A Uniform Type Structure for Secure Information Flow

Kohei Honda
Deptartment of Computer Science
Queen Mary
University of London, UK
kohei@dcs.qmw.ac.uk

Nobuko Yoshida
Department of Mathematics and
Computer Science
University of Leicester, UK.
ny11@mcs.le.ac.uk

ABSTRACT

The π-calculus is a formalism of computing in which we can compositionally represent dynamics of major programming constructs by decomposing them into a single communication primitive, the name passing. This work reports our experience in using a linear/affine typed π-calculus for the analysis and development of type systems of 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 by the development of a novel type discipline for imperative programs which extends both a secure multi-threaded imperative language by Smith and Volpano and (a call-by-value version of) DCC. In each case, the embedding gives a simple proof of noninterference.

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, which now have both well-developed theories and an increasing number of applications. Can we use types to describe, reason about and control the behaviour of such an aggregation? For this to be effective, it should be possible to type-check one component with a specific type, say $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$ (where $\mathbb{N}$ is a type for a natural number and $\Rightarrow$ is a function type constructor), and combine it with other parts, which may have different type structures, with a guarantee that it behaves as decreed by the original type discipline. For example, if $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$ is inferred in a strongly normalising type discipline, 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.

POPL '02, Jan. 16-18, 2002 Portland, OR USA

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 [36]. 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 using a single format.

The π-Calculus. The π-calculus [26, 25, 7, 17] is an extension of CCS based on name passing. A basic form of its dynamics can be written down as the following reduction.

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

Here a vector of names $\vec{w}$ are communicated, via x, to an input process, resulting in name instantiation. Perhaps surprisingly, this single operation can compositionally represent dynamics of diverse language constructs, including function application, sequencing, assignment, exception, object, not to speak of communication and concurrency. We are thus prompted by the following question: can we have a foundational type structure for this calculus, similar to those for the 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. [25, 24, 30, 39], 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 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 [25, 30, 15, 39, 24] and game semantics [3, 4, 23, 20], the present authors, with Martin Berger, recently reported [6, 40] that basic type structures for the π-calculus which precisely capture existing type structure do exist, allowing fully abstract translation of prominent functional typed calculi. In [6, 40], 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, [6, 40] have shown PCF and strongly normalising λ-calculi are fully abstractly embeddable in the affine and linear π-calculus, respectively. In spite of faithfulness in embeddings, the form of types is quite different from that of function types, articulating a broader realm of typed behaviour. In particular, both call-by-value and call-by-name λ-calculi are embeddable into a single typing system by changing translation of types.

Secure Information Flow. The present paper reports how we can apply the linear and affine type structures of the π-calculus, as proposed in [6, 40], for the study of type disciplines of programming languages, taking type-based analysis of secure information flow [2, 12, 29, 33, 34, 35, 38] as an application domain. In this analysis, we use a typing system to ensure the safety of information flow in a given program, i.e. a high-level (secure) data never flows down to low-level (public) channels. Information flow analysis needs precise understanding of observable behaviour of program phrases and their interplay, because of the existence of covert channels [9]. In the π-calculus representation, computational dynamics is decomposed into interaction, where the notion of observables is made explicit. This makes the π-calculus a potentially effective tool for analysing subtle information flow among program phrases. Further, in many type-based information flow analysis, distinction between totality and partiality is crucial, both in functional [2] and imperative [38] settings, strongly suggesting its connection to linear/affine type structures. A uniform treatment of call-by-name and call-by-value pure functions as well as stateful computation in secrecy 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 basic noninterference property.
- The embeddability of the dependency core calculus (DCC) [2] in the secrecy-enhanced linear/affine π-calculus, and a simple operational proof of its noninterference property. We also present a novel call-by-value version of DCC.
- A new type system for secrecy in concurrent imperative programs with references and higher-order procedures. Its embeddability in the linear/affine π-calculus with state again gives a simple proof of non-interference.

A picture of typed calculi used in this text is given in Figure 1. Each box represents a name of the typed π-calculus with a specific type structure ("L", "A" and "μ" mean linear, affine and state, respectively). The right-hand side of the box

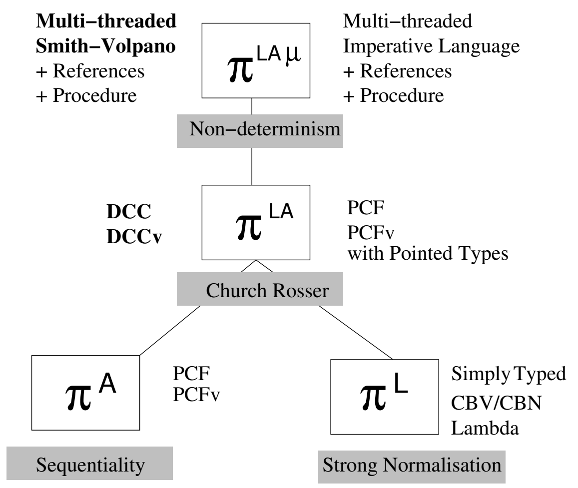

Figure 1: A Family of Linear/Affine π-Calculi

shows systems we can embed in the basic typed π-calculus. The left-hand side shows the secure languages we can embed in the secure version of the π-calculus. The grey box shows a basic property satisfied by the calculus.

Related Work. There are a few prominent examples of integrated function-based type disciplines, which often use monads. Basic examples include pointed types [21, 27] and the incorporation of imperative constructs in Haskell [22].

The dependency core calculus [2] is a powerful functional metalanguage for secrecy, using pointed types. 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.

Smith and Volpano (cf.[34, 35, 38]) studied various aspects of secrecy in imperative languages. Sequential procedures are studied in [38]. Multi-threading is studied in [35], whose typability was enlarged by our work with Vasconcelos [18] using the π-calculus, based on which a further enhancement was done in [34] (cf.[8]). A proper extension of the concurrent language of [35, 34] integrating higher-order procedures and general references would be new.

A recent work [42] presents a typed control calculus with references, intended as a meta-language 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, [42] does not treat multi-threading, and is not (intended as) an extension of the language in [35, 34].

Linearity has been studied in diverse forms, both for functions, cf. [11] and for processes, cf. [15, 24, 39]. Clear behavioural articulation and characterisation of linearity and affinity as typed processes are first presented in [6, 40], one of whose initial applications is reported here.

Secrecy and other security issues in processes are widely studied recently, cf. [1, 31, 10, 13, 18]. [1] includes insightful discussions on secrecy. These studies focus on modelling security concerns in distributed systems, and do not pursue integrated secrecy typing for language constructs.

Outline. Section 2 introduces the secrecy π-calculus based on linear/affine type disciplines. Section 3 embeds DCC and its call-by-version in the calculus. Section 4 presents a stateful extension of the calculus in Section 2. Section 5 presents applications to imperative secrecy.

Acknowledgements. We thank anonymous referees for helpful comments and Martin Berger for our ongoing collaboration. The first author is partially supported by EPSRC grant GR/N/37633. The second author is partially supported by EPSRC grant GR/R33465/01.

2. SECURE LINEAR/AFFINE TYPES

2.1 Processes

Following [6, 40], we use the asynchronous version of the π-calculus [7, 17] with bound output [32] and branching [14, 16, 18]. Let $x, y, \ldots$ and sometimes $a, b, \ldots$ range over a countable set of names (also called channels). The set of untyped terms, which we often call *processes*, is given by the following grammar.

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

In $!P$ we require P to be an input or a branching. The bound/free names are defined as usual. Here and henceforth we assume names in a vector $\vec{y}$ are pairwise distinct. Up to the structural equality, whose definition is given in [6, 40], the output $\overline{x}(\vec{y})Q$ acts as $(\boldsymbol{\nu}\,\vec{y})(\overline{x}\langle\vec{y}\rangle|Q)$ in the standard syntax. Bound name passing has essentially equivalent expressive power as free name passing [32], and is convenient for obtaining precise correspondence with functional type structures [6, 40]. We assume the branching allows for any countable indexing set with cardinality more than one. Branching is used for representing base values as well as conditionals [6, 40], and plays an essential rôle in the information flow analysis later.

The reduction relation is generated by the following rules, closing under parallel composition, restriction and output, taking processes modulo $\equiv$.

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

and, for branching,

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

The multi-step reduction $\twoheadrightarrow$ is given as: $\twoheadrightarrow \stackrel{\text{def}}{=} \equiv \cup \longrightarrow^*$. As a simple example of processes, $[\![n]\!]_u \stackrel{\text{def}}{=} !u(c).\overline{c}\mathtt{in}_n$ is a *natural number agent*, which acts as a server that necessarily returns a fixed answer, n.

2.2 Action modes and channel types

The type structure we shall use combines affine types [6] and linear types [40], and is enhanced by secrecy. First we introduce *action modes* [6, 15, 18, 40], which prescribe different modes of interaction at each channel.

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

We also use the mode $*$ to indicate uncomposability. $p, p', \ldots$ range over action modes. The modes in the left column are *input modes* while those in the right are *output modes*. The pair of modes in each row are *dual* to each other, writing $\overline{p}$ for the dual of p. We set $\mathcal{M}_{\downarrow} = \{\downarrow_{\text{L}}, \downarrow_{\text{A}}\}$, similarly for $\mathcal{M}_!$, $\mathcal{M}_?$ etc. The L-modes correspond to linear modes in [40] while the A-modes to affine modes in [6]. The difference between linearity and affinity in non-replicated channels is that, in a linear channel, an interaction takes place *precisely once*, while it does so *at most once* in an affine channel.

Fix a complete lattice $(\mathcal{L}, \sqsubseteq, \top, \bot)$ of secrecy levels (higher means more secure), whose elements are written $s, s', \ldots$. Then *channel types* are given by the following grammar. Below p_{I} (resp. p_{O}) denotes input (resp. output) modes.

$$\tau \;::=\; \tau_{\text{I}} \mid \tau_{\text{O}} \mid * \qquad \begin{array}{rcl} \tau_{\text{I}} & ::= & (\vec{\tau})^{p_{\text{I}}}_s \;\mid\; [\&_i\vec{\tau}_i]^{p_{\text{I}}}_s \\ \tau_{\text{O}} & ::= & (\vec{\tau})^{p_{\text{O}}}_s \;\mid\; [\oplus_i\vec{\tau}_i]^{p_{\text{O}}}_s \end{array}$$

$(\vec{\tau})^p$ is called *input/output unary type*, $[\&_i\vec{\tau}_i]^p$ and $[\oplus_i\vec{\tau}_i]^p$ *branching/selection type*. We write $\mathsf{sec}(\tau)$ for the outermost secrecy level of τ; $\mathsf{sec}(*) = \top$. $\overline{\tau}$ is the *dual* of τ by dualising all action modes and exchanging & and $\oplus$. The *mode* of τ, denoted $\mathsf{md}(\tau)$, is $*$ if $\tau = *$, else its outermost mode.

On types we define $\odot$ as the least commutative partial operation s.t. (1) $\tau \odot \overline{\tau} = *$ $(\mathsf{md}(\tau) \in \mathcal{M}_{\downarrow})$, (2) $\tau \odot \overline{\tau} = \tau$ $(\mathsf{md}(\tau) \in \mathcal{M}_!)$ and (3) $\tau \odot \tau = \tau$ $(\mathsf{md}(\tau) \in \mathcal{M}_?)$. Intuitively, (1) says once we compose input-output linear/affine channels, the channel becomes uncomposable; while (2) and (3) together say that a server should be unique, to which an arbitrary number of clients can request interactions. Note the composition between affine and linear types is prohibited; for example, $()^{\downarrow_{\text{L}}}_s \odot ()^{\uparrow_{\text{A}}}_s$ is undefined, while $()^{\downarrow_{\text{L}}}_s \odot ()^{\uparrow_{\text{L}}}_s = *$. We also assume the sequential constraint on channel types as given in [6, 40], which we list in Appendix A.

2.3 Action types

Following [6, 40], we use an *action type* which is essentially an assignment of channel types to free names in a process together with causality information. Formally an action type is a finite directed acyclic graph such that:

(G1) Each node has form $x : \tau$. No two nodes in the graph have identical subjects.

(G2) Each edge has form $x : \tau \to x' : \tau'$ with either $\mathsf{md}(\tau) = \downarrow_{\text{L}}$ and $\mathsf{md}(\tau') = \uparrow_{\text{L}}$, or $\mathsf{md}(\tau) = !_{\text{L}}$ and $\mathsf{md}(\tau') = ?_{\text{L}}$.

Note, in (G2) we do not record dependency for affine channels. This is because we permit circularity, hence divergence, for affine channels. $A, B, \ldots$ range over action types. A node (or its name) is *active* in A if it has no incoming edges.

The partial operator $A \odot B$ is defined iff channel types in common names compose by $\odot$ given above, and, moreover, the adjoined graph do not have a cycle. If so, the result is a graph in which, in the adjoined graph, each maximal causal chain is collapsed into an edge connecting its two ends (see [40] for a formal definition). To avoid divergence at linear channels, this operator ensures that processes never exhibit circular dependency in actions. For example, $x : \tau_1 \to y : \tau_2$ and $y : \overline{\tau_2} \to x : \overline{\tau_1}$ are not composable. We write $|A|$, $\mathsf{md}(A)$, $\mathsf{fn}(A)$ and $\mathsf{active}(A)$ for the sets of nodes, modes, names and

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

$$\text{(Weak)}\ \frac{\vdash P \triangleright A^{\text{-}x} \qquad \mathsf{md}(\tau) \in \mathcal{M}_{*,?}}{\vdash P \triangleright A \otimes x:\tau} \qquad \text{(Out)}\ (p \in \mathcal{M}_{\uparrow,?})\ \frac{\vdash P \triangleright C\langle \vec{y}:\vec{\tau}\rangle \qquad C/\vec{y} = A \asymp x:(\vec{\tau})^p_s}{\vdash \overline{x}(\vec{y})P \triangleright A \odot x:(\vec{\tau})^p_s}$$

$$\text{(In}^{\downarrow_{\mathrm{L}}}\text{)}\ \frac{\vdash P \triangleright \vec{y}:\vec{\tau} \otimes \uparrow_{\mathrm{L}} A^{\text{-}x} \otimes \uparrow_{\mathrm{A}} B^{\text{-}x} \otimes ?C^{\text{-}x}}{\vdash x(\vec{y}).P \triangleright (x:(\vec{\tau})^{\downarrow_{\mathrm{L}}}_s \to A) \otimes B \otimes C} \qquad \text{(In}^{\downarrow_{\mathrm{A}}}\text{)}\ s \sqsubseteq \mathsf{tamp}(B)\ \frac{\vdash P \triangleright \vec{y}:\vec{\tau} \otimes \uparrow_{\mathrm{A}} ?B^{\text{-}x}}{\vdash x(\vec{y}).P \triangleright x:(\vec{\tau})^{\downarrow_{\mathrm{A}}}_s \otimes B}$$

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

Figure 2: Linear/Affine Secrecy Typing Rules

active names in A, respectively. Further notations:

$A\langle \vec{y}:\vec{\tau}\rangle$	$y_i:\tau_i$ occurs in A.
$A/\vec{x}$	the result of taking off $x_i:\tau_i$ in A
$\vec{p}A$	A such that $\mathsf{md}(A) = \{\vec{p}\}$
$?A$	A such that $\mathsf{md}(A) \subset \mathcal{M}_?$
$A^{\text{-}x}$	A such that $x \notin \mathsf{fn}(A)$
$A \otimes B$	a disjoint union of A and B s.t. $\mathsf{fn}(A) \cap \mathsf{fn}(B) = \emptyset$.

Finally $x:\tau \to A$ adds edges from $x:\tau$ to A's active nodes.

2.4 Tamper level

A *tamper level* indicates a lower bound of effects the process would have on its environment. It is first defined on types, and is lifted to action types. In its definition, modes of actions play an important role. Below we say τ is *immediately tampering* if either $\tau = [\oplus_i \vec{\tau}_i]^{\uparrow_{\mathrm{L}}}$ or $\mathsf{md}(\tau) = \uparrow_{\mathrm{A}}$.

Definition 2.1. $\mathsf{tamp}(\tau)$ is inductively given by:

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

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

As an illustration, let $\mathbb{N}^{\circ}_s \stackrel{\text{def}}{=} ([\oplus_{i\in\omega}]^{\uparrow_{\mathrm{L}}}_s)^{!_{\mathrm{L}}}$ and consider $[\![2]\!]_x \stackrel{\text{def}}{=} !x(c).\overline{c}\mathtt{in}_2$, which emits "2" after getting invoked and which has type $\mathbb{N}^{\circ}_s$ at x (in the typing system we introduce below). This process does contain information, but it only comes after a replicated input, which itself does not emit information. Thus its observable information is located at the linear selection. Similarly, with x typed as $\tau = (()^{\uparrow_{\mathrm{A}}}_s)^{!_{\mathrm{A}}}$, $!x(c).\overline{c}$ has information at c since output at c may not come out (in fact, Ω^{τ}_x in Example 2.2 (4) later has the same type). $?_{\mathrm{L}}/?_{\mathrm{A}}$ actions do not tamper since they only touch stateless replication. For a further account, see [41].

2.5 Typing

The sequent has the form $\vdash P \triangleright A$, which we read: *$P$ is typable by A*. The typing rules are given in Figure 2, where in $(\mathrm{In}^{\downarrow_{\mathrm{L}}})$ we stipulate $|A \otimes B|$ is at most a singleton (this condition corresponds to the condition for sequentialisation used in [40], and is currently used in the proof of noninterference). (Par) uses $\asymp$ and $\odot$ for controlling composition. (Res) allows hiding of a name only when its action mode is $*$ or $!$ (which intuitively says channels of modes $\uparrow$, $\downarrow$ or $?$ are always compensated by their duals before restricted). For prefix, Figure 2 lists the unary rules (for the branching rules see Appendix A). Among them, only $(\mathrm{In}^{\downarrow_{\mathrm{A}}})$ uses a secrecy level non-trivially. Intuitively it says that if a process receives non-trivial information at s, then it should not transmit this effect to the levels lower than s. Other unary prefixes do not directly receive information, hence are not constrained by secrecy levels. We also observe that input never suppresses input, $!_{\mathrm{L}}$ and $!_{\mathrm{A}}$ never suppress $\uparrow_{\mathrm{L}}$ or $\uparrow_{\mathrm{A}}$, $\downarrow_{\mathrm{A}}$ never suppresses $\uparrow_{\mathrm{L}}$, and that $\downarrow_{\mathrm{L}}$ may suppress $\uparrow_{\mathrm{L}}$ and $\uparrow_{\mathrm{A}}$ (the last two points are crucial for integrating affinity into linearity consistently). Also note the outermost secrecy levels of $!_{\mathrm{L}}$, $!_{\mathrm{A}}$ and unary $\downarrow_{\mathrm{L}}$-types, as well as their duals, are irrelevant in typing, which we shall often omit from now on.

The resulting typed calculus, which we hereafter call $\pi^{\mathsf{LA}}_{\mathrm{sec}}$, satisfies subject reduction, and inherits behavioural properties from the systems in [6] and [40], including liveness at linear channels. Some examples of typed terms follow.

Example 2.2. 1. $\vdash [\![n]\!]_x \triangleright x:\mathbb{N}^{\circ}_s$.

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

3. (copy-cat) Let $[x \to x']^{\tau_{\mathrm{I}}}$ be given by: $[x \to x']^{[\&_i \vec{\tau}_i]^p_s} \stackrel{\text{def}}{=} x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i(\vec{y'}_i)\Pi_{ij}[y'_{ij} \to y_{ij}]^{\overline{\tau_{ij}}}]$ $(p \in \mathcal{M}_{\downarrow})$ and $[x \to x']^{[\&_i \vec{\tau}_i]^p_s} \stackrel{\text{def}}{=} !x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i(\vec{y'}_i)\Pi_{ij}[y'_{ij} \to y_{ij}]^{\overline{\tau_{ij}}}]$ $(p \in \mathcal{M}_!)$, similarly for the unary cases. This is the *copy-cat agent*, linking two locations, x and x'.

4. (omega) $\Omega^{\tau}_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)([u \to y]^{\tau} | [y \to u]^{\tau})$ immediately diverges after the initial input. If $\tau \stackrel{\text{def}}{=} ([\oplus_{i\in\omega}\,]^{\uparrow_{\mathrm{A}}}_s)^{!_{\mathrm{A}}}$, then Ω^{τ}_u is typable but if $\tau = \mathbb{N}^{\circ}_s$, it is untypable.

In $\pi^{\mathsf{LA}}_{\mathrm{sec}}$, we can naturally define a contextual congruence relativised by secrecy levels. Write $P^A \Downarrow^s_x$ when $P \longrightarrow^* P'$ s.t. $\mathsf{sec}(A(x)) \sqsubseteq s$ and either $P' \equiv \overline{x}(\vec{u})P''$ or $P' \equiv \overline{x}\mathtt{in}_i(\vec{u})P''$ (that is, P has an action observable at level s). We then say a typed congruence $\cong$ is *s-sound* when (1) it is reduction-closed [19], i.e. $\vdash P_1 \cong P_2 \triangleright A$ and $P_1 \twoheadrightarrow P'_1$ implies $P_2 \twoheadrightarrow P'_2$ with $\vdash P'_1 \cong P'_2 \triangleright A$, and (2) it respects $\Downarrow^s_x$, i.e. if $\vdash P_1 \cong P_2 \triangleright x:\tau$ s.t. $\mathsf{md}(\tau) = \uparrow_{\mathrm{A}}$, then $P_1^{x:\tau} \Downarrow^s_x$ implies $P_2^{x:\tau} \Downarrow^s_x$. The maximum s-sound congruence is denoted $\cong^{\pi}_s$. Using $\cong^{\pi}_s$, we can state a basic property of $\pi^{\mathsf{LA}}_{\mathrm{sec}}$, underpinning its theory and applications. The proof uses a secrecy-sensitive bisimulation, see [41].

Proposition 2.3. (non-interference) *If* $\vdash P_{1,2} \triangleright A$ *s.t.* $\mathsf{tamp}(A) = s$ *and* $s \not\sqsubseteq s'$, *then* $\vdash P_1 \cong^{\pi}_{s'} P_2 \triangleright A$.

$$
\begin{array}{llll}
[\mathit{Var}] & \Gamma, x{:}T \vdash x : T & [\mathit{Unit}] & \Gamma \vdash () : \mathtt{unit} \\[2ex]
[\mathit{Lam}] & \dfrac{\Gamma, x{:}T \vdash M : T'}{\Gamma \vdash \lambda x{:}T.M : T \Rightarrow T'} & [\mathit{App}] & \dfrac{\Gamma \vdash M : T \Rightarrow T' \quad \Gamma \vdash N : T}{\Gamma \vdash MN : T'} \\[2ex]
[\mathit{Inl}] & \dfrac{\Gamma \vdash M : T_1}{\Gamma \vdash \mathtt{inl}(M) : T_1 + T_2} & [\mathit{Case}] & \dfrac{\Gamma \vdash M : T_1 + T_2 \quad \Gamma, x{:}T_i \vdash M_i : T}{\Gamma \vdash \mathtt{case}\ M\ \mathtt{of}\ \mathtt{inl}(x)M_1\ \mathtt{or}\ \mathtt{inr}(x)M_2 : T} \\[2ex]
[\mathit{UnitM}] & \dfrac{\Gamma \vdash M : T}{\Gamma \vdash M : (T)_s} & [\mathit{BindM}] & \dfrac{\Gamma \vdash N : (T)_s \quad \Gamma, x{:}T \vdash M : T'}{\Gamma \vdash \mathtt{bind}\ x = N\ \mathtt{in}\ M : T'}\ \ s \sqsubseteq \mathsf{protect}(T') \\[2ex]
[\mathit{Lift}] & \dfrac{\Gamma \vdash M : T}{\Gamma \vdash \mathtt{lift}(M) : \llcorner T \lrcorner} & [\mathit{Seq}] & \dfrac{\Gamma \vdash N : \llcorner T \lrcorner \quad \Gamma, x{:}T \vdash M : T'}{\Gamma \vdash \mathtt{seq}\ x = N\ \mathtt{in}\ M : T'}\ \ T'\ \text{pointed} \\[2ex]
[\mathit{Rec}] & \dfrac{\Gamma, x{:}T \vdash M : T}{\Gamma \vdash \mu x{:}T.M : T}\ \ T\ \text{pointed} & &
\end{array}
$$

Figure 3: Dependency Core Calculus

3. SECRECY IN PURE FUNCTIONS

3.1 Dependency core calculus

The dependency core calculus [2] (DCC) is interesting in the present context at least in two ways. First it is one of the effective examples of a functional meta-language for type-based information flow analysis. Second it crucially relies on pointed types to combine total function types and partial ones [21, 27]. After outlining DCC in this subsection, we show a faithful embedding of DCC in $\pi^{\mathsf{LA}}_{\mathrm{sec}}$, leading to a new proof of its non-interference.

We use a slightly different, but equivalent, presentation of DCC. This is to allow a simpler presentation of the embedding. In particular the lifting associated with secrecy is used implicitly. We omit products for brevity (their incorporation poses no technical difficulty). The set of DCC-types are given by the following grammar. We use the same lattice $\mathcal{L}$ of secrecy levels.

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

We omit s if $s = \bot$. Unit, sums and function types should be familiar. The lifted type $\llcorner T \lrcorner_s$ is a so-called pointed type, which denotes potential divergence. The level s in $(T)_s$ indicates a secrecy level which protects a datum. We consider types modulo the following equations (which come from isomorphisms in the denotational universe in [2]).

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

By reading the above equations from left to right, we can rewrite types to simpler forms. In fact each type has a unique normal form, which has shape $T_1 \Rightarrow (T_2 \Rightarrow (\ldots (T_{n-1} \Rightarrow \gamma) \ldots)$ with $n \geq 1$, where γ is given by the grammar (with $T, T_{1,2}$ being normal forms again):

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

We write $[T_1 T_2 ... T_{n-1}\gamma]$ for $T_1 \Rightarrow (T_2 \Rightarrow (\ldots (T_{n-1} \Rightarrow \gamma) \ldots)$. Two key ideas in the DCC-types:

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

We can check that T is protected at s in the sense of [2] iff $s \sqsubseteq \mathsf{protect}(T)$. Similarly the notion of pointedness coincides with [2].

Figure 3 presents the typing rules of DCC. The sequent has form $\Gamma \vdash M : T$ where M is a λ-preterm with units, sums and recursion. We can check the typability coincides with the system in [2] up to the erasure of type annotations. The β-reduction is defined in the standard way (with $\mathtt{seq}\, x = \mathtt{lift}(N)\, \mathtt{in}\, M \longrightarrow_\beta M\{N/x\}$), for which we can easily verify the subject reduction property.[1]

We conclude the presentation by stipulating a Morris-like contextual congruence on DCC-terms, relativised by secrecy levels. It suffices to use the simplest possible pointed observable. Let $\mathbb{O}_s \stackrel{\mathrm{def}}{=} \llcorner \mathtt{unit} \lrcorner_s$ and $\Downarrow$ denote termination by the standard β-reduction. Then $E \vdash M \cong^{\mathrm{DCC}}_s N : T$ when, for any context $C[\,\cdot\,]_T : \mathbb{O}_s$ such that $C[M]$ and $C[N]$ are closed, we have $C[M] \Downarrow$ iff $C[N] \Downarrow$.

3.2 Embedding

The embedding of DCC in $\pi^{\mathsf{LA}}_{\mathrm{sec}}$ is done by mapping non-pointed types to linear types and pointed ones to affine ones. The lifting $\llcorner T \lrcorner$ is replaced by a transformation from linearity to affinity. Apart from this, the overall scheme comes from [25, 23, 6]. First, the translation of types is performed on their normal forms:

$$
\begin{array}{lll}
\text{(type)} & \mathtt{unit}^\bullet_s \stackrel{\mathrm{def}}{=} ()^{\uparrow_{\mathrm{L}}}_s \quad (T_1 +_s T_2)^\bullet \stackrel{\mathrm{def}}{=} [T_1^\circ \oplus T_2^\circ]^{\uparrow_{\mathrm{L}}}_s \\
& \llcorner T \lrcorner^\bullet_s \stackrel{\mathrm{def}}{=} (T^\circ)^{\uparrow_{\mathrm{A}}}_s \\
& [T_1 ... T_{n-1}\gamma]^\circ \stackrel{\mathrm{def}}{=} \begin{cases} (\overline{T}^\circ_1 ... \overline{T}^\circ_{n-1}\gamma^\bullet)^{!_{\mathrm{A}}} & \gamma \text{ pointed} \\ (\overline{T}^\circ_1 ... \overline{T}^\circ_{n-1}\gamma^\bullet)^{!_{\mathrm{L}}} & \text{else} \end{cases} \\
\text{(base)} & \emptyset^\circ \stackrel{\mathrm{def}}{=} \emptyset \qquad (E \cdot x{:}T)^\circ \stackrel{\mathrm{def}}{=} E^\circ \cdot x{:}\overline{T^\circ} \\
\text{(action)} & \langle T \rangle_{u,E} \stackrel{\mathrm{def}}{=} \begin{cases} (u{:}T^\circ \to A) \otimes B & T \text{ non-pointed}, \\ & E^\circ = ?_{\mathrm{L}} A \otimes ?_{\mathrm{A}} B \\ u{:}T^\circ \otimes E^\circ & \text{else} \end{cases}
\end{array}
$$

The above translation elucidates the operational content of DCC-types: the type $[T_1 ... T_{n-1}\gamma]$ is now interpreted as interaction which may receive data at each T_i (at different

[1]We observe the original DCC does not satisfy the subject reduction because of the coercion $\eta_l M$. As an example, take $x : \mathtt{unit} \vdash x : \mathtt{unit}$ and $x : \mathtt{unit} \vdash \eta_\top x : (\mathtt{unit})_\top$. Then $\eta_l\ x \longrightarrow x$, but $x : \mathtt{unit} \not\vdash x : (\mathtt{unit})_\top$. In our presentation this issue does not arise due to implicit treatment of coercion.

$$
\begin{array}{llcllcl}
\textbf{(Total)} & [\mathit{Var}] & E, x\!:\!T \vdash x : T & [\mathit{Const}]\ \ E \vdash n : \mathbb{N}_s & [\mathit{Succ}] & \dfrac{E \vdash e : \mathbb{N}_s}{E \vdash \mathtt{succ}(e) : \mathbb{N}_s} & \\[2ex]
 & [\mathit{Lam}] & \dfrac{E, x\!:\!S \vdash M : T}{E \vdash \lambda x\!:\!S.M : S \Rightarrow T} & & [\mathit{App}] & \dfrac{E \vdash M : S \Rightarrow T \quad E \vdash N : S}{E \vdash MN : T} & \\[2ex]
\textbf{(Partial)} & [\mathit{LamP}] & \dfrac{E, x\!:\!U' \vdash M : U}{E \vdash \lambda x\!:\!U'.M : U' \rightharpoonup U} & & [\mathit{AppP}] & \dfrac{E \vdash M : U_1 \rightharpoonup_s U_2 \quad E \vdash N : U_1}{E \vdash MN : U_2} & \begin{array}{r}\mathsf{protect}(U_1) \sqcup s \\ \sqsubseteq \mathsf{protect}(U_2)\end{array} \\[2ex]
 & [\mathit{Rec}] & \dfrac{E, x\!:\!U \vdash M : U}{E \vdash \mu x\!:\!U.M : U}\ U \text{ pointed} & [\mathit{Lift}]\ \ \dfrac{E \vdash M : S}{E \vdash M : \llcorner S \lrcorner} & [\mathit{Seq}] & \dfrac{E \vdash N : \llcorner S \lrcorner \quad E, x\!:\!S \vdash M : U}{E \vdash \mathtt{seq}\ x = N\ \mathtt{in}\ M : U} & \\[2ex]
\textbf{(Common)} & [\mathit{If}] & \dfrac{E \vdash M : \mathbb{N}_s \quad E \vdash N_i : T}{E \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : T} & s \sqsubseteq \mathsf{protect}(T) & [\mathit{UnitM}] & \dfrac{\Gamma \vdash M : T}{\Gamma \vdash M : (T)_s} & \\[2ex]
 & [\mathit{BindM}] & \dfrac{\Gamma \vdash N : (T)_s \quad \Gamma, x\!:\!T \vdash M : T'}{\Gamma \vdash \mathtt{bind}\ x = N\ \mathtt{in}\ M : T'} & s \sqsubseteq \mathsf{protect}(T') & & &
\end{array}
$$

Figure 4: Typing Rules of Call-by-Value DCC

secrecy levels) and returns a result at γ (again at a specific secrecy level). This sheds a new light on DCC in a way quite different from the original denotational interpretation [2]. Also we observe $\mathsf{protect}(T) = \mathsf{tamp}(T^\circ)$ except when T has shape $[T_1..T_{n-1}\mathtt{unit}_s]$ (note s in $\mathtt{unit}_s$ is insignificant in its denotation since $\mathtt{unit}$ is a singleton).

The translation of DCC-terms, written $[\![M : T]\!]_u$, closely follows that of types, see Figure 8 in Appendix. Basic properties of the embedding follow. Below in (2), Ω^τ_u appears in Example 2.2 (4).

Proposition 3.1.

1. (typability) $E \vdash M\!:\!T$ *implies* $\vdash [\![M]\!]_u \triangleright \langle T\rangle_{u,E}$.
2. (adequacy) *Let* $\vdash M\!:\!\mathbb{O}_s$. *Then* $M \Downarrow$ *iff* $[\![M]\!]_u \not\cong^\pi_s \Omega^{\mathbb{O}_s}_u$.
3. (soundness) $[\![M_1]\!]_u \cong^\pi_s [\![M_2]\!]_u$ *implies* $M_1 \cong^{\mathrm{DCC}}_s M_2$.

We are now ready to establish the non-interference of DCC-terms. The result also follows from the soundness of the denotational interpretation in [2]. The present proof method has interest in that it smoothly extends to other settings such as stateful computation, cf. § 5. Write $E \vdash \sigma_1 \sim_s \sigma_2$ if, for well-typed substitutions $\sigma_{1,2}$, we have $\sigma_1(x) = \sigma_2(x)$ whenever $\mathsf{protect}(E(x)) \sqsubseteq s$. The theorem follows. The corresponding results hold for other non-trivial types.

Theorem 3.2. (non-interference) *Let* $E \vdash M\!:\!\mathbb{O}_s$. *Then for any closing* $\sigma_{1,2}$ *s.t.* $E \vdash \sigma_1 \sim_s \sigma_2$, $M\sigma_1 \Downarrow$ *iff* $M\sigma_2 \Downarrow$.

Proof: Let $\vdash N_i : T$ $(i = 1, 2)$ with $\mathsf{protect}(T) \not\sqsubseteq s$. Since $\mathsf{protect}(T) \sqsubseteq \mathsf{tamp}(T^\circ)$ we can apply Prop. 2.3 to obtain $[\![N_1]\!]_x \cong^\pi_s [\![N_2]\!]_x$. Now assume $x\!:\!T \vdash M\!:\!\mathbb{O}_s$ (the reasoning trivially extends to multiple variables). We now reason as follows. The second implication is by the replication theorem [6, 40], while the third is by Proposition 3.1 (3).

$$
\begin{array}{ll}
[\![N_1]\!]_x \cong^\pi_s [\![N_2]\!]_x & \\
\quad\Rightarrow & (\nu x)([\![M]\!]_u|[\![N_1]\!]_x) \cong^\pi_s (\nu x)([\![M]\!]_u|[\![N_2]\!]_x) \\
\quad\Rightarrow & [\![M\{N_1/x\}]\!]_u \cong^\pi_s [\![M\{N_2/x\}]\!]_u \\
\quad\Rightarrow & M\{N_1/x\} \cong^{\mathrm{DCC}}_s M\{N_2/x\} \\
\quad\Rightarrow & M\{N_1/x\} \Downarrow \text{ iff } M\{N_2/x\} \Downarrow . \qquad \blacksquare
\end{array}
$$

$\pi^{\mathsf{LA}}_{\mathrm{sec}}$ may also be used for justifying a call-by-value DCC given in [2]. Interestingly, $\pi^{\mathsf{LA}}_{\mathrm{sec}}$ motivates a more direct formulation of a call-by-value version of DCC, which is useful when we consider combination with imperative features, including concurrency. We use PCF-like types and syntax, which is more convenient for our discussion in Section 5.

$$
\begin{array}{llll}
\text{(type)} & T & ::= & S \ \mid\ U \ \mid\ (T)_s \\
 & S & ::= & \mathbb{N}_s \ \mid\ S \Rightarrow T \qquad U \ ::=\ \llcorner S \lrcorner_s \ \mid\ U_1 \rightharpoonup_s U_2
\end{array}
$$

Here S is a total type, while U is a partial one (among partial types those of form $U_1 \rightharpoonup_s U_2$ are pointed). We use a non-standard lifting for brevity of encoding. $\mathsf{protect}(T)$ is defined as before for total types and for others by the outermost secrecy level. The typing rules are given in Figure 4. The non-interference is proved in the same way, using an embedding into $\pi^{\mathsf{LA}}_{\mathrm{sec}}$. The encoding of types is given by:

$$
\begin{array}{lll}
\text{(type)} & S^\bullet \stackrel{\mathrm{def}}{=} (S^\circ)^{\uparrow_{\mathrm{L}}} & U^\bullet \stackrel{\mathrm{def}}{=} (U^\circ)^{\uparrow_{\mathrm{A}}}_s \ \ (\mathsf{sec}(U) = s) \\
 & \mathbb{N}^\circ_s = ([\otimes_{i\in\omega}\ ]^{\uparrow_{\mathrm{L}}}_s)^{!_{\mathrm{L}}} & (S \Rightarrow T)^\circ \stackrel{\mathrm{def}}{=} (\overline{S^\circ} T^\bullet)^{!_{\mathrm{L}}} \\
 & \llcorner S \lrcorner^\circ_s \stackrel{\mathrm{def}}{=} S^\circ & (U_1 \rightharpoonup_s U_2)^\circ \stackrel{\mathrm{def}}{=} (\overline{U^\circ_1} U^\bullet_2)^{!_{\mathrm{A}}}
\end{array}
$$

Here $\mathsf{sec}(T)$ is T's outermost secrecy level. Base is mapped as before, using $(\ )^\circ$. Then $\langle T\rangle_{u,E} \stackrel{\mathrm{def}}{=} u : T^\bullet \otimes E^\circ$. The encoding of terms is given in Figure 9 in Appendix.

4. INTEGRATING STATE

4.1 Mutable interaction

In this section we present a framework for a consistent integration of stateful, or mutable, computation into linear/affine type disciplines. We only discuss the incorporation of linear mutable replication, which is all we use in the application in the next section. We first extend the set of processes.

$$P \ ::=\ \ldots \mid \mathsf{ref}\langle xy\rangle^\tau$$

Here $\mathsf{ref}\langle xy\rangle^\tau$ is a constant with the following dynamics.

$$
\begin{array}{l}
\mathsf{ref}\langle xy\rangle^\tau \mid \overline{x}\mathtt{in}_1(c)P \longrightarrow \mathsf{ref}\langle xy\rangle^\tau \mid (\boldsymbol{\nu}\, c)(P \mid \overline{c}\langle y\rangle^\tau) \\
\mathsf{ref}\langle xy\rangle^\tau \mid \overline{x}\mathtt{in}_2(wc)P \longrightarrow (\boldsymbol{\nu}\, wc)(\mathsf{ref}\langle xw\rangle^\tau \mid P \mid \overline{c})
\end{array}
$$

where $\overline{x}\langle y\rangle^\tau \stackrel{\mathrm{def}}{=} \overline{x}(z)[z \to y]^{\overline{\tau}}$ (note this agent has an output at y). We use this constant together with replication instead of recursion, because the class of realizable (typed) behaviour is identical, cf. [4]. For information, we also present the recursive definition of $\mathsf{ref}\langle xy\rangle$.

$$\mathsf{ref}\langle xy\rangle^\tau \stackrel{\mathrm{def}}{=} x[(c).(\mathsf{ref}\langle xy\rangle^\tau|\overline{c}\langle y\rangle^\tau)\&(wc).(\mathsf{ref}\langle xw\rangle^\tau|\overline{c})]$$

For types, we write $!_{L\mu}$ and $?_{L\mu}$ for the mutable versions of $!_L$ and $?_L$, which are mutually dual. We add $!_{L\mu}/?_{L\mu}$ to $\mathcal{M}_!/\mathcal{M}_?$. The channel types are extended as follows.

$$\begin{array}{llll} \tau_{\mathtt{I}} & ::= & \ldots \mid (\vec{\tau})^{!_{L\mu}} \mid [\&_i \vec{\tau}_i]_s^{\langle p_i \rangle} & p_i \in \{!_L, !_{L\mu}\} \\ \tau_{\mathtt{O}} & ::= & \ldots \mid (\vec{\tau})_s^{?_{L\mu}} \mid [\oplus_i \vec{\tau}_i]_s^{\langle p_i \rangle} & p_i \in \{?_L, ?_{L\mu}\} \end{array}$$

We let each branch of a branching/selection type take an action mode. The channel types of the above shape are called *mutable*. Let $p_i \in \{!_L, !_{L\mu}\}$. We set $\sqcap\{p_i\}_{i\in I} = !_L$ if $p_i = !_L$ for each $i \in I$ else $\sqcap\{p_i\}_{i\in I} = !_{L\mu}$. Then we define $\mathsf{md}([\&_i \vec{\tau}_i]_s^{\langle p_i \rangle}) = \sqcap\{p_i\}$. Dually for selection. For algebra of types, we add the following:

$$\begin{array}{lll} [\&_{i\in I}\vec{\tau}_i]_s^{\langle p_i \rangle} \odot [\oplus_{j\in J}\overline{\vec{\tau}_j}]_s^{\overline{\langle p_j \rangle}} & = & [\&_{i\in I}\vec{\tau}_i]_s^{\langle p_i \rangle} \qquad (J \subset I) \\ [\oplus_{i\in I}\vec{\tau}_i]_s^{\langle p_i \rangle} \odot [\oplus_{j\in J}\vec{\tau}_j]_s^{\langle p_j \rangle} & = & [\oplus_{h\in I\cup J}\vec{\tau}_h]_s^{\langle p_h \rangle} \end{array}$$

For example, $[\oplus_{i=1}\vec{\tau}_i]_s^{\langle ?_L \rangle} \odot [\oplus_{i=2}\vec{\tau}_i]_s^{\langle ?_{L\mu} \rangle} = [\oplus_{i\in\langle 1,2\rangle}\vec{\tau}_i]_s^{\langle ?_L, ?_{L\mu}\rangle}$. We also assume the same sequentiality constraint as in Appendix A (with $!_{L\mu}$ and $?_{L\mu}$ acting as $!_L$ and $?_L$, respectively).

Action types now use the extended set of channel types, where we additionally consider $x : \tau \to y : \tau'$ with τ, τ' mutable and $\mathsf{md}(\tau') \in \{?_L, ?_{L\mu}\}$ when τ and τ' have the same height (the *height* of a channel type is given in Appendix B). The use of height of types is to avoid an analogue of well-known discrepancy between strong normalisability and state in λ-calculi. We note there are several ways to maintain linearity in the presence of mutable interaction, which will be discussed elsewhere.

4.2 Structural security

A new element in secrecy typing is a well-formedness condition for channel types. It reflects a different way in which information leaks in stateful computing. We first state the condition, then illustrate the idea. Below $\mathsf{tamp}(\tau)$ is given by the same clauses as Definition 2.1, setting τ such that $\mathsf{md}(\tau) = ?_{L\mu}$ to be immediately affecting.

DEFINITION 4.1. τ is *structurally secure* if for each occurrence τ' in τ (1) $\mathsf{sec}(\tau') \sqsubseteq \mathsf{tamp}(\tau')$ when $\mathsf{md}(\tau') = !_{L\mu}$, and (2) $\mathsf{sec}(\tau') \sqsubseteq \mathsf{tamp}(\overline{\tau'})$ when $\mathsf{md}(\tau') = ?_{L\mu}$.

The definition says a mutable type should have higher tampering levels in carried types. This is to prevent leakage of information, as the following example shows. Below $\mathbb{N}_s^\star$ stands for $([\oplus_{i\in\omega}\ ]_s^{\uparrow_A})^{!_A}$; H and L for $\top$ and $\bot$; $\mathsf{ref}\langle x1\rangle$ for $(\boldsymbol{\nu}\, v)(\mathsf{ref}\langle xv\rangle \mid [\![1]\!]_v)$; and $\overline{x}\mathtt{in}_2(2c)P$ for $\overline{x}\mathtt{in}_2(wc)([\![2]\!]_w \mid P)$.

EXAMPLE 4.2. Let $\tau \stackrel{\text{def}}{=} [(\mathbb{N}_L^\star)^{\uparrow_L}\&\overline{\mathbb{N}_L^\star}()^{\uparrow_L}]_H^{!_{L\mu}, !_{L\mu}}$, which is *not* structurally secure. Now consider:

$$P \stackrel{\text{def}}{=} \mathsf{ref}\langle x1\rangle \mid \overline{x}\mathtt{in}_2(2c)c.\mathbf{0} \mid \overline{x}\mathtt{in}_1(c)c(y).\overline{y}(e)e[\&_{i\in\{1,2\}}P_i]$$

where $P_1 = \overline{u}\mathtt{in}_n$ and $P_2 = \Omega_u$. By the racing condition at x, this agent may or may not emit at u, i.e. we have either:

$$P \longrightarrow^+ \mathsf{ref}\langle x2\rangle \mid \overline{u}\mathtt{in}_n \quad \text{ or } \quad P \longrightarrow^+ \mathsf{ref}\langle x2\rangle \mid \Omega_u.$$

Hence writing at the high-level channel x affects an action at a low-level channel u.

The anomaly takes place because stateful agents can transmit information using time-difference, storing what has happened in its state to transmit it later [25]. Structurally secure types prevent this leakage by requiring that a stateful replication to transmit information at the same, or higher, level than it receives. *Hereafter we assume all channel types are structurally secure.*

4.3 Secrecy typing

We have the following additional typing rule for constant.

$$\frac{\text{(Ref)} \quad \text{if } \mathsf{md}(\tau) = !_{L\mu} \text{ then } p = !_{L\mu} \text{ else } p = !_L}{\vdash \mathsf{ref}\langle xy\rangle^{\overline{\tau}} \triangleright x : [(\tau)^{\uparrow_L}\&\overline{\tau}()^{\uparrow_L}]_s^{p, !_{L\mu}} \otimes y : \overline{\tau}}$$

The right-branch of a reference receives the "write" action, so it is always mutable; while the left-branch receives the "read" action, whose mode depends on that y (at which it has an output, cf.§ 4.1). We also need the typing rules for mutable prefixes, which we list in Appendix B. The resulting typed calculus, which we hereafter call $\pi_{\text{sec}}^{\mathsf{LA}\mu}$, satisfies subject reduction as well as the following non-interference property. Below $\cong_s^\pi$ is defined precisely as before.

PROPOSITION 4.3. (non-interference) *Let* $\vdash P_{1,2} \triangleright A$ *such that* $\mathsf{tamp}(A) = s$*. Then* $s \not\sqsubseteq s'$ *implies* $\vdash P_1 \cong_{s'}^\pi P_2 \triangleright A$*.*

5. CONCURRENCY, REFERENCE AND PROCEDURE

5.1 A Volpano-Smith language

We briefly overview 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. λ-abstraction mentions type T, which will be introduced later.

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

The syntax of commands is from [35], extended with general references, local variable declaration and higher-order procedures. We use two let commands for simpler presentation of typing rules, though we shall be sometimes informal about them, writing e.g. $x := !y$ instead of $\mathtt{let}\ z = !y\ \mathtt{in}\ x := z$. For brevity of presentation we do not include $\mathtt{let}$s and $\mathtt{new}$ in expressions.

The reduction rules of commands are given in Figure 5. The reduction takes form $(c, \sigma) \longrightarrow (c', \sigma')$ where $\sigma, \sigma', \ldots$ denote *environments*, i.e. finite maps from names to values. We use a special command 0 for which we set: $(c, \sigma) \longrightarrow^* (0, \sigma')$ when $(c, \sigma) \longrightarrow^* \sigma'$. We write $(o, \sigma) \Downarrow \sigma'$ when $(o, \sigma) \longrightarrow^* (\Pi_i 0, \sigma')$. For expressions we assume the standard call-by-value (single-step) reduction.

5.2 Secrecy with reference and procedure

We first illustrate the subtlety in secrecy with local references and procedure by examples. *For brevity we assume* u, v, w *are low-level variables while* x, y, z *are high-level variables in the following examples.*

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

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

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

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

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

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

$$(\texttt{let } x = e \texttt{ in } c,\sigma) \longrightarrow (\texttt{let } x = e' \texttt{ in } c,\sigma') \qquad ((e,\sigma) \longrightarrow (e',\sigma'))$$

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

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

$$\frac{(c,\ \sigma \cup [x \mapsto v]) \longrightarrow (c',\ \sigma' \cup [x \mapsto v'])}{(\texttt{new } x \mapsto v \texttt{ in } c,\sigma) \longrightarrow (\texttt{new } x \mapsto v' \texttt{ in } c',\sigma')}$$

$$\frac{(c_i,\ \sigma) \longrightarrow (c_i',\ \sigma')}{(\prod_i c_i,\sigma) \longrightarrow (\prod_i c_i',\sigma')}$$

We omit reduction rules for expressions.

Figure 5: Reduction of VS-Calculus with Reference and Procedure

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

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

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

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

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

Imperative procedures. DCC and its CBV version in § 3 capture non-trivial features of secrecy in pure higher-order functions. With imperative features, procedures add different kinds of subtlety.

- (Divergence) Let $e_1 \stackrel{\text{def}}{=} \lambda y.(!x)y$, $e_2 \stackrel{\text{def}}{=} \lambda y.y$ and $c_3 \stackrel{\text{def}}{=} u := 1; (\texttt{if } z \texttt{ then } x := e_1 \texttt{ else } x := e_2); z' := (!x)0; u := 0.$ Then c_3 reveals z at u by diverging when $z = \texttt{true}$.
- (Side effects) Take $e_3 \stackrel{\text{def}}{=} \lambda x.\ u := x \texttt{ return } 0$. Then
  $$c_4 \stackrel{\text{def}}{=} \texttt{if } z \texttt{ then let } y = (e_3)0 \texttt{ in skip}.$$
  leaks information at u, though e_3 is secure as a function.
- (Aliasing) Given $e_5 \stackrel{\text{def}}{=} \lambda u.\ !!u := 1 \texttt{ return } 0$,
  $$c_5 \stackrel{\text{def}}{=} \texttt{if } z \texttt{ then new } v \mapsto w \texttt{ in let } x = (e_5)v \texttt{ in skip}$$
  is not secure since w can be aliased. However if we further hide w, the command becomes secure.

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

5.3 Types

The syntax of types for commands and expressions follows. We only treat total types in the sense of call-by-value DCC (see § 3.2). The incorporation of partial types easily follows, which is briefly mentioned in § 5.5. For command types we use *action sets*, denoted $X, Y, \ldots$. An action set contains elements of form $\mathtt{w}x$ and $\mathtt{r}x$, which respectively indicate a possible write and a possible read at x. In $E \cdot x : T$ below, we assume x does not occur in E.

$$\begin{array}{lll}
\text{(value)} & T & ::= \mathbb{N}_s \mid \texttt{ref}_s(T) \mid T_1 \Rightarrow T_2 \mid T_1 \stackrel{*}{\Rightarrow}_s T_2 \\
\text{(base)} & E & ::= \emptyset \mid E \cdot x : T \\
\text{(command)} & \rho & ::= \mathrm{cmd}\, \tau_s\, X \qquad (\tau \in \{\Downarrow, \Uparrow\})
\end{array}$$

In value types, $\Rightarrow$ indicates a pure (total) function space, while $\stackrel{*}{\Rightarrow}$ indicates a (total) function type with side effects. $\stackrel{*}{\Rightarrow}$ mentions a secrecy level, just as reference types. We say T is *mutable* if it is of form either $\texttt{ref}_s(T)$ or $T_1 \stackrel{*}{\Rightarrow}_s T_2$. We write $E \vdash X$ when: (1) $mx \in X$ implies $x \in \mathtt{dom}(E)$ ($\mathtt{dom}(E)$ is the domain of E), and (2) $\mathtt{w}x \in X$ implies $E(x)$ is mutable.

In $\mathrm{cmd}\,\tau_s X$, $\tau = \Downarrow$ (resp. $\tau = \Uparrow$) indicates convergence (resp. potential divergence); s is a lower bound at which the termination may be observed (or, as Smith [34] puts it, at which variables a termination depends upon). $\mathtt{w}x$ (resp. $\mathtt{r}x$) indicate x may be written to (resp. read from).

We use the subtyping on value and command types, which largely come from [35, 18, 34]. For value types, we have:

$$\frac{s \sqsubseteq s'}{\mathbb{N}_s \leq \mathbb{N}_{s'}} \qquad \frac{s' \sqsubseteq s}{\texttt{ref}_s(T) \leq \texttt{ref}_{s'}(T)}$$

$$\frac{T_1' \leq T_1 \quad T_2 \leq T_2'}{T_1 \Rightarrow T_2 \leq T_1' \Rightarrow T_2'} \qquad \frac{T_1' \leq T_1 \quad T_2 \leq T_2' \quad s' \sqsubseteq s}{T_1 \stackrel{*}{\Rightarrow}_s T_2 \leq T_1' \stackrel{*}{\Rightarrow}_{s'} T_2'}$$

Note T does not vary in $\texttt{ref}_s(T)$, see illustration in § 5.4. The subtyping on command types uses $E \vdash X$ defined above.

$$\frac{X \subset Y \quad E \vdash Y}{\mathrm{cmd} \Downarrow_s X \leq_E \mathrm{cmd} \Downarrow_{s'} Y} \qquad \frac{X \subset Y \quad E \vdash Y}{\mathrm{cmd} \Downarrow_s X \leq_E \mathrm{cmd} \Uparrow_s Y}$$

$$\frac{s' \sqsubseteq s \quad X \subset Y \quad E \vdash Y}{\mathrm{cmd} \Uparrow_s X \leq_E \mathrm{cmd} \Uparrow_{s'} Y}$$

Note secrecy levels are irrelevant in converging commands.

5.4 Information level and safety

As in DCC, we use the protection level of value types. As before, $\mathsf{sec}(T)$ gives the outermost secrecy level of T.

- $\mathsf{protect}(\mathbb{N}_s) = s$, $\mathsf{protect}(T_1 \Rightarrow T_2) = \mathsf{protect}(T_2)$.
- $\mathsf{protect}(\texttt{ref}_s(T)) = \mathsf{protect}(T) \sqcap \mathsf{sec}(T)$ if T is mutable. $\mathsf{protect}(\texttt{ref}_s(T)) = \mathsf{protect}(T)$ if else.
- $\mathsf{protect}(T_1 \stackrel{*}{\Rightarrow}_s T_2) = \mathsf{sec}(T_1) \sqcap \mathsf{protect}(T_2)$ if T_1 is mutable. $\mathsf{protect}(T_1 \stackrel{*}{\Rightarrow}_s T_2) = \mathsf{protect}(T_2)$ if else.

The definition takes into account the level of types which occur contravariantly (cf. Def.2.1). The condition on mutability will be illustrated later via its translation into the $\pi^{\mathsf{LA}\mu}_{\mathsf{sec}}$-types. We can now introduce a basic condition on value types, which plays a key rôle for harnessing aliases.

Definition 5.1. (safety) T is *safe* when: (a) $T = \mathbb{N}_s$, (b) $T = T_1 \Rightarrow T_2$ and $T_{1,2}$ are safe, (c) $T = \texttt{ref}_s(T')$, T' is safe and $s \sqsubseteq \mathsf{protect}(T)$, and (d) $T = T_1 \stackrel{*}{\Rightarrow}_s T_2$, $T_{1,2}$ are safe and $s \sqsubseteq \mathsf{protect}(T)$.

$$[\mathit{Skip}]\quad E \vdash \mathtt{skip} : \mathrm{cmd} \Downarrow_s X \qquad E \vdash X$$

$$[\mathit{Seq}]\quad \frac{E \vdash c_i : \mathrm{cmd}\, \tau_{s_i}\, X_i \quad (i=1,2)}{E \vdash c_1 ; c_2 : \mathrm{cmd}\, \tau_{s_2}\, X_1 \cup X_2} \qquad \text{if } \tau = \Uparrow \text{ then } s_1 \sqsubseteq s_2 \sqcap \mathsf{tamp}_E(X_2)$$

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

$$[\mathit{Let}]\quad \frac{E \vdash e : T, X \quad E \cdot x : T \vdash c : \mathrm{cmd}\, \tau_{s'}\, Y}{E \vdash \mathtt{let}\ x = e\ \mathtt{in}\ c : \mathrm{cmd}\, \tau_{s'}\, X \cup Y/x}$$

$$[\mathit{New}]\quad \frac{E \vdash v : T, X \quad E \cdot x : \mathtt{ref}_s(T) \vdash c : \mathrm{cmd}\, \tau_{s_0}\, Y}{E \vdash \mathtt{new}\ x \mapsto v\ \mathtt{in}\ c : \mathrm{cmd}\, \tau_{s_0}\, X \cup Y/x}$$

$$[\mathit{Ass}]\quad \frac{E(x) = \mathtt{ref}_{s_0}(T) \quad E \vdash v : T, X}{E \vdash x := v : \mathrm{cmd} \Downarrow_s X \cup \{\mathtt{w}x\}}$$

$$[\mathit{Sub}]\quad \frac{E \vdash c : \rho}{E \vdash c : \rho'} \qquad \rho \leq_E \rho'$$

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

$$[\mathit{Deref}]\quad \frac{E \vdash z : \mathtt{ref}_s(T), Y \quad E \cdot x : T \vdash c : \mathrm{cmd}\, \tau_{s_0}\, Y}{E \vdash \mathtt{let}\ x = !z\ \mathtt{in}\ c : \mathrm{cmd}\, \tau_{s_0}\, X/x} \qquad \text{if } T \text{ mutable then } \mathtt{w}z \in X$$

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

Figure 6: Typing System for VS-Calculus with Reference and Procedure (command)

The condition is directly suggested by $\pi_{\mathrm{sec}}^{\mathsf{LA}\mu}$(cf. Def. 4.1). In essence, it says that, as a command unfolds a sequence of references, the secrecy level either remains the same or gets higher. As an example, take the following program:

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

The condition statically ensures that w is higher than x and z. From the viewpoint that a program should be prohibited from writing at a low-level as the result of getting a high-level information, as well as observing we may safely raise the secrecy level of a local resource, we claim that the constraint is reasonable in practice, at least for basic programming. *Hereafter we assume we only use safe types.*

5.5 Typing

The typing rules are given in Figure 6 (for commands) and 7 (for expressions), using judgements $E \vdash e : T, X$ (for expressions), $E \vdash c : \mathrm{cmd}\, \tau_s\, X$ (for commands), and $E \vdash o : \mathrm{cmd}\, \tau_s\, X$ (for threads). In the rules, $\mathsf{md}(X)$ denotes the set of $\mathtt{w}$ and $\mathtt{r}$ in X. X/x is the result of taking off $\mathtt{r}x$ and $\mathtt{w}x$ (if any) from X. The *height* of T, $\mathsf{ht}(T)$, is given as: $\mathsf{ht}(\mathbb{N}_s) = 1$, $\mathsf{ht}(\mathtt{ref}_s(T)) = \mathsf{ht}(T) + 2$, and $\mathsf{ht}(T_1 \Rightarrow T_2) = \mathsf{ht}(T_1 \overset{*}{\Rightarrow}_s T_2) = \mathsf{ht}(T_1) + \mathsf{ht}(T_2) + 2$ (this is in accordance with the encoding of types in § 5.6 later). $\mathsf{tamp}_X(E)$ (cf. Def.2.1) is defined as:

$$\mathsf{tamp}_E(X) \overset{\mathrm{def}}{=} \sqcap\{\mathsf{protect}(E(x)) \mid \mathtt{w}x \in X\}$$

The system is a conservative extension of [34] (neglecting $\mathsf{protect}$ [8, 18]). Below we illustrate the typing rules, concentrating on those points which are new in the present system.

- *General.* The typing system uses an action set for capturing the level of writing and for ensuring convergence for total types. Its manipulation is crucial for capturing aliasing effects.
- *Assignment.* The rule crucial relies on the safety condition (Definition 5.1). For example, $u := !x$ with u and x typed as $\mathtt{ref}_L(\mathbb{N}_L)$ and $\mathtt{ref}_H(\mathbb{N}_L)$ (which is unsafe), respectively, becomes typable without safety. As expected, the rule adds $\mathtt{w}x$ as a write variable.
- *Seq, If, While. [Seq]*'s side condition is equivalent to [34], which enhances [18, 35]. If the preceding command may not terminate, the termination (at s_1) should not flow down to c_2's termination (s_2) and tampering ($\mathsf{tamp}_E(X)$). *[If]* and *[While]* are standard, requiring the conditional variable cannot influence later behaviour at lower levels.
- *Deref.* Note $\mathtt{w}x$ is added if x is mutable, even though x is read. To see its necessity, consider:

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

  z looks local, but may be aliased to a free name. By keeping x (which is lower than z by safety) in the action set, we effectively record the writing at z.
- *Id.* Similar to *[Deref]*, we record $\mathtt{w}x$ when it is a reference. To understand its necessity, consider:

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

  Note y (like z) should have a reference type. Hence when $x \mapsto y$ is inferred, $\mathtt{w}y$ is recorded, which subsumes the writing at z since z is higher than y by safety.
- *Lam, Lam-∗. [Lam]* prohibits access to free names of mutable types inside a pure procedure. In *[Lam-∗]*, this constraint does not exist. In *[Lam-∗]* we require constraint on the height of types, to avoid divergence on total types (which are dropped if we treat a partial type).
- *App, App-∗.* These rules do not mention secrecy levels since they assume the arguments always terminate. If we assume possibly nonterminating arguments, applications become secrecy-sensitive for both termination and tampering as in *[Seq]*, cf. *[AppP]* in Figure 4.

Typing examples follow (commands/expressions are from § 5.2; x, y, z are high while u, v, w are low in E).

(i) $E/u \vdash c_1 : \mathrm{cmd} \Downarrow_s \mathtt{w}x, \mathtt{r}v$ for arbitrary s (we omit such s from now on). Hence its tamper level is high.

(ii) $E/xz \vdash c_2 : \mathrm{cmd} \Downarrow \mathtt{w}u$ (with bound x and z typed low). Hence its tamper level is low.

(iii) e_1, hence c_3, is untypable by the condition on heights in [Lam-∗].

(iv) $E/x \vdash e_3 : \mathbb{N}_H \overset{*}{\Rightarrow}_L \mathbb{N}_H, \mathtt{w}u$ while c_4 is ill-typed under E, not because of $\overset{*}{\Rightarrow}_L$ but by $\mathtt{w}u$.

(v) c_5 is ill-typed by recording $\mathtt{w}w$ in the action set.

The typing system satisfies standard properties such as subject reduction. For the noninterference property, we again use the embedding into secure processes.

$[Id]$	$E, x{:}T \vdash x : T, X \quad (E \vdash X)$	if T mutable then $\mathtt{w}x \in X$ else $\mathtt{r}x \in X$	$[Const]$	$E \vdash n : \mathbb{N}_s, X \qquad (E \vdash X)$
$[Lam]$	$\dfrac{E, x{:}T \vdash e : T', X}{E \vdash \lambda x{:}T.e : T \Rightarrow T', X/x}$	If $y \in \mathsf{fn}(X)$ then $E(y)$ not mutable.	$[App]$	$\dfrac{E \vdash e : T \Rightarrow T', X \quad E \vdash e' : T, Y}{E \vdash ee' : T', X \cup Y}$
$[Lam\text{-}*]$	$\dfrac{E, x{:}T \vdash e : T', X}{E \vdash \lambda x{:}T.e : T \overset{*}{\Rightarrow}_s T', X/x}$	$s \sqsubseteq \mathsf{tamp}_E(X/x), \ (*)$	$[App\text{-}*]$	$\dfrac{E \vdash e : T \overset{*}{\Rightarrow}_s T', X \quad E \vdash e' : T, Y}{E \vdash ee' : T', X \cup Y}$
$[Ret]$	$\dfrac{E \vdash c : \mathrm{cmd} \Downarrow_s X \quad E \vdash e : T, Y}{E \vdash c\ \mathtt{return}\ e : T, X \cup Y}$		$[Sub]$	$\dfrac{E \vdash e : T, X \quad T <_E T'}{E \vdash e : T', X}$

$(*)$ $\mathsf{ht}(T \Rightarrow T') \gtrsim \mathsf{ht}(E(y))$ for each $y \in \mathsf{fn}(X/x)$ s.t. $E(y)$ mutable.

Figure 7: Typing System for VS-Calculus with Reference and Procedure (expressions)

5.6 Embedding and noninterference

We now embed the typing system in $\pi^{\mathsf{LA}\mu}_{\mathsf{sec}}$. First we embed types. Below in (base), $\mathtt{r}(\tau)$ is the result of taking off $?_{\mathrm{L}\mu}$-mode branches (if any) from τ.

$$\text{(value)}\quad T^{\bullet} \stackrel{\text{def}}{=} (T^{\circ})^{\uparrow_{\mathrm{L}}} \quad \mathbb{N}^{\circ}_s = ([\oplus_i\ ]^{\uparrow_{\mathrm{L}}}_s)^{!_{\mathrm{L}}}$$

$$\mathtt{ref}_s(T)^{\circ} \stackrel{\text{def}}{=} \begin{cases} [(T^{\circ})^{\uparrow_{\mathrm{L}}} \& \overline{T^{\circ}}()^{\uparrow_{\mathrm{L}}}]^{!_{\mathrm{L}\mu}, !_{\mathrm{L}\mu}}_s & (T \text{ mutable}) \\ [(T^{\circ})^{\uparrow_{\mathrm{L}}} \& \overline{T^{\circ}}()^{\uparrow_{\mathrm{L}}}]^{!_{\mathrm{L}}, !_{\mathrm{L}\mu}}_s & (\text{else}) \end{cases}$$

$$(T_1 \Rightarrow T_2)^{\circ} \stackrel{\text{def}}{=} (\overline{T_1{}^{\circ} T_2^{\bullet}})^{!_{\mathrm{L}}} \quad (T_1 \overset{*}{\Rightarrow}_s T_2)^{\circ} \stackrel{\text{def}}{=} (\overline{T_1{}^{\circ} T_2^{\bullet}})^{!_{\mathrm{L}\mu}}_s$$

$$\text{(base)}\quad (\emptyset)^{\circ}_X = \emptyset \quad (E \cdot x^s : T)^{\circ}_X = \begin{cases} (E)^{\circ}_X \cdot x : \mathtt{r}(\overline{T^{\circ}}) & (\mathtt{w}x \notin X) \\ (E)^{\circ}_X \cdot x : \overline{T^{\circ}} & (\text{else}) \end{cases}$$

$$\text{(action)}\quad \Downarrow^{\circ}_s \stackrel{\text{def}}{=} ()^{\uparrow_{\mathrm{L}}} \qquad \Downarrow^{\circ}_s \stackrel{\text{def}}{=} ()^{\uparrow_{\mathrm{A}}}_s$$

$$\langle \mathrm{cmd}\, \tau_s\, X \rangle_{u,E} \stackrel{\text{def}}{=} u : \tau^{\circ}_s \otimes (E)^{\circ}_X$$

$$\langle T, X \rangle_{u,E} \stackrel{\text{def}}{=} u : T^{\bullet} \otimes (E)^{\circ}_X$$

The subtyping in command types for converging commands, cf. § 5.3, is now given a clear account: the termination channel has a unary $\uparrow_{\mathrm{L}}$-type, so its level is insignificant. Similarly, the invariance in subtyping of reference types is elucidated by observing the content type now occurs both covariantly and contravariantly [4, 30] (in fact the subtyping on value types in § 5.3 precisely corresponds to a naturally defined secrecy subtyping in $\pi^{\mathsf{LA}\mu}_{\mathsf{sec}}$-types). Finally $\mathsf{protect}(T) = \mathsf{tamp}(T^{\circ}) (= \mathsf{tamp}(T^{\bullet}))$, using which we also know T is safe iff T° (hence $T^{\bullet}$) is structurally secure.

The encoding of commands and expressions is given in Figure 10 in Appendix. Expressions use call-by-value encoding [25, 20]. `while` is translated using tail recursion. We can easily verify:

Proposition 5.2. *$E \vdash e : T, X$ implies $\vdash [\![e]\!]^E_u \triangleright \langle T, X \rangle_{u,E}$, $E \vdash c : \rho$ implies $\vdash [\![c]\!]^E_u \triangleright \langle \rho \rangle_{u,E}$ and $E \vdash o : \rho$ implies $\vdash [\![o]\!]^E_u \triangleright \langle \rho \rangle_{u,E}$.*

Now define $E \vdash \sigma_1 \sim_s \sigma_2$ precisely as in § 3.2 and let $[\![\sigma]\!]^E \stackrel{\text{def}}{=} \prod_i \mathsf{ref}\langle x_i v_i \rangle^{T_i}$. Then we have $E \vdash \sigma_1 \sim_s \sigma_2$ iff $\vdash [\![\sigma_1]\!]^E \cong^{\pi}_s [\![\sigma_2]\!]^E$ ["only if" is from Proposition 4.3, while for "if" we use contextual reasoning for each i-th component]. Using the simulation in reduction in addition and reasoning just as in Section 3, we conclude:

Theorem 5.3. (non-interference) *Let $E \vdash o : \mathit{cmd}\, \tau_s\, X$ and $E \vdash \sigma_1 \sim_s \sigma_2$. Then $(o, \sigma_1) \longrightarrow^* \sigma'_1$ implies $(o, \sigma_2) \longrightarrow^* \sigma'_2$ such that $E \vdash \sigma'_1 \sim_s \sigma'_2$.*

There are a few remaining topics. First the semantic counterpart of $\pi^{\mathsf{LA}\mu}_{\mathsf{sec}}$ is worth studying, which can be used for refining typing rules and justifying safety of untypable programs. See [41] for a recent work in this direction. Second we have not touched the possibility of refining $\pi^{\mathsf{LA}\mu}_{\mathsf{sec}}$ to take information leak by time consumption into account [5, 8, 34]. Since the simulation in the given embeddings is close, we believe this direction is feasible. Finally we have not considered in this report how other elements such as polymorphism and control as well as other security concerns can be incorporated in the present framework.

6. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] 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.

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

[18] Honda, K., Vasconcelos, V. and Yoshida, N., Secure Information Flow as Typed Process Behaviour, *ESOP'00*, LNCS 1782, 180–199, 2000.

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

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

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

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

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

[24] Kobayashi, N., Pierce, B., and Turner, D., Linear types and π-calculus, *POPL'96*, 358–371, 1996.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[39] Yoshida, N. Graph Types for Mobile Processes. *FST/TCS'16*, LNCS 1180, 371–386, Springer, 1996.

[40] Yoshida, N., Berger, M. and Honda, K., Strong Normalisation in the π-Calculus, *LICS'01*, 311–322, IEEE, 2001.

[41] Yoshida, N., Honda, K. and Berger, M., Linearity and Bisimulation, To appear as MCS technical report, Leicester, 2001.

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

APPENDIX

A. ADDITIONAL DEFINITIONS FOR $\pi^{\mathsf{LA}}_{\mathsf{sec}}$

The following gives the sequentiality constraint for unary types: for branching/selection, we require the same constraint for each summand. Let $\vec{\tau} = \tau_1..\tau_n$ below.

(C1) In $(\vec{\tau})^p$ with $p \in \mathcal{M}_\downarrow$, $\mathsf{md}(\tau_i) \in \mathcal{M}_?$ for each $1 \leq i \leq n$. Dually when $p \in \mathcal{M}_\uparrow$.

(C2) In $(\vec{\tau})^{!_\mathrm{L}}$ $\mathsf{md}(\tau_i) \in \mathcal{M}_?$ for each $1 \leq i \leq n$ except at most one j for which $\mathsf{md}(\tau_j) \in \mathcal{M}_\uparrow$. Dually for $(\vec{\tau})^{?_\mathrm{L}}$.

(C3) In $(\vec{\tau})^{!_\mathrm{A}}$, $\mathsf{md}(\tau_i) \in \mathcal{M}_?$ for each $1 \leq i \leq n$ except at most one j for which $\mathsf{md}(\tau_j) = \uparrow_\mathrm{A}$. Dually for $(\vec{\tau})^{?_\mathrm{A}}$.

The key constraint for integration is that $\uparrow_\mathrm{L}$ can only be carried by a linear replication $!_\mathrm{L}$. If this is violated, then linearity can no longer be maintained. This IO-alternation and a unique answer at $\uparrow$ at each server type come from game semantics [3, 20, 23] (see [6]). The typing rules for selection and branching are defined as follows.

$$\dfrac{(\mathsf{Sel})\quad (p \in \mathcal{M}_{\uparrow,?}) \qquad \vdash P \triangleright C\langle \vec{y}:\vec{\tau}_j\rangle \quad C/\vec{y} = A \asymp x:[\oplus_i \vec{\tau}_i]^p_s}{\vdash \overline{x}\mathtt{in}_j(\vec{y})P \triangleright A \odot x:[\oplus_i \vec{\tau}_i]^p_s}$$

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

Note that we need to check tampering level in $(\mathrm{Bra}^{\downarrow_\mathrm{L}})$ in contrast to $(\mathrm{In}^{\downarrow_\mathrm{L}})$. $(\mathrm{Bra}^{\downarrow_\mathrm{A}})$, $(\mathrm{Bra}^{!_\mathrm{L}})$ and $(\mathrm{Bra}^{!_\mathrm{A}})$ are defined just as $(\mathrm{In}^{\downarrow_\mathrm{A}})$, $(\mathrm{In}^{!_\mathrm{L}})$ and $(\mathrm{In}^{!_\mathrm{A}})$, respectively.

B. ADDITIONAL DEFINITIONS FOR $\pi^{\mathsf{LA}\mu}_{\mathsf{sec}}$

The *height* of types, $\mathsf{ht}(\cdot)$, is given as:

$$\mathsf{ht}((\vec{\tau})^p_s) = (\textstyle\sum_i \mathsf{ht}(\tau_i)) + 1$$
$$\mathsf{ht}([\&_i \vec{\tau}_i]^{\{p_i\}}_s) = \mathsf{ht}([\oplus_i \vec{\tau}_i]^{\{p_i\}}_s) = \max(\{\mathsf{ht}((\vec{\tau}_i)^{p_i}_s)\})$$

Then $\mathsf{ht}(A) = \max(\{\mathsf{ht}(A(x_i)) \mid x_i \in \mathsf{active}(A)\})$. In typing, $?^r_{\mathrm{L}\mu} A$ indicates all types in A have $?_\mathrm{L}$-modes but they can be mutable, while $?_\mathrm{L} A$ indicates $\mathsf{md}(A) = ?_\mathrm{L}$ as well as no types in A are mutable. With this convention, the typing rules are those in Figure 2 and the rule for reference in § 4.3, together with the following rules for mutable replicated input/output.

$$\dfrac{(\mathsf{In}^{!_{\mathrm{L}\mu}})\quad s \sqsubseteq \mathsf{tamp}(A \otimes B) \qquad \mathsf{ht}((\vec{\tau})^{!_{\mathrm{L}\mu}}_s) \gneq \mathsf{ht}(B) \qquad \mathsf{ht}((\vec{\tau})^{!_{\mathrm{L}\mu}}_s) = \mathsf{ht}(A(x)) \quad (\forall x \in \mathsf{fn}(A)) \qquad \vdash P \triangleright \vec{y}:\vec{\tau} \otimes ?_{\mathrm{L}\mu}?^r_{\mathrm{L}\mu} A^{\text{-}x} \otimes ?_{\mathrm{L}\mu}?^r_{\mathrm{L}\mu} B^{\text{-}x} \otimes ?_\mathrm{L}?_\mathrm{A} C^{\text{-}x}}{\vdash !\, x(\vec{y}).P \triangleright (x:(\vec{\tau})^{!_{\mathrm{L}\mu}}_s \to A) \otimes B \otimes C}$$

$$\dfrac{(\mathsf{Out}^{?_{\mathrm{L}\mu}})\quad \mathsf{active}(B) = \{\vec{y}\} \qquad \mathsf{ht}((\vec{\tau})^{?_{\mathrm{L}\mu}}_s)) \gneq \mathsf{ht}(E) \qquad \vdash P \triangleright B\langle \vec{y}:\vec{\tau}\rangle^{\text{-}x} \otimes C \quad B/\vec{y} = ?_{\mathrm{L}\mu}?^r_{\mathrm{L}\mu} E \otimes ?_\mathrm{L}?_\mathrm{A} F}{\vdash \overline{x}(\vec{y})P \triangleright (x:(\vec{\tau})^{?_{\mathrm{L}\mu}}_s \otimes E \otimes F) \odot C}$$

The rules for mutable branching/selection are similar.

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

$$\llbracket x : T \rrbracket_u \stackrel{\mathrm{def}}{=} [u \to x]^{T^\circ}$$

$$\llbracket () : \mathtt{unit} \rrbracket_u \stackrel{\mathrm{def}}{=} !u(x).\overline{x}$$

$$\llbracket \lambda x_0 {:} T_0.M : T_0 \Rightarrow T \rrbracket_u \stackrel{\mathrm{def}}{=} !\, u(x_0\vec{x}z).(\boldsymbol{\nu}\, u')(\llbracket M : T \rrbracket_{u'} \mid \mathsf{Arg}\langle u'\vec{x}z\rangle^{T})$$

$$\llbracket MN : T \rrbracket_u \stackrel{\mathrm{def}}{=} !\, u(\vec{x}z).(\boldsymbol{\nu}\, mx_0)(\llbracket M : T_0 \Rightarrow T \rrbracket_m \mid \llbracket N : T_0 \rrbracket_{x_0} \mid \mathsf{Arg}\langle mx_0\vec{x}z\rangle^{T_0 \Rightarrow T})$$

$$\llbracket \mathtt{inl}(M) : T_1 + T_2 \rrbracket_u \stackrel{\mathrm{def}}{=} !\, u(c).\overline{c}\mathtt{in}_1(m)\llbracket M : T_1 \rrbracket_m$$

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

$$\llbracket \mathtt{bind}\ x = N : T'\ \mathtt{in}\ M : T \rrbracket_u \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, x)(\llbracket M : T \rrbracket_u \mid \llbracket N : T' \rrbracket_x)$$

$$\llbracket \mathtt{lift}(M) : \llcorner T \lrcorner_s \rrbracket_u \stackrel{\mathrm{def}}{=} !u(c).\overline{c}(m)\llbracket M : T \rrbracket_m$$

$$\llbracket \mathtt{sec}\ x = N : T'\ \mathtt{in}\ M : T \rrbracket_u \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, xn)(\llbracket M : T \rrbracket_u \mid \llbracket N : T' \rrbracket_n \mid \mathsf{Lift}\langle xn\rangle^{T'})$$

$$\llbracket \mu x : T.M : T \rrbracket_u \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, m)([u \to m]^{T^\circ} \mid \llbracket M : T \rrbracket_m \mid [x \to m]^{T^\circ})$$

$$\overline{x}\langle \vec{y}\rangle^{\vec{\tau}} \stackrel{\mathrm{def}}{=} \overline{x}(\vec{z})\Pi_i[z_i \to y_i]^{\overline{\tau_i}} \qquad (\mathsf{md}(\tau_i) \in \mathcal{M}_{\uparrow} \cup \mathcal{M}_{?})$$

$$\mathsf{Arg}\langle x\vec{y}z\rangle^{[T_1..T_n\gamma]} \stackrel{\mathrm{def}}{=} \overline{x}\langle \vec{y}z'\rangle^{\overline{T_1^\circ}..\overline{T_n^\circ}\gamma^\bullet}$$

$$\mathsf{Sum}\langle l\vec{z}, (x_i)M_i\rangle^{T} \stackrel{\mathrm{def}}{=} \overline{l}(c)c[\&_{1,2}(x_i).(\boldsymbol{\nu}\, m)(\llbracket M_i : T \rrbracket_m \mid \mathsf{Arg}\langle m\vec{z}\rangle^{T})]$$

$$\mathsf{Lift}\langle xn\rangle^{T} \stackrel{\mathrm{def}}{=} !x(\vec{z}w).\overline{n}(c)c(m).\mathsf{Arg}\langle m\vec{z}w\rangle^{T}$$

$$[x \to x']^{[\&_i\vec{\tau}_i]^{\downarrow_{\mathrm{L}}}} \stackrel{\mathrm{def}}{=} x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i(\vec{y'}_i)\Pi_{ij}[y'_{ij} \to y_{ij}]^{\overline{\tau_{ij}}}]$$

$$[x \to x']^{[\&_i\vec{\tau}_i]^{!_{\mathrm{L}}}} \stackrel{\mathrm{def}}{=} !x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i(\vec{y'}_i)\Pi_{ij}[y'_{ij} \to y_{ij}]^{\overline{\tau_{ij}}}]$$

The copy-cats of unary types are defined by reducing the indexing sets to singletons. Similarly for the copy cats of affine types.

Figure 8: Encoding of Dependency Core Calculus

$$\langle x : T \rangle_u \stackrel{\mathrm{def}}{=} \overline{u}\langle x\rangle^{\overline{T^\circ}}$$

$$\langle n : \mathbb{N}_s \rangle_u \stackrel{\mathrm{def}}{=} \overline{u}(c)\llbracket n \rrbracket_c$$

$$\langle \lambda x : S.M : S \Rightarrow T \rangle_u \stackrel{\mathrm{def}}{=} \overline{u}(c)!\, c(xm).\langle M \rangle_m$$

$$\langle MN : T \rangle_u \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, m)(\langle M : S \Rightarrow T \rangle_m \mid m(c).(\boldsymbol{\nu}\, n)(\langle N : S \rangle_n \mid \overline{n}\langle eu\rangle^{\overline{S^\circ}T^\bullet})$$

$$\langle \mathtt{sec}\ x = N\ \mathtt{in}\ M : T \rangle_u^E \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, n)(\langle N : T' \rangle_n \mid n(x).\langle M : T \rangle_u)$$

$$\langle \mu x : U.M : U \rangle_u \stackrel{\mathrm{def}}{=} \begin{cases} C[\overline{u}(c)(\boldsymbol{\nu}\, x)(P_i \mid [x \to c]^{U^\circ})]_i & (x \notin \mathsf{fn}(C)) \\ (\boldsymbol{\nu}\, m)(\Omega_m^{(U^\bullet)^{!_{\mathrm{A}}}} \mid \overline{m}\langle u\rangle^{U^\bullet}) & (x \in \mathsf{fn}(C)) \end{cases}$$

$$(\langle M : U \rangle_u \stackrel{\mathrm{def}}{=} C[\overline{u}(c)P_i]_i)$$

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

$\langle \lambda x : T_1.M : U_1 \rightharpoonup U_2 \rangle_u$ and $\langle \mathtt{bind}\ x = N\ \mathtt{in}\ M \rangle_u$ are the same as $\langle \lambda x : S.M : S \Rightarrow T \rangle_u$ and $\langle \mathtt{seq}\ x = N\ \mathtt{in}\ M \rangle_u$, respectively.

Figure 9: Encoding of Call-by-Value DCC

(Expression)

$$\langle n \rangle_u^E \stackrel{\mathrm{def}}{=} \overline{u}(c)\llbracket n \rrbracket_c \qquad \langle y \rangle_u^E \stackrel{\mathrm{def}}{=} \overline{u}\langle y\rangle^{\overline{T^\circ}} \quad (E \vdash y : T)$$

$$\langle \lambda x.e \rangle_u^E \stackrel{\mathrm{def}}{=} \overline{u}(c)!c(xn).\langle e \rangle_m^E \qquad \langle \mathtt{succ}(y) \rangle_u^E \stackrel{\mathrm{def}}{=} \overline{u}(c)!c(e).\overline{y}\mathtt{inl}(c)c[\&_i\overline{e}\mathtt{in}_{i+1}] \qquad \langle c\ \mathtt{return}\ e \rangle_u^E \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, c)(\llbracket c \rrbracket_c^E \mid c.\langle e \rangle_u^E)$$

$$\langle e_1 e_2 \rangle_u^E \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, c_1)(\langle e_1 \rangle_{c_1}^E \mid c_1(y_1).(\boldsymbol{\nu}\, c_2)(\langle e_2 \rangle_{c_2}^E \mid c_2(y_2).\overline{y_1}\langle y_2 u\rangle^{\overline{T'^\circ}T^\bullet}))$$

(Command)

$$\llbracket \mathtt{skip} \rrbracket_u^E \stackrel{\mathrm{def}}{=} \overline{u}$$

$$\llbracket x := v \rrbracket_u^E \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, m)(\langle v \rangle_m^E \mid m(w).\overline{x}\mathtt{in}_2\langle wu\rangle^{\overline{T^\circ}()^{\uparrow_{\mathrm{L}}}}) \qquad (E \vdash v : T)$$

$$\llbracket c_1 ; c_2 \rrbracket_u^E \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, e)(\llbracket c_1 \rrbracket_e^E \mid e.\llbracket c_2 \rrbracket_u^E)$$

$$\llbracket \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \rrbracket_u^E \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu} xe)(\langle v \rangle_x^E \mid \overline{x}(z)z[\&_i\llbracket c_i \rrbracket_e^E] \mid e.\overline{u}) \quad (c_i = c_2 \text{ if } i \geq 2)$$

$$\llbracket \mathtt{while}\ !x\ \mathtt{do}\ c \rrbracket_u^E \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, e)(\overline{e}(k)k.\overline{u} \mid [f \to e] \mid !e(k).\overline{x}\mathtt{inl}(c)c(y).\overline{y}(z)z[(\boldsymbol{\nu}\, l)(\llbracket c \rrbracket_l^E \mid l.\overline{f}(k').k'.\overline{k})\&\overline{k}])$$

$$\llbracket \mathtt{let}\ x = e\ \mathtt{in}\ c \rrbracket_u^E \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, c)(\langle e \rangle_c^E \mid u(x).\llbracket c \rrbracket_u^E)$$

$$\llbracket \mathtt{new}\ x \mapsto y\ \mathtt{in}\ c \rrbracket_u^E \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, x)(\mathsf{ref}\langle xy\rangle^{T^\circ} \mid \llbracket c \rrbracket_u^E) \quad (E \vdash y : T)$$

$$\llbracket \mathtt{let}\ x = !y\ \mathtt{in}\ c \rrbracket_u^E \stackrel{\mathrm{def}}{=} \overline{y}\mathtt{inl}(e)e(x).\llbracket c \rrbracket_u^E$$

$$\llbracket \Pi_i c_i \rrbracket_{\vec{u}}^E \stackrel{\mathrm{def}}{=} \Pi_i\llbracket c_i \rrbracket_{u_i}^E$$

Figure 10: Encoding of VS-Calculus with Reference and Procedure