu}\vec{x}')(o',\sigma')$ implies $(\boldsymbol{\nu}\,\vec{x})([\![o]\!]_{\vec{u}}|[\![\sigma]\!]) \longrightarrow\mapsto^+ (\boldsymbol{\nu}\,\vec{x}')([\![o']\!]_{\vec{u}}|[\![\sigma']\!])$ where $\mapsto$ is the extended reduction as given in [11; 74; 75]. 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 $\mathbf{!}_{\text{L}}$ or $\mathbf{!}_{\text{A}}$-type (hence $\mapsto$ satisfies CR [10; 74; 75]). Since $\mapsto$ stays within $\cong$, 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). $\square$

D.5 Theorem 7.18, Claim B

For the proof of the claim, we use the context $C[\,\cdot\,] \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,\vec{u})(S\langle\vec{u},z\rangle|[\,\cdot\,])$ 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 conjunctions w.l.o.g.).

$$S\langle\vec{u},z\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,\vec{x}y)(\Pi_i u_i.\overline{z}\,\mathtt{write}(Tc)c.\mathbf{0} \mid \Pi_i \mathsf{Ref}\langle x_i F\rangle \mid \mathsf{Ref}\langle yF\rangle \mid [\![\mathtt{while}\,\neg y\,\mathtt{do}\,y := \wedge_i x_i]\!]_z)$$

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 π^{LAR}) that $C[P] \Downarrow_z$ iff $P \twoheadrightarrow \Pi_i\overline{u_i}|R$.

D.6 Theorem 7.19 and Theorem 7.20

First, for general agent we have the following strengthened version of Proposition 7.16 (2).

Proposition D.2 (strengthened computational adequacy) *Assume $E \vdash o : cmd\,\tau_s$ is semi-closed and $E \vdash \sigma$. Then $(o,\sigma) \Downarrow (\boldsymbol{\nu}\,\vec{x})\sigma'$ iff $([\![o]\!]_{\vec{u}} \mid [\![\sigma]\!]) \longrightarrow^* \Pi_i\overline{u_i} \mid R \not\longrightarrow$ such that $R \cong [\![(\boldsymbol{\nu}\,\vec{x})\sigma']\!]$.*

Proof. We use the same two-way simulation as used for proving Proposition 7.16 (2). See Appendix D.3 (2). $\square$

The following corresponds to Claim A.

Proposition D.3 *$\vdash [\![(\boldsymbol{\nu}\,\vec{x}_1)\sigma_1]\!] \cong_s [\![(\boldsymbol{\nu}\,\vec{x}_2)\sigma_2]\!] \triangleright E^\circ$ implies $E \vdash (\boldsymbol{\nu}\,\vec{x}_1)\sigma_1 \cong_s (\boldsymbol{\nu}\,\vec{x}_2)\sigma_2$.*

Proof. By observing all stores and "contexts" (threads) for testing stores are encodable by processes, so that the latter can precisely simulate the behaviour of the former by Proposition D.2. $\square$

Theorem 7.19. *(non-interference for first-order store)* If $E \vdash o : \mathrm{cmd}\,\tau_s$, $E \vdash \sigma_1 \cong_s^{\text{vs}} \sigma_2$ and $E(x_i) = \mathbb{N}_{s_i}$ such that $s' \sqsubseteq s$, then $(o,\sigma_1) \Downarrow_{\vec{x}:\vec{n}}$ iff $(o,\sigma_2) \Downarrow_{\vec{x}:\vec{n}}$.

Proof. By precisely following the arguments in the proof of Theorem 7.18 except: (1) Claim A is replaced by Proposition D.3 and (2) Claim B is replaced by the following claim. Below we write $P \prec R$ when $P \cong R|S$ for some S.

Claim. Assume given $\vdash P \triangleright A$ such that $A \stackrel{\text{def}}{=} u_1 : ()^{\uparrow_A}, .., u_h : ()^{\uparrow_A}, x_1 : \mathbb{N}_{s_1}, .., x_m : \mathbb{N}_{s_m}, !_L B$ such that $s_i \sqsubseteq s$ for each i. Then, for each sequence of natural numbers $n_1, .., n_m$, there exists a well-typed context $C[\,\cdot\,]^{x:\tau}_A$ with $\tau = ()^{\uparrow_A}_s$ for which the following holds: $P \twoheadrightarrow (\Pi u_i)|R$ with $R \prec \Pi_i [\![n_i]\!]_{x_i}$ if and only if $C[P] \Downarrow_x$.

The proof follows that of Claim B for Theorem 7.18, interrogating each x_i after obtaining outputs at $u_1...u_h$. See Appendix D.3 (5) for the construction. The secrecy condition $s_i \sqsubseteq s$ is essential for the well-typed construction of the context. □

Theorem 7.20. *(sequential non-interference for generalised store)* In the sequential VS-calculus, if $E \vdash c : \text{cmd}\,\tau_s$ and $E \vdash \sigma_1 \cong^{\text{vs}}_s \sigma_2$ then $(c, \sigma_1) \Downarrow (\boldsymbol{\nu}\,\vec{x}_1)\sigma_1'$ implies $(c, \sigma_2) \Downarrow (\boldsymbol{\nu}\,\vec{x}_2)\sigma_2'$ such that $(\boldsymbol{\nu}\,\vec{x}_1)\sigma_1' \cong^{\text{vs}}_s (\boldsymbol{\nu}\,\vec{x}_2)\sigma_2'$.

Proof. For the sequential VS calculus, Proposition 7.16 (embeddability and soundness), Proposition D.3 (soundness of embedding of store), and Proposition D.2 (strengthened computational adequacy) automatically hold since the arguments do not rely on concurrency. We use the following claim (which roughly stands for Claim B in the proof of Theorem 7.18).

Claim. Assume given $\vdash P_{1,2} \triangleright A$ such that $A \stackrel{\text{def}}{=} u : ()^{\uparrow_A}_s, !_L B$ and $P_i \twoheadrightarrow \overline{u}|R_i$ $(i = 1, 2)$. Then there exists a well-typed context $C[\,\cdot\,]^{x:()^{\uparrow_A}_s}_A$ for which the following holds: $P_2 \twoheadrightarrow \overline{u}|R_2$ such that $R_1 \cong_s R_2$ iff $C[P_1] \Downarrow_u \;\Leftrightarrow\; C[P_2] \Downarrow_u$.

The context simply waits for an output at x, then interacts with the resulting process as a context which differentiates R_1 and R_2 (we only have to consider the case when both converge because, by precisely the same argument as before, we already know the convergence/divergence of two configurations coincide. We now reason as follows. Assume $(c, \sigma_i) \Downarrow (\boldsymbol{\nu}\,\vec{x})\sigma_i'$ $(i = 1, 2)$.

$$
\begin{array}{lll}
\sigma_1 \cong^{\text{vs}}_s \sigma_2 & \Rightarrow & [\![\sigma_1]\!] \cong_s [\![\sigma_2]\!] \hfill \text{(Claim A)} \\
 & \Rightarrow & [\![c]\!]_{\vec{u}} \,|\, [\![\sigma_1]\!] \cong_s [\![c]\!]_{\vec{u}} \,|\, [\![\sigma_2]\!] \hfill \text{(congruency)} \\
 & \Rightarrow & \exists C[\,\cdot\,].\ (C[[\![c]\!]_u \,|\, [\![\sigma_i]\!]] \Downarrow_u \Leftrightarrow\ [\![c]\!]_u \,|\, [\![\sigma_i]\!] \twoheadrightarrow (\Pi\overline{u})|R_i) \\
 & & R_1 \cong_s R_2 \hfill \text{(Claim)} \\
 & \Rightarrow & (\boldsymbol{\nu}\,\vec{x}_1)\sigma_1 \cong^{\text{vs}}_s (\boldsymbol{\nu}\,\vec{x}_2)\sigma_2 \hfill \text{(Pro. D.3)}
\end{array}
$$

as required. □

E. SYNTAX, REDUCTION AND SECRECY TYPING RULES FOR THE FULL CALCULUS

This section summarises the full calculus for reference.

(Syntax)

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

(Structural Rules)

$$P \equiv Q \quad \text{if } P \equiv_\alpha Q \qquad P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P$$
$$P|(Q|R) \equiv (P|Q)|R \qquad (\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P \equiv (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P$$
$$(\boldsymbol{\nu}\, x)(P|Q) \equiv ((\boldsymbol{\nu}\, x)P)|Q \quad (x \notin \mathsf{fn}(Q))$$

(Reduction)

$$x(\vec{y}).P \mid \overline{x}\langle\vec{v}\rangle \longrightarrow P\{\vec{v}/\vec{y}\} \qquad !\, x(\vec{y}).P \mid \overline{x}\langle\vec{v}\rangle \longrightarrow !\, x(\vec{y}).P|P\{\vec{v}/\vec{y}\}$$
$$P \longrightarrow P' \implies P|Q \longrightarrow P'|Q \qquad P \longrightarrow Q \implies (\boldsymbol{\nu}\, x)P \longrightarrow (\boldsymbol{\nu}\, x)Q$$
$$P \equiv P' \longrightarrow Q' \equiv Q \implies P \longrightarrow Q$$

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

$$\mathsf{Ref}\langle xv\rangle \mid \overline{x}\,\mathtt{read}\langle c\rangle \longrightarrow \mathsf{Ref}\langle xv\rangle \mid \overline{c}\langle v\rangle \qquad \mathsf{Ref}\langle xv\rangle \mid \overline{x}\,\mathtt{write}\langle v'c\rangle \longrightarrow \mathsf{Ref}\langle xv'\rangle \mid \overline{c}$$

where `read` and `write` means `inl` and `inr`, respectively.

(Subtyping)

$$\frac{\tau_i \leq \tau'_i}{(\vec{\tau})^p \leq (\vec{\tau}')^p} \qquad \frac{p_{\mathrm{O}} \neq {\uparrow_{\mathrm{L}}} \quad \tau_i \leq \tau'_i \quad s \sqsubseteq s'}{(\vec{\tau})^{p_{\mathrm{O}}}_s \leq (\vec{\tau}')^{p_{\mathrm{O}}}_{s'}} \qquad \frac{\tau_{ij} \leq \tau'_{ij} \quad s \sqsubseteq s'}{[\oplus_i \vec{\tau}_i]^{p_{\mathrm{O}}}_s \leq [\oplus_i \vec{\tau}'_i]^{p_{\mathrm{O}}}_{s'}} \qquad \frac{\overline{\tau_{\mathrm{I}}'} \leq \overline{\tau_{\mathrm{I}}}}{\tau_{\mathrm{I}} \leq \tau'_{\mathrm{I}}}$$

$$\frac{\tau \leq \tau' \quad s \sqsubseteq s'}{\mathsf{r}_s\langle\tau\rangle \leq \mathsf{r}_{s'}\langle\tau'\rangle} \qquad \frac{\tau' \leq \tau \quad s \sqsubseteq s'}{\mathsf{w}_s\langle\tau\rangle \leq \mathsf{w}_{s'}\langle\tau'\rangle} \qquad \frac{\tau \leq \tau' \quad s \sqsubseteq s'}{\mathsf{r}_s\langle\tau\rangle \leq \mathsf{rw}_{s'}\langle\tau'\rangle} \qquad \frac{\tau' \leq \tau \quad s \sqsubseteq s'}{\mathsf{w}_s\langle\tau\rangle \leq \mathsf{rw}_{s'}\langle\tau'\rangle} \qquad \frac{s \sqsubseteq s'}{\mathsf{rw}_s\langle\tau\rangle \leq \mathsf{rw}_{s'}\langle\tau\rangle}$$

(Typing System)

(Zero)
$$\frac{-}{\vdash_{\mathsf{sec}} \mathbf{0} \triangleright \_}$$

(Par)
$$\frac{\vdash_{\mathsf{sec}} P_i \triangleright A_i \quad (i=1,2) \qquad A_1 \asymp A_2}{\vdash_{\mathsf{sec}} P_1|P_2 \triangleright A_1 \odot A_2}$$

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

(Weak)
$$\frac{\vdash_{\mathsf{sec}} P \triangleright A^{\text{-}x} \qquad \mathsf{md}(\tau) \in \mathcal{M}_{?} \cup \{\updownarrow\}}{\vdash_{\mathsf{sec}} P \triangleright A, x : \tau}$$

(In$^{\downarrow_{\mathrm{L}}}$)
$$\frac{\vdash_{\mathsf{sec}} P \triangleright \vec{y} : \vec{\tau}, {\uparrow_{\mathrm{L}}} A^{\text{-}x}, {\uparrow_{\mathrm{A}}} \mathbf{?} B^{\text{-}x}}{\vdash_{\mathsf{sec}} x(\vec{y}).P \triangleright (x : (\vec{\tau})^{\downarrow_{\mathrm{L}}} \rightarrow A), B}$$

(In$^{!_{\mathrm{L}}}$) $s \sqsubseteq \mathsf{tamp}(A, B)$
$$\frac{\vdash_{\mathsf{sec}} P \triangleright \vec{y} : \vec{\tau}, \mathbf{?}_{\mathrm{L}} A^{\text{-}x}, \mathbf{?}_{\mathrm{A}} B^{\text{-}x}}{\vdash_{\mathsf{sec}} !\, x(\vec{y}).P \triangleright (x : (\vec{\tau})^{!_{\mathrm{L}}}_s \rightarrow A), B}$$

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

(In$^{!_{\mathrm{A}}}$) $s \sqsubseteq \mathsf{tamp}(A)$
$$\frac{\vdash_{\mathsf{sec}} P \triangleright \vec{y} : \vec{\tau}, \mathbf{?} A^{\text{-}x}}{\vdash_{\mathsf{sec}} !\, x(\vec{y}).P \triangleright x : (\vec{\tau})^{!_{\mathrm{A}}}_s, A}$$

(Out)
$$\frac{-}{\vdash \overline{x}\langle\vec{y}\rangle \triangleright x : (\vec{\tau})^{p_{\mathrm{O}}}_s, \vec{y} : \overline{\vec{\tau}}}$$

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

(Bra$^{!_{\mathrm{L}}}$) $s \sqsubseteq \mathsf{tamp}(A, B)$
$$\frac{\vdash_{\mathsf{sec}} P \triangleright \vec{y}_i : \vec{\tau}_i, \mathbf{?}_{\mathrm{L}} A^{\text{-}x}, \mathbf{?}_{\mathrm{A}} B^{\text{-}x}}{\vdash_{\mathsf{sec}} !\, x[\&_i(\vec{y}_i).P_i] \triangleright (x : [\&_i \vec{\tau}_i]^{!_{\mathrm{L}}}_s \rightarrow A), B}$$

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

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

(Sel)
$$\frac{-}{\vdash \overline{x}\mathtt{in}_i\langle\vec{y}\rangle \triangleright x : [\oplus_i \vec{\tau}_i]^{p_{\mathrm{O}}}_s, \vec{y} : \overline{\vec{\tau}_i}}$$

(Sub)
$$\frac{\vdash_{\mathsf{sec}} P \triangleright A \quad A \leq A'}{\vdash_{\mathsf{sec}} P \triangleright A'}$$

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

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

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

Fig. 19. Summary of Secrecy Subtyping and Typing Rules for π^{LAR}

Index