Channel Dependent Types for Higher-Order Mobile Processes

(Extended Abstract)

Nobuko Yoshida
Imperial College London

ABSTRACT

We introduce a new expressive theory of types for the higher-order π-calculus and demonstrate its applicability via non-trivial security analyses of a simple class-based language with distributed code mobility. The new theory significantly improves our previous one presented in [52] by the use of *channel dependent/existential types*. New dependent types control dynamic change of process accessibility via channel passing, while existential types guarantee safe scope-extrusion in higher-order process passing. This solves an open issue in [52], leading to significant enlargement of original typability. Two basic security concerns for mobile computation, secrecy for data confidentiality and access controls for authorised resources are analysed in a uniform type-based static framework, culminating in the noninterference theorem and authority-error freedom in the presence of higher-order code mobility. The generality and expressiveness of the new type discipline are tested with a sound embedding of multi-threaded class-based language with dynamic code/class distribution, enforcing secrecy and accessibility.

1. INTRODUCTION

Background. Nowadays the use of mobile code is widespread throughout computing scenes, from intra-net applications in corporation to migrating code in active network to dynamic content in the world-wide web. One of the principal benefits of mobile code (as found in Java [13, 25] and CLR [29, 8]) is to allow *extensibility* [12, 45], where a piece of code migrates from a source node to a target node and gets linked to the run-time environment of the target, to serve its purposes. Such extensibility leads to rich repertoire of new functionalities, including dynamic, tightly-coupled use of remote computational resources not restricted by network bandwidth/latency, multi-media applications where real-time interaction with resources are essential, and incremental software/service addition/update. It also poses a new challenge in software safety: the infrastructure is required to manage security mechanisms by which pieces of code with different origins and functionalities, embodying different principals, can safely interact in the presence of mobility and dynamic linkage. One of the promising analytical tools in this regard is the use of formal calculi of computation ensuring key security concerns such as integrity and confidentiality [1, 2, 11, 36, 20, 22, 14, 33, 42, 44, 53].

The higher-order π-calculus [38] is a general formalism of interaction in which two kinds of mobility, name passing and process passing, are integrated in a simple, universal form. In the untyped (or weakly typed) setting, it is one of the most well-studied and understood formalisms for mobility, starting from Sangiorgi's thesis [38] and reaching a recent work by Jeffrey and Rathke [26]. Inheriting from CCS and the π-calculus, its dynamics centres on interaction at channels, where processes communicate not only (possibly new) channel names, as in the π-calculus, but also *parameterised processes*, i.e. processes that can be instantiated by names and other processes — just like a piece of mobile code is instantiated with local capability after migration. This addition to the already high expressiveness of the π-calculus, offers a powerful basis for describing, analysing and understanding mobile computation, since moving data and behaviour from one place (context) to another place (context) is a key feature of general mobility. The simplicity and generality of the formalism make it a promising candidate as a core formalism for modelling distributed mobile computation. One issue when considering the applicability, however, is the lack of expressive type structure for the calculus.

Types play an important rôle when applying computational calculi such as the λ-calculus to the study of programming languages and systems. Not only many modern programming languages themselves are richly typed, but also types are the basis of various program analysis/reasoning methods. In the context of language-based security, type soundness gives foundations for various security mechanisms by offering unforgeable references to resources. Further, in the context of the π-calculus, strong type structures are essential for the precise modelling of language dynamics [5, 50, 53, 32, 47, 20, 17, 6]. The λ-calculus has function-based notions of types, while the π-calculus has name-based notions of types, both with rich theories and powerful applications [33, 11, 36, 2]. Being a combination of these two formalisms, can we find a basic type structure for the higher-order π-calculus that would serve as a basis of modelling, analysing and developing extensible language infrastructure?

This paper introduces a new theory of types that can assign expressive interface types to higher-order mobile processes, and demonstrates its applicability in two areas of security in mobile code. The theory is a significant improvement on the author's preceding work with Matthew Hennessy [52, 51], leading to a general typed formalism, which allows precise modelling and analyses of statics and dynamics of programming languages with code distribution. In the heart of the formalism is a new form of dependent types, called *channel dependent types*, that can represent, intuitively speaking, communication of typed anonymous names in carried code. Let us illustrate the key ideas with simple examples, starting from a brief review of our previous work in [51].

Assigning Types to Processes. We use the call-by-value version of the higher-order π-calculus, λπ [52, 51]. In the calculus, not only names but also arbitrary higher-order abstracted processes are passed around along communication channels. As a simple example, a thunked process $\lambda(x{:}\mathtt{unit}).P$, abbreviated to $\ulcorner P \urcorner$, can be sent and received as a value. The following expresses a server which repeatedly receives thunked processes via port a.

$$\mathsf{Server}(a) \stackrel{\text{def}}{=} !a(y{:}\tau).\mathit{run}\ y \tag{1}$$

Here *run* stands for a function $\lambda y.y()$ which executes a thunked process by applying the unit $()$. Since accepting arbitrary processes for execution obviously breaks access control of local resources [13, 25], one might wish to restrict the behaviour of incoming code so that it can only access some specified channels.

In [52], Matthew Hennessy and the present author introduced a type discipline which can control the effect of migrating code, by assigning a different type to each process depending on its intended use, so that a process can use a typed inputting channel (τ at a in (1) above) to detect, for example, malicious behaviour of received code via static type checking. A type is given as a finite channel environment Δ, prescribing channel usage of each process.

$$\Gamma \vdash P\ :\ \Delta \tag{2}$$

This judgement means "P accesses channels at most as specified by Δ under environment Γ". For example, under appropriate $\Gamma \supset \{b{:}(\mathtt{nat})^{\mathtt{0}}, c{:}(\mathtt{nat})^{\mathtt{0}}\}$ (where $b{:}(\mathtt{nat})^{\mathtt{0}}$ means a channel b can output values of type `nat`), a message carrying a natural number may be assigned a different type depending on its destination.

$$\Gamma \vdash \overline{b}\langle 1\rangle :\ b{:}(\mathtt{nat})^{\mathtt{0}} \quad \text{and} \quad \Gamma \vdash \overline{c}\langle 1\rangle :\ c{:}(\mathtt{nat})^{\mathtt{0}} \tag{3}$$

Now the following indicates a server only accepts a process which accesses at most the specified resource.

$$!a(y{:}\ulcorner b{:}(\mathtt{nat})^{\mathtt{0}}\urcorner).\mathit{run}\ y$$

where $\ulcorner\Delta\urcorner$ stands for an abbreviation of $\mathtt{unit} \to \Delta$. Using the type system in [52], one can check $\mathsf{Server}(a)\,|\,\overline{a}\langle\ulcorner\overline{b}\langle 1\rangle\urcorner\rangle$ is typable while $\mathsf{Server}(a)\,|\,\overline{a}\langle\ulcorner\overline{c}\langle 1\rangle\urcorner\rangle$ is not. Note that if we use the non-differentiating constant type `proc` (also written *o* or *ok* in the literature), as found in all work [38, 51, 26, 27, 39] before [52], τ can be assigned only to $\ulcorner\mathtt{proc}\urcorner$, hence parallel composition with both messages in (3) become only typable with type `proc`, thus losing all type information.

Issue: Scope Opening and `proc` ***Type.*** The typing system in [52] can assign non-trivial types to a variety of forms of higher-order code; however it has one basic limitation, in that it cannot assign a non-trivial type when a process demonstrates, in higher-order code passing, *scope extrusion*, i.e. sending a local name across a local boundary (which is a common practice in standard distributed object-oriented programming, see §.5). Consider the following sender to the above server, which wishes to get the acknowledgement $\overline{\mathtt{ack}}$ at the time P is executed in the remote site. Naturally, this acknowledgement will be sent via a private channel to ensure its safe delivery.

$$\mathsf{Send}(a) \stackrel{\text{def}}{=} (\nu\mathtt{ack})(\overline{a}\langle\ulcorner P\,|\,\overline{\mathtt{ack}}\urcorner\rangle\,|\,\mathtt{ack}.Q) \tag{4}$$

Here $\overline{\mathtt{ack}}$ is a trigger carrying the empty vector, hence typed by $\sigma = ()^{\mathtt{0}}$. But how can we type a when $\overline{\mathtt{ack}}$ is restricted? In the system in [52], the only possible typing for a is $(\ulcorner\mathtt{proc}\urcorner)^{\mathtt{0}}$, using the most uninformative type `proc`, so that, for composition with $\mathsf{Send}(a)$, $\mathsf{Server}(a)$ can only be assigned the type `proc`. But if we wish to control these processes by types following a natural specification, one *should* be able to say a can output a thunked process which contains some anonymous channel of type $()^{\mathtt{0}}$, together with the type of P. A general solution to this issue is difficult due to a subtle interplay between a process type and a name hiding: in particular, an ad hoc solution such as erasing the type of a from the typing leads to an inconsistent system (see [52] for more discussions). A related case is non-deterministic name instantiation:

$$e(x).\overline{a}\langle\ulcorner\overline{x}\langle 1\rangle\urcorner\rangle\,|\,\overline{e}\langle e_1\rangle\,|\,\overline{e}\langle e_2\rangle \tag{5}$$

Note this process includes another sender to $\mathsf{Server}(a)$ under input prefix. Since we cannot decide which of e_1 and e_2 may be instantiated, the system in [52] can only assign $(\ulcorner\mathtt{proc}\urcorner)^{\mathtt{0}}$ to a, hence $\mathsf{Server}(a)$ again ends up with `proc`.

Channel Dependent Types. In this paper we introduce *channel dependent types* as an effective solution to this problem. As the name suggests, these types introduce quantification on names rather than on types (as in standard existential types). For example, if we assume a type of P is Δ, the new typing system assigns to the channel a of $\mathsf{Server}(\mathtt{a})$ in (1), the following type:

$$(\Pi[x{:}\sigma]\ulcorner x{:}\sigma,\Delta\urcorner)^{\mathtt{I}}$$

Here the occurrences of x in the thunked type $\ulcorner x{:}\sigma,\Delta\urcorner$ are bound by the preceding dependent quantifier. The type indicates that a channel inputs a thunk process which contains some unknown channel name of type σ and the known channels specified by Δ. When the communication is involved, a parameter is dynamically instantiated into x in $\ulcorner x:\sigma,\Delta\urcorner$ at the time of code passing, just as in function application of channel dependent types (which essentially correspond to type instantiation in reduction of explicitly typed λ-calculi with dependent/polymorphic types). As we shall see in the main sections, the server (1) can now be given a non-trivial type $\{a:(\Pi[x:\sigma]\ulcorner x:\sigma,\Delta\urcorner)^{\mathtt{I}},\Delta\}$. A special case of channel dependent types is also worth mentioning, called *channel existential types*, where an abstracted name is ensured not to be touched by the receiver. Since this requirement is met in the above server example, it can also be typed as $(\exists[x:\sigma]\ulcorner x:\sigma,\Delta\urcorner)^{\mathtt{I}}$. Many examples, including an example in (5), can be typed with existential types, which satisfy strong properties such as a type erasure theorem and a response property of higher-order code with hidden names.

The introduction of channel dependent/existential types dramatically increases the expressiveness of process types, opening a broad range of applications. The type `proc` is no longer necessary for typing well-behaved agents. As in the first-order (name passing) setting [5, 6, 16, 28, 48, 49, 50, 53], assigning appropriate types to extruded names is crucial for having significant behavioural constraint by, and based on, types. Since it allows organised treatment of hidden channels, the resulting type discipline as a whole becomes simpler. It also leads to a novel type-preserving encoding of higher-order processes into the π-calculus, an effective tool for reasoning.

We next present a brief preview of a series of its applications starting from the integration of linear/affine types [5, 6, 20, 49, 50, 53] which enables the precisely modelling of the statics and dynamics of programming language constructs, either local or distributed.

Integration with Linear/Affine Types. Ensuring linear/affine usage of resources [28, 16, 5, 6, 50, 53, 22, 43, 48] is important in both processes and functions. The notion also works in harmony with channel dependently typed higher-order mobile processes. Recall the previous $\mathsf{Sender}(a)$ in (4). In a refined typing, the sender may wish to specify $\overline{\mathtt{ack}}$ is used *at most once*, which is a natural constraint for remote communication such as RMI and is also useful for resource usage prediction. We may call this name discipline *affine* [5], and denote the type of this trigger as $()^{\uparrow}$. For example, in the following processes:

$$\mathsf{Server}(a)\,|\quad \overline{a}\langle\ulcorner P_1\,|\,\overline{\mathtt{ack}}\urcorner\rangle\,|\,\overline{a}\langle\ulcorner P_2\,|\,\overline{\mathtt{ack}}\urcorner\rangle \tag{6}$$

the server violates affinity since it should repeatedly accept an affine message $\overline{\mathtt{ack}}$. However if we *hide* $\overline{\mathtt{ack}}$ at the sender side:

$$\begin{array}{lll} \mathsf{Server}(a) & | & (\nu\,\mathtt{ack})(\overline{a}\langle \ulcorner P_1 \,|\, \overline{\mathtt{ack}} \urcorner\rangle \,|\, \mathtt{ack}.Q_1) \\ & & (\nu\,\mathtt{ack})(\overline{a}\langle \ulcorner P_2 \,|\, \overline{\mathtt{ack}} \urcorner\rangle \,|\, \mathtt{ack}.Q_2) \end{array} \quad (7)$$

this process should be typable with affine types, since hidden $\mathtt{ack}$ now denote distinct names. This becomes possible by the use the new dependent types, so that it now has the type $\{a:(\Pi[x:()^{\uparrow}]\ulcorner x:()^{\uparrow},\Delta\urcorner)^{!},\Delta\}$ with Δ a type of $P_{1,2}$ (! indicates a is replicated, § 3).

Affinity is useful for guaranteeing the unicity of channels and resources: a stronger constraint, *linearity*, has also a benefit when a precise embedding of language constructs is necessary. A general integration of linear/affine types in the dependently typed $\lambda\pi$ will be shown in §.3, and is essential for security analyses, which we discuss next.

Secrecy and Mobility. Secrecy in general means a program does not transform high-level information to effects at low-level channels [10, 2, 14, 20, 31, 42]. In the context of processes, interaction carries information, so, for example, a process $a^{\mathbb{H}}.\overline{b}^{\mathbb{L}}$ (where we attach secrecy labels to channels) first has secret interaction at a, then less secret interaction at b, which may be known by a wider public. Then this process is insecure, since any process observing b, which can be done by low-level process, has the possibility to know interaction at a, which should be kept secret. In the presence of higher-order mobility, we can consider a process carrying another process which is assigned some secrecy level:

$$P \equiv \overline{a}^{\mathbb{H}}\langle \ulcorner \overline{b}^{\mathbb{L}} \urcorner\rangle \quad (8)$$

where a thunk of a low level program $\ulcorner \overline{b} \urcorner$ is transfered via the high channel a. In the fine-grained types, the safety of this code depends on the receiver. Let us consider the following two receivers:

$$\begin{array}{lll} Q_1 & \equiv & a^{\mathbb{H}}(x:\tau).\mathit{run}\ x \mid b^{\mathbb{L}}.\overline{c}^{\mathbb{L}} \\ Q_2 & \equiv & a^{\mathbb{H}}(x:\tau).((\lambda y.\mathbf{0})x) \mid b^{\mathbb{L}}.\overline{c}^{\mathbb{L}} \end{array} \quad (9)$$

Note Q_1 does use the thunk while Q_2 does not. Since our fine-grained process typing can detect this distinction, the secrecy analysis based on them can indicate Q_2 is safe while Q_1 is not. This offers more flexible and host-oriented protection than deciding the safety/unsafety at the sender side, without looking at the receiver, presenting a general and clear framework for secrecy assurance. In §.4, we shall clarify how this framework leads to greater typability than some of the possible alternatives.

Access Control and Mobility. The access control, a traditional subject in security, is a means to control execution of operations on sensitive resources by principals following a specified (and often changeable) security policy. In the context of extensible architecture, one basic topic is how to set a current authority level of a thread depending on which pieces of code (coming from different origins) it has executed [45, 46, 12, 3]. Recently several analyses of this concept appear, using the λ-calculus as a central formalism [11, 36]. In this paper we show how the typed linear/affine $\lambda\pi$ can precisely model access control mechanisms in the presence of code mobility, concurrency and distribution, including nested instantiation of channels and code. Due to interaction with resources managed by OSs and databases, as well as policy change, self-contained static checking of mobile code in this regard would be generally infeasible. Yet when it is possible, it can lead to substantial optimisation [11]. Further, the $\lambda\pi$ allows clean modelling of different access control mechanisms for extensible architecture [45, 46, 36, 11, 12, 3], which scales to a simple but realistic language with code mobility leading to new insights on this topic.

Summary of Contributions. We summarise the main technical contributions of the present work.

1. Introduction of channel dependent/existential types for the $\lambda\pi$-calculus, a higher-order π-calculus first presented in [52], significantly enlarging typability and offering a general basis for applications. Type-safety is proved by extending the technique developed in [52].
2. Consistent integration of linear/affine type disciplines [50, 5, 53, 20] into the resulting typed calculus, allowing the calculus to be used for refined behavioural modelling and analysis of programming languages.
3. Type-based analyses of two basic security concerns in mobile code, secrecy [10, 2, 14, 20, 31, 42] and access control of resources [3, 11, 12, 36, 45, 46], both using the typed formalism. We prove noninterference and authority-error freedom for well-typed processes in the presence of higher-order code mobility.
4. An application of these analyses to a simple multi-threaded class-based language with distributed communication and code mobility [24, 7, 44, 20]. The type disciplines for secrecy and access control are developed on the basis of those of the $\lambda\pi$-calculus. Correctness (type-safety, non-interference and the lack of authority error) of the language is proved via the type-preserving embedding into the $\lambda\pi$.

To our knowledge, our formalism is the first to offer a general framework for describing and analysing secrecy and access control in higher-order mobile computing with precision so as to be directly applicable to the analysis of these concerns in major language constructs with code mobility.

Structure of the Paper. Section 2 introduces the $\lambda\pi$ with channel dependent/existential types. Section 3 shows a consistent integration of the linear/affine types. Section 4 discusses modelling and analysis of secrecy and access control in the typed calculus. Section 5 demonstrates an application to a mobile programming language. Section 6 discusses related work and further issues. Due to the space limitation, auxiliary definitions and detailed examples are relegated to the separated appendix [54] which is available from `http://www.doc.ic.ac.uk/~yoshida`.

2. PROCESSES AND TYPES

2.1 Processes

The syntax of processes [51, 52] is given in Figure 1, which is the π-calculus augmented with the call-by-value λ-calculus primitives. We assume an infinite set of *channels* or *names* $\mathbf{N}$, ranged over by $a,b,...$, and an infinite set of *variables* $\mathbf{V}$, ranged over by $x,y,...$ τ and σ denote value and channel types which will be given later. Type annotations are often omitted unless they are relevant and we often write $\overline{a}(c)P$ for the abbreviation for $(\nu\,c)(\overline{a}\langle c\rangle \,|\, P)$.

The reduction rules of $\lambda\pi$ is a simply combination of those of the π-calculus and the call-by-value λ-calculus. The main reduction rules of $\lambda\pi$ are the following β-reduction rule and communication rules.

$$\begin{array}{ll} (\beta) & (\lambda(x:\tau)P)\,V \longrightarrow P\{V/x\} \\ (\mathrm{com}) & a(x_1:\tau_1,...,x_n:\tau_n).P \mid \overline{a}\langle V_1,...,V_n\rangle \\ & \qquad \longrightarrow P\{V_1,...,V_n/x_1,...,x_n\} \\ (\mathrm{rep}) & !a(x_1:\tau_1,...,x_n:\tau_n).P \mid \overline{a}\langle V_1,...,V_n\rangle \\ & \qquad \longrightarrow P\{V_1,...,V_n/x_1,...,x_n\} \,|\, !a(x_1:\tau_1,...,x_n:\tau_n).P \end{array}$$

(Term)

$P,Q,\ldots$	$::=$	V	value
	$\mid$	$\mathbf{0}$	nil
	$\mid$	$P \mid P$	parallel
	$\mid$	$\overline{u}\langle V_1,\ldots,V_n\rangle$	output
	$\mid$	$u(x_1:\tau_1,\ldots,x_n:\tau_n).P$	input
	$\mid$	$!u(x_1:\tau_1,\ldots,x_n:\tau_n).P$	replicator
	$\mid$	$(\nu a:\sigma)P$	restriction
	$\mid$	PP	application

(Identifier)

$u,v,w,\ldots$	$::=$	$x,y,z,\ldots$	variable
	$\mid$	$a,b,c,\ldots$	channel

(Value)

$V,W,\ldots$	$::=$	$u,v,w,\ldots$	identifier
	$\mid$	$(),1,2,..$	unit, numbers, ..
	$\mid$	$\lambda(x:\tau)P$	abstraction

(**Abbreviation**)

$\ulcorner P \urcorner$	$\stackrel{\text{def}}{=}$	$\lambda(x:\mathtt{unit})P$ $\quad x \notin \mathsf{fv}(P)$		thunk
run	$\stackrel{\text{def}}{=}$	$\lambda(x:\mathtt{unit}\to\Delta)x()$		run

Figure 1: Syntax

Other rules are standard using the structural equivalence (denoted by $\equiv$), hence omitted (cf. [52, Table 4]). We also denote multi-step reduction as $\twoheadrightarrow \stackrel{\text{def}}{=} \equiv \cup \longrightarrow^*$. Note that substitution by channels $P\{u/x\}$ affects not only terms but also types which annotate bound variables.

2.2 Types

Syntax. The grammar of types are given in Figure 2. These types are from [51, 52] except for the introduction of channel dependent types. We use a simpler form of IO-modes based on duality [5, 20, 17, 16, 20, 50]. There is `unit`, `nat`, .., higher-order function type $\tau\to\tau'$, (functional) name dependent type $\Pi(x:\sigma)\tau$, process type Δ and channel type σ. The last one incorporates dependent quantification on which we shall discuss soon. In $\Pi(x:\sigma)\tau$, we allow the type τ to contain occurrences of the channel variable x; then x in τ is bound. A process type Δ, assigned to a process, is a mapping from a finite subset of $\mathbf{N}\cup\mathbf{V}$ to channel types. Unlike in [51, 52], we no longer have `proc` types, which are now unnecessary due to the introduction of channel dependent types.

Channel Dependent Types. A channel type incorporates *dependent quantification*, and has the form $(\Pi[x_1:\sigma_1,\ldots,x_n:\sigma_n]\tau_1,\ldots,\tau_m)^p$. This indicates:

- A channel is used with IO-modes p, either $\mathtt{I}$ (input) or $\mathtt{O}$ (output).
- A channel carries a vector of channels typed by $\sigma_1,..,\sigma_n$ and a vector of higher-order values typed by $\tau_1,..,\tau_m$; and
- Free occurrences of x_i in $\sigma_{i+1},\ldots,\sigma_n$ as well as $\tau_1,\ldots,\tau_m$ are bound occurrences, just as in a functional name dependent type $\Pi(x_1:\sigma_1)(\Pi(x_2:\sigma_2)...(\Pi(x_n:\sigma_n)\tau_i)...)$.

By the abbreviation in Figure 2, we can write, for example, $(\sigma\tau)^p$ for $(\Pi[x:\sigma]\tau)^p$ if x does not occur in τ. This is similar to the standard convention in the functional dependent type where $\mathtt{nat}\to\mathtt{nat}$

Value:	$\tau ::=$	$\mathtt{unit} \mid \mathtt{nat} \mid \tau\to\tau' \mid \Pi(x:\sigma)\tau \mid \Delta \mid \sigma$
Channel:	$\sigma ::=$	$(\Pi[x_1:\sigma_1,\ldots,x_n:\sigma_n]\tau_1,\ldots,\tau_m)^p \mid \langle\sigma,\sigma'\rangle$
Environment:	$\Delta ::=$	$\emptyset \mid \Delta,x:\tau \mid \Delta,u:\sigma$

Abbreviation:

$$(\Pi[x_1:\sigma_1,\ldots,x_n:\sigma_n]\tau_1,\ldots,\tau_m)^p \stackrel{\text{def}}{=} (\sigma_1,\ldots,\sigma_n,\tau_1,\ldots,\tau_m)^p \quad (\text{if } x_1,\ldots,x_n \notin \mathsf{fv}(\sigma_1,\ldots,\sigma_n,\tau_1,\ldots,\tau_m))$$

thunk: $\ulcorner\Delta\urcorner \stackrel{\text{def}}{=} \mathtt{unit}\to\Delta$

Figure 2: Types

is a degenerate case $\Pi x:\mathtt{nat}.\mathtt{nat}$. Thus we can write simple channel types (without dependency) using the standard notation.

Let $(\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^p$ be $(\Pi[x_1:\sigma_1,\ldots,x_n:\sigma_n]\tau_1,\ldots,\tau_m)^p$. If $m=0$, this becomes $(\Pi[\vec{x}:\vec{\sigma}])^p$ (where only channels are carried); while if $n=0$, it becomes $(\vec{\tau})^p$ (where only higher-order values are carried); and if both are 0 then it becomes $()^p$ (where nothing is carried). $\langle\sigma,\sigma'\rangle$ indicates the unordered *pair type*, one is input and the other is output. The mode of $\langle\sigma,\sigma'\rangle$ is defined as $\mathtt{IO}$ (input and output). $\overline{\sigma}$, the *dual of* σ, is the result of dualising all action modes in σ.

The sets of free names/variables in term P, written $\mathsf{fn}(P)$, $\mathsf{fv}(P)$ and $\mathsf{fnv}(P)$, respectively, incorporate those occurring in annotating types. For example, we have $\mathsf{fv}(\lambda(x:\tau).P) = (\mathsf{fv}(\tau)\cup\mathsf{fv}(P))\setminus x$. Examples of types follow.

EXAMPLE 2.1. (types)

1. (simple IO-types) A channel which outputs a natural number has type $(\Pi[x:\mathtt{nat}])^{\mathtt{O}}$, which is written as $(\mathtt{nat})^{\mathtt{O}}$ using abbreviation in Figure 2. $(\mathtt{nat})^{\mathtt{I}}$ is its dual.
2. (higher-order type) A thunked process which can output `nat` at a has type $\ulcorner a:(\mathtt{nat})^{\mathtt{O}}\urcorner$. The higher-order identity for this type has type $\ulcorner a:(\mathtt{nat})^{\mathtt{O}}\urcorner\to\ulcorner a:(\mathtt{nat})^{\mathtt{O}}\urcorner$. A dependent function type which constructs such a thunked process from a given name has type $\Pi(x:(\mathtt{nat})^{\mathtt{O}})\ulcorner x:(\mathtt{nat})^{\mathtt{O}}\urcorner$.
3. (channel dependent type) A process which inputs output name b and thunked process $\ulcorner a:(\mathtt{nat})^{\mathtt{O}}\urcorner$ has type $((\mathtt{nat})^{\mathtt{O}},\ulcorner a:(\mathtt{nat})^{\mathtt{O}}\urcorner)^{\mathtt{I}}$. On the other hand, a process which inputs anonymous name a and its thunked process $\ulcorner a:(\mathtt{nat})^{\mathtt{O}}\urcorner$ has type $(\Pi[x:(\mathtt{nat})^{\mathtt{O}}]\ulcorner x:(\mathtt{nat})^{\mathtt{O}}\urcorner)^{\mathtt{I}}$. Both types input two arguments, one is a channel and another is a value. We then have $\overline{(\Pi[x:(\mathtt{nat})^{\mathtt{O}}]\ulcorner x:(\mathtt{nat})^{\mathtt{O}}\urcorner)^{\mathtt{I}}} = (\Pi[x:(\mathtt{nat})^{\mathtt{I}}]\ulcorner x:(\mathtt{nat})^{\mathtt{O}}\urcorner)^{\mathtt{O}}$.

Following [51, 52], we use a partial commutative operator $\sigma\odot\sigma'$ to control composition of processes. $\sigma\odot\sigma'$ is defined when σ and σ' are *coherent*, written $\sigma\asymp\sigma'$, which intuitively means the structures of σ and σ' (e.g. arity) are consistent (see [54] for formal definition). For example, we have $(\mathtt{nat})^{\mathtt{O}}\asymp(\mathtt{nat})^{\mathtt{I}}$, so $(\mathtt{nat})^{\mathtt{O}}\odot(\mathtt{nat})^{\mathtt{I}}$ is defined and has the value $\langle(\mathtt{nat})^{\mathtt{I}},(\mathtt{nat})^{\mathtt{O}}\rangle$. Also $(\mathtt{nat})^{\mathtt{O}}\asymp(\mathtt{nat})^{\mathtt{O}}$ and $(\mathtt{nat})^{\mathtt{O}}\odot(\mathtt{nat})^{\mathtt{O}}=(\mathtt{nat})^{\mathtt{O}}$. $\odot$ and $\asymp$ are pointwise extended to the composition of channel environments.

When the name u is substituted for x in a process type Δ, then the types σ of x and σ' of u are composed by $\odot$, resulting in $u:\sigma\odot\sigma'$. For example, we have:

$\{a:(\mathtt{nat})^{\mathtt{O}},x:(\mathtt{nat})^{\mathtt{I}}\}\{b/x\}=\{a:(\mathtt{nat})^{\mathtt{O}},b:(\mathtt{nat})^{\mathtt{I}}\}$ and
$\{a:(\mathtt{nat})^{\mathtt{O}},x:(\mathtt{nat})^{\mathtt{I}}\}\{a/x\}=\{a:\langle(\mathtt{nat})^{\mathtt{O}},(\mathtt{nat})^{\mathtt{I}}\rangle\}$,

while $\{a:(\mathtt{nat})^{\mathtt{O}},x:(\mathtt{unit})^{\mathtt{I}}\}\{a/x\}$ is undefined. We can apply the same rule to the substitution into terms [51, §.2]. For example, given $\lambda(x:(\mathtt{nat})^{\mathtt{O}}).\lambda(y:\ulcorner x:(\mathtt{nat})^{\mathtt{O}},b:(\mathtt{unit})^{\mathtt{O}}\urcorner).y$ we can observe:

$(\lambda(y{:}\ulcorner x{:}(\mathtt{nat})^{\mathtt{0}}, b{:}(\mathtt{unit})^{\mathtt{0}}\urcorner).y)\{a/x\}$ is defined but
$(\lambda(y{:}\ulcorner x{:}(\mathtt{nat})^{\mathtt{0}}, b{:}(\mathtt{unit})^{\mathtt{0}}\urcorner).y)\{b/x\}$ is undefined.
Our typing system shall ensure such an undefined substitution never happens during interaction between typable terms.

2.3 Typing with Channel Dependent Types

The typing system for $\lambda\pi$ is given in Figure 3, which uses the two kinds of judgements:

$\Gamma \vdash P{:}\tau$ a term P has a type τ under Γ
$\Gamma \vdash u{:}\sigma$ a channel u has a type σ under Γ

In the inference rules we also use judgements for the well-formdness of environments and types, for which the derivation rules are given in the standard manner (see [54]).

The inference rules, coming from [52] except (In) and (Out) rules which use new channel dependent types, are divided into three categories. The first group, **(Common)**, is standard rules for variables, channels and constant. The second one, **(Function)**, comes from the simply typed λ-calculus (where the rules for channel abstraction, (Abs_N) and (App_N), and those for the higher-order abstractions, (Abs_H) and (App_H), are separately given). The final group, **(Process)**, are about processes. In (Nil), we start from the empty interface, and in (Par), we merge two interfaces together. The rule (Weak) corresponds to the process subsumption rule in [52]; since $\Gamma \vdash P : \Delta$ means "P would access channels specified at most by Δ", we can increment its interface. (Res) hides a name.[1] We shall now illustrate the two main rules, (In) and (Out), followed by simple examples of inference.

(In) We illustrate the rule assuming we wish to type an input that receives a single channel and a single higher-order value.

$$\frac{\Gamma \vdash u{:}(\Pi[x{:}\sigma]\tau)^{\mathtt{I}} \quad \Gamma, x{:}\sigma, y{:}\tau \vdash P :: x{:}\sigma, \Delta}{\Gamma \vdash u(x{:}\sigma, y{:}\tau).P :: u{:}(\Pi[x{:}\sigma]\tau)^{\mathtt{I}}, \Delta}$$

The assumption $\Gamma \vdash u{:}(\Pi[x{:}\sigma]\tau)^{\mathtt{I}}$ ensures u can input a channel typed by σ and a higher-order value typed by τ. In the conclusion, the free occurrences of x in both P and τ are bound (in the assumption, we can ensure x does not appear in Δ). Finally note ($\mathsf{In}^!$) is derived from (In).

(Out) Again we illustrate the rule considering an output of one channel and one higher-order value.

$$\frac{\Gamma \vdash u{:}(\Pi[x{:}\overline{\sigma}]\tau)^{\mathtt{0}} \quad \Gamma \vdash v : \sigma \quad \Gamma \vdash V : \tau\{v/x\}}{\Gamma \vdash \overline{u}\langle v, V\rangle :: u{:}(\Pi[x{:}\overline{\sigma}]\tau)^{\mathtt{0}}, v{:}\sigma}$$

We discuss each assumption and conclusion.

- $\Gamma \vdash u{:}(\Pi[x{:}\overline{\sigma}]\tau)^{\mathtt{0}}$ ensures u outputs a pair of a name typed by σ and a higher-order value typed by τ. $\overline{\sigma}$ is dualised because v is used in the input with type σ [5, 50, 53, 20].
- if the first argument is v, then the second argument should have type $\tau\{v/x\}$ since x binds free occurrences of x in τ.
- the effect of channel u and v should be recorded as a type of $\overline{u}\langle v, V\rangle$ because they will be used by the opponent input after interaction (note that we do *not* have to record the effect of V).

[1] In [52], we used the operator $\tau \setminus a$ which deletes all free occurrences of $a{:}\sigma$ form τ in order to gain the typability. Using more informative dependent/existential types, we can dispense with this operator; instead we can simply erase a which does not appear in Δ, keeping information of a in Δ as an quantified name.

One may observe the similarity between (Out) rule above and the $\exists$-introduction rule in polymorphic λ-calculi, since $\langle v, V\rangle$ can be seen as an existential package of type v and term V. This point is further discussed in the next subsection. Below we show an example of inference which uses channel dependent type non-trivially.

EXAMPLE 2.2. Let $P \stackrel{\text{def}}{=} c(x,y).(run\, y \,|\, \overline{x}\langle 1\rangle)$. We show how we can significantly constrain the behaviour of P using different type assignment. First let us assume c has the following input type:

$$\sigma_c \stackrel{\text{def}}{=} (\Pi[x{:}(\mathtt{int})^{\mathtt{0}}]\ulcorner x{:}(\mathtt{int})^{\mathtt{0}}\urcorner)^{\mathtt{I}}$$

and $\Gamma = \{c{:}\sigma_c, x{:}(\mathtt{int})^{\mathtt{0}}, y{:}\ulcorner x{:}(\mathtt{int})^{\mathtt{0}}\urcorner\}$. Then we can infer:

$$\dfrac{\dfrac{\Gamma \vdash run\, y : x{:}(\mathtt{int})^{\mathtt{0}} \quad \Gamma \vdash \overline{x}\langle 1\rangle : x{:}(\mathtt{int})^{\mathtt{0}}}{\Gamma \vdash run\, y \,|\, \overline{x}\langle 1\rangle : x{:}(\mathtt{int})^{\mathtt{0}}}\ (\mathsf{Par})}{c{:}\sigma_c \vdash c(x,y{:}\ulcorner x{:}(\mathtt{int})^{\mathtt{0}}\urcorner).(run\, y \,|\, \overline{x}\langle 1\rangle) :: c{:}\sigma_c}\ (\mathsf{In})$$

Hence this input process can only get a thunked value which has the same interface as the first argument. On the other hand, suppose c has the following channel type where x does not appear in τ.

$$\sigma'_c \stackrel{\text{def}}{=} (\Pi[x{:}(\mathtt{int})^{\mathtt{0}}]\ulcorner b{:}(\mathtt{int})^{\mathtt{0}}\urcorner)^{\mathtt{I}}$$

This is written as $((\mathtt{int})^{\mathtt{0}}\ulcorner b{:}(\mathtt{int})^{\mathtt{0}}\urcorner)^{\mathtt{I}}$ by using the abbreviation in Figure 2. Then we have:

$$c{:}\sigma'_c, b{:}(\mathtt{int})^{\mathtt{0}} \vdash c(x,y{:}\ulcorner b{:}(\mathtt{int})^{\mathtt{0}}\urcorner).(run\, y \,|\, \overline{x}\langle 1\rangle) : \{c{:}\sigma'_c, b{:}(\mathtt{int})^{\mathtt{0}}\}$$

Here c can only input a thunked process which has less than b-*accessibility*, which means it has either type $\ulcorner\urcorner$ or $\ulcorner b{:}(\mathtt{int})^{\mathtt{0}}\urcorner$.

The difference between σ_c and σ'_c above will become clearer by composing the opponent message, given in the next paragraph.

EXAMPLE 2.3. Now let us revisit the two examples discussed before. Then we obtain the following typed terms.

$\Delta \vdash \overline{c}\langle v, \ulcorner\overline{v}\langle 2\rangle\urcorner\rangle : \{c{:}\overline{\sigma_c}, v : (\mathtt{int})^{\mathtt{0}}\}$
$\Delta' \vdash \overline{c}\langle v, \ulcorner\overline{b}\langle 2\rangle\urcorner\rangle : \{c{:}\overline{\sigma'_c}, v : (\mathtt{int})^{\mathtt{0}}, b : (\mathtt{int})^{\mathtt{0}}\}$

with $\Delta = \{v{:}(\mathtt{int})^{\mathtt{0}}, c{:}\langle\sigma_c, \overline{\sigma_c}\rangle\}$ and $\Delta' = \{v{:}(\mathtt{int})^{\mathtt{0}}, b{:}(\mathtt{int})^{\mathtt{0}}, c{:}\langle\sigma'_c, \overline{\sigma'_c}\rangle\}$. Then we have: $\Delta \vdash P \,|\, \overline{c}\langle v, \ulcorner\overline{v}\langle 2\rangle\urcorner\rangle : \Delta$, which reduces to $\overline{v}\langle 2\rangle \,|\, \overline{v}\langle 1\rangle$. $\Delta' \vdash P \,|\, \overline{c}\langle v, \ulcorner\overline{b}\langle 2\rangle\urcorner\rangle : \Delta'$, which reduces to $\overline{b}\langle 2\rangle \,|\, \overline{v}\langle 1\rangle$.

Now we analyse the scope opening rule using these examples. Let us compose the above messages with environment processes $!v(z : \mathtt{int}).R_z$ and $!b(z : \mathtt{int}).R_z$ for some $z{:}\mathtt{int} \vdash R_z : \emptyset$, respectively. Then $(\nu v)(\overline{c}\langle v, \ulcorner\overline{v}\langle 2\rangle\urcorner\rangle \,|\, !v(z : \mathtt{int}).R_z)$ is typable while $(\nu b)(\overline{c}\langle v, \ulcorner\overline{b}\langle 2\rangle\urcorner\rangle \,|\, !b(z : \mathtt{int}).R_z)$ is not. We can easily check the former can be inferred by (Res) after (Par) as follows.

$$\frac{\Gamma \vdash \overline{c}\langle v, \ulcorner\overline{v}\langle 2\rangle\urcorner\rangle \,|\, !v(z).R_z : \{c{:}\overline{\sigma_c}, v{:}\sigma_v\}}{\Gamma/v \vdash (\nu v)(\overline{c}\langle v, \ulcorner\overline{v}\langle 2\rangle\urcorner\rangle \,|\, !v(z).R_z) : c{:}\overline{\sigma_c}}\ (\mathsf{Res})$$

with $\sigma_v = \langle(\mathtt{int})^{\mathtt{I}}, (\mathtt{int})^{\mathtt{0}}\rangle$ and $\Gamma = \{c{:}\overline{\sigma_c}, v : \sigma_v\}$. Note σ_c is an dependent type with $v \notin \mathsf{fn}(\sigma_c)$, hence this satisfies the side condition of (Res). On the other hand, we cannot apply (Res) to obtain the latter since name b appears free in σ'_c. If we had allowed to type the latter, the ill behaviour occurs in the following processes

$$P \,|\, (\nu b)(\overline{c}\langle v, \ulcorner\overline{b}\langle 2\rangle\urcorner\rangle \,|\, !b(z).R_z)$$

The annotated type σ'_c in P no longer guarantees c will receive a thunk of process less than b-accessibility. Actually the above untypable term is not closed under the scope opening rule. On the other hand, the reader can check $P \,|\, (\nu v)(\overline{c}\langle v, \ulcorner\overline{v}\langle 2\rangle\urcorner\rangle \,|\, !v(z).R_z)$ behaves correctly and closed under the scope opening rule.

(Common)

$$\text{(Val)}\ \frac{\Gamma, x{:}\tau, \Gamma' \vdash \mathtt{Env}}{\Gamma, x{:}\tau, \Gamma' \vdash x : \tau} \qquad \text{(Chan)}\ \frac{\Gamma, u{:}\sigma, \Gamma' \vdash \mathtt{Env}}{\Gamma, u{:}\sigma, \Gamma' \vdash u : \sigma} \qquad \text{(Pair)}\ \frac{\Gamma, u{:}\langle \sigma, \sigma' \rangle, \Gamma' \vdash \mathtt{Env}}{\Gamma, u{:}\langle \sigma, \sigma' \rangle, \Gamma' \vdash u : \sigma} \qquad \text{(Base)}\ \frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash () : \mathtt{unit}} \quad \textit{etc.}$$

(Function)

$$\text{(Abs}_H\text{)}\ \frac{\Gamma, x{:}\tau \vdash P : \tau' \quad \Gamma \vdash \tau \to \tau' : \mathtt{tp}_H}{\Gamma \vdash \lambda(x{:}\tau)P : \tau \to \tau'} \qquad \text{(App}_H\text{)}\ \frac{\Gamma \vdash P : \tau \to \tau' \quad \Gamma \vdash Q : \tau}{\Gamma \vdash PQ : \tau'}$$

$$\text{(Abs}_N\text{)}\ \frac{\Gamma, x{:}\sigma \vdash P : \tau \quad \Gamma \vdash \Pi(x{:}\sigma)\tau : \mathtt{tp}_H}{\Gamma \vdash \lambda(x{:}\sigma)P : \Pi(x{:}\sigma)\tau} \qquad \text{(App}_N\text{)}\ \frac{\Gamma \vdash P : \Pi(x{:}\sigma)\tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash Pu : \tau\{u/x\}}$$

(Process)

$$\text{(Nil)}\ \frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash \mathbf{0} : \emptyset} \qquad \text{(Par)}\ \frac{\Gamma \vdash P_{1,2} : \Delta_{1,2}}{\Gamma \vdash P_1 \,|\, P_2 : \Delta_1 \odot \Delta_2} \qquad \text{(Res)}\ \frac{\Gamma, a{:}\sigma \vdash P : \Delta, a{:}\sigma}{\Gamma \vdash (\nu a{:}\sigma)P : \Delta} \qquad \text{(Weak)}\ \frac{\Gamma \vdash u{:}\sigma \quad \Gamma \vdash P : \Delta}{\Gamma \vdash P : \Delta \odot u{:}\sigma}$$

$$\text{(In)}\ \frac{\Gamma \vdash u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{\mathtt{I}} \quad \Gamma, \vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau} \vdash P : \ \vec{x}{:}\vec{\sigma}, \Delta}{\Gamma \vdash u(\vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau}).P : \ u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{\mathtt{I}}, \Delta} \qquad \text{(In}^!\text{)}\ \frac{\Gamma \vdash u(\vec{x}{:}\vec{\tau}).P : \Delta}{\Gamma \vdash\, !u(\vec{x}{:}\vec{\tau}).P : \Delta}$$

$$\text{(Out)}\ \frac{\Gamma \vdash u : (\Pi[\vec{x}{:}\overline{\vec{\sigma}}]\vec{\tau})^{\mathtt{O}} \quad \Gamma \vdash V_j : \tau_j\{\vec{v}/\vec{x}\} \quad \Gamma \vdash v_i : \sigma_i\{\vec{v}/\vec{x}\}}{\Gamma \vdash \overline{u}\langle \vec{v}, \vec{V} \rangle \ : \ u : (\Pi[\vec{x}{:}\overline{\vec{\sigma}}]\vec{\tau})^{\mathtt{O}}, \ \vec{v}{:}\vec{\sigma}\{\vec{v}/\vec{x}\}}$$

Figure 3: Higher-Order IO Typing System

We list basic syntactic properties of typed processes. First, the proof of the following follows the routine in [52].

THEOREM 2.4. *Suppose* $\Gamma \vdash P : \tau$ *and* $P \longrightarrow\!\!\!\!\!\rightarrow P'$. *Then* $\Gamma \vdash P' : \tau$.

The following properties do not hold in the original system [52] (for example, we can take $c(x,y).(\mathit{run}\ y \,|\, \overline{x}\langle 1\rangle)$ with type σ_c in Example 2.3).

PROPOSITION 2.5. (minimum/maximum interface types) *Given* Γ *and P, suppose P is typable under* Γ. *Then:*

1. (the minimum) *there exists* Δ *such that* $\Gamma \vdash P : \Delta$ *and for all such* Δ, *there exists the unique* Δ_{min} *such that* $\Delta_{min} \subseteq \Delta$ *and* $\Gamma \vdash P : \Delta_{min}$.[2]
2. (the maximum) $\Gamma \vdash P : \Delta_{max}$ *with* $\Delta_{max} = \{u{:}\sigma \mid u{:}\sigma \in \Gamma\}$.

COROLLARY 2.6. 1. (type checking) *Given* Γ, τ *and (annotated) term P, it is decidable whether one has* $\Gamma \vdash P : \tau$.
2. (typability) *Given* Γ *and (annotated) term P, it is decidable whether there exists a type* τ *such that* $\Gamma \vdash P : \tau$

Next we examine the relationship between process types and process behaviour. Define:

$$P \Downarrow a^{\mathtt{O}} \ \Leftrightarrow\ \exists R, \vec{c}, \vec{V}.\ \text{s.t.}\ P \longrightarrow\!\!\!\!\!\rightarrow (\nu \vec{c})(\overline{a}\langle \vec{V} \rangle \,|\, R) \ \ (a \notin \{\vec{c}\})$$

Dually we define $P \Downarrow a^{\mathtt{I}}$. Below $\mathsf{md}(\sigma)$ denotes the mode of σ. The proof follows [52, Proposition 5.2].

PROPOSITION 2.7. *Suppose* $\Gamma \vdash P : \Delta$.

1. *If* $a \notin \mathsf{dom}(\Delta)$, *then* $\neg P \Downarrow a^{\mathtt{I}}$ *and* $\neg P \Downarrow a^{\mathtt{O}}$.
2. *If* $\mathsf{md}(\Delta(a)) = \mathtt{I}$, *then* $\neg P \Downarrow a^{\mathtt{O}}$. *Dually for* $\mathtt{O}$.

2.4 Channel Existential Types

Channel dependent types allow names in a dependent binding to occur both in the (receiving) process and in the carried type. Often this generality is unnecessary: by restricting the bound names to occur only in carried types, we arrive at *channel existential types*, which ensure stronger syntactic properties including type erasure.

As an example, recall the input process in the previous section where c has type $(\Pi[x{:}(\mathtt{int})^{\mathtt{O}}]\ulcorner x{:}(\mathtt{int})^{\mathtt{O}\urcorner})^{\mathtt{I}}$:

$$c(x,y{:}\ulcorner x{:}(\mathtt{int})^{\mathtt{O}\urcorner}).(\mathit{run}\ y \,|\, \overline{x}\langle 1\rangle)$$

Here x appears in $\overline{x}\langle 1\rangle$ and types of y, hence the opponent output process always needs to send the value V together with binding name v as in $\overline{c}\langle v, V\rangle$. We can, however, erase names x and v for binding if they are used only as *linking* into interface types; that is, the scope of existential quantified name is limited to the communicated value V. For example, in the following term, the output can safely send V without attaching v.

$$c(y).(y() \,|\, \overline{a}\langle y\rangle) \,|\, (\nu v)(\overline{c}\langle V\rangle \,|\, !v(z).R) \tag{10}$$

Following existential types in polymorphic λ, we write $\exists[x{:}\sigma]\tau$ for the channel existential type. In the above case, the term in (10) has a type $\{c{:}\langle \sigma_a, \overline{\sigma_a}\rangle, a{:}\sigma_a\}$ where $\sigma_a = (\exists[x{:}\overline{\sigma_x}]\ulcorner x{:}\sigma_x\urcorner)^{\mathtt{O}}$, $V = \ulcorner \overline{v}\langle 2\rangle\urcorner$ and $\sigma_x = (\mathtt{int})^{\mathtt{O}}$ (assuming $!v(z).R$ has type $v{:}(\mathtt{int})^{\mathtt{I}}$).

Many examples of higher-order code mobility can be typed using only channel existential types.[3] Not only does it satisfy the standard erasure theorem (for existential quantifiered names), but also it ensures a stronger contract between a sender and a receiver, ensuring the opaqueness and inaccessibility of the abstracted names at the receiver's side. In the following technical development, we shall mainly use this restricted form of channel dependent types.

2.5 Example: Remote Procedure Migration

We conclude this section with a larger example which use several elements of the typed $\lambda\pi$. Consider a data server at location l, which first checks accessibility of incoming code, and if it is safe, it provides a value n to it repeatedly to a local channel c inside its location. This server can be written as:

$$\mathtt{Server}(l) \ \stackrel{\text{def}}{=}\ (\nu c)(!l(y{:}\tau).(yc) \,|\, !c(x).\overline{x}\langle n\rangle) \tag{11}$$

Then the following parameterised code inputs the server's data twice and returns the final result to e.

$$\mathtt{return\ twice\ to}\ e \ \stackrel{\text{def}}{=}\ \lambda(c)\overline{c}(f)f(z_1).\overline{c}(f)f(z_2).\overline{e}\langle z_1 + z_2\rangle$$

[2] $\Delta_1 \subseteq \Delta_2$ means for all $u \in \mathsf{dom}(\Delta_1)$, $\Delta_1(u) = \Delta_2(u)$ or $\Delta_2(u) = \langle \sigma, \overline{\sigma}\rangle$ with $\sigma = \Delta_1(u)$.

[3] The generality of channel dependent types becomes important in some significant theoretical and pragmatic settings. One such instance is discussed in [54].

This function has the following functional name dependent type:

$$\tau' = \Pi(x{:}\sigma_x)\{x{:}\sigma_x, e{:}(\mathtt{nat})^{\mathtt{0}}\}$$

where $\sigma_x = ((\mathtt{nat})^{\mathtt{I}})^{\mathtt{0}}$. A client can send this higher-order request to the server, and wait for the result at the hidden name e:

$$\mathtt{let}\ x = \overline{l}\langle \mathtt{twice}\rangle\ \mathtt{in}\ R \stackrel{\text{def}}{=} (\nu e)(\overline{l}\langle \mathtt{return\ twice\ to}\ e\rangle \,|\, e(x).R) \quad (12)$$

This process can be composed with $\mathtt{Server}(l)$ if τ annotating y in $\mathtt{Server}(l)$ is specified as:

$$\tau \;=\; \exists[e{:}(\mathtt{nat})^{\mathtt{0}}]\tau'$$

The example shows that the combination of types in the $\lambda\pi$ offers: (1) protection for the server against the client (by τ, which may be checked at the time of arrival of the code), (2) the invisibility (hence inaccessibility) of the local service point c in the server to the client; (3) flexible programming by the client (i.e. it can easily replace `twice` by `thrice`); (4) ensuring that the returned message to e has not been touched (by existential type $\exists[e{:}(\mathtt{nat})^{\mathtt{0}}]\tau'$, the server will not touch $\overline{e}\langle z_1 + z_2\rangle$ if it is well-typed); and (5) safe return to the destination specified by the client (again by existential type). These features are the key to the applications later.

3. INTEGRATION WITH LINEAR/AFFINE TYPES

3.1 Linearity and Affinity

In this section we briefly summarise the integration of linear/affine types, originally introduced for the π-calculus in [5, 6, 20, 49, 50, 53], into the $\lambda\pi$. We also augment the calculus with branching and reference. Together we can precisely model a wide range of language constructs, preparing us for the applications in Sections 4 and 5.

As noted in Introduction, affinity means possibly diverging behaviour in which a question is given an answer *at most once*; while linearity means a question is always given an answer *precisely once*. Further these types can be mixed, called *linear/affine types*. Integrating $\lambda\pi$ with these three disciplines is straightforward. The syntax of terms and types remain the same. All we need to do is to enrich the mode p in channel types. For affine types we consider:

$\downarrow_{\mathrm{A}}$	Affine input	$\uparrow_{\mathrm{A}}$	Affine output
$!_{\mathrm{A}}$	Unique Affine Server	$?_{\mathrm{A}}$	Client requests to $!_{\mathrm{A}}$

We also use $\updownarrow$ to indicate uncomposability at affine channels. $\downarrow_{\mathrm{A}}, !_{\mathrm{A}}$ are input modes, while $\uparrow_{\mathrm{A}}, ?_{\mathrm{A}}$ are output modes, and the left is dual of the right. The composition of types $\sigma \odot \sigma'$ now incorporates the following algebra of these enriched modes: $\downarrow_{\mathrm{A}} \odot \uparrow_{\mathrm{A}} = \updownarrow$, $!_{\mathrm{A}} \odot ?_{\mathrm{A}} = !_{\mathrm{A}}$ and $?_{\mathrm{A}} \odot ?_{\mathrm{A}} = ?_{\mathrm{A}}$. The first rule says an affine input can only be composed with an affine output, with no further connection, while the last two say a server can be composed with zero or more clients. If all common ports of two processes are pairwised composable, then their parallel composition is defined. For example, if $\overline{a}$ has a $\uparrow_{\mathrm{A}}$-mode, $\overline{a}\,|\,a.\mathbf{0}$ is typable, while $\overline{a}\,|\,\overline{a}$ is not.

For linearity, the only addition is causality edges between input and output channels as $a{:}()^{\downarrow_{\mathrm{L}}} \to b{:}()^{\uparrow_{\mathrm{L}}}$. Edges prevent circular causality when composing two processes. For example, $a.\overline{b}\,|\,b.\overline{a}$ is typable in the affine system, but not in the linear one. We use $\downarrow_{\mathrm{L}}, \uparrow_{\mathrm{L}}, !_{\mathrm{L}}, ?_{\mathrm{L}}$ to denote the linear mode. We can then combine the linear and affine systems by incorporating simple conditions given in [20, 5, 50, 53, 49]. The typing rules are defined just by replacing **(Process)** in Figure 3 with the corresponding rules of the affine, linear, and linear/affine π-calculus. We call these calculi $\lambda\pi^{\mathsf{A}}$, $\lambda\pi^{\mathsf{L}}$ and $\lambda\pi^{\mathsf{LA}}$, respectively. We leave the typing rules to [54].

The following lists the basic properties satisfied in $\lambda\pi^{\mathsf{A}}$, $\lambda\pi^{\mathsf{L}}$ and $\lambda\pi^{\mathsf{LA}}$. Below $\Downarrow_{a^p}$ is defined as $\Downarrow_{a^{\mathtt{0}}}$ in Proposition 2.7, and Δ is closed means $\{!_{\mathrm{L}}, !_{\mathrm{A}}, \updownarrow\} \supseteq \mathsf{md}(\Delta)$. Note that (2) closes the open problem posed in [40].

THEOREM 3.1. (1) (subject reduction) *Suppose* $\Gamma \vdash P : \tau$ *and* $P \longrightarrow\!\!\!\!\to P'$. *Then* $\Gamma \vdash P' : \tau$.
(2) (strong normalisation) $\longrightarrow$ *in* $\lambda\pi^{\mathsf{L}}$ *is strong normalising.*
(3) (linear liveness) *Suppose* $\Gamma \vdash P : \Delta, e{:}\sigma$ *and* $\mathsf{fv}(P) = \emptyset$ *in* $\lambda\pi^{\mathsf{LA}}$ *with* $\mathsf{md}(\sigma) = \uparrow_{\mathrm{L}}$ *and* Δ *closed. Then for all* P' *such that* $P \longrightarrow\!\!\!\!\to P'$, *we have* $P' \Downarrow_{e^{\uparrow_{\mathrm{L}}}}$.

(2) is proved by first translating $\lambda\pi^{\mathsf{L}}$ into π^{L} using a type-directed encoding (on which we shall discuss in § 4.2), then applying the result in [50]. This is used for proving (3), reducing transitions of π^{LA} into π^{L} by the result in [49]. Intuitively, (3) says that, when composed with all complementary processes, a linear channel *eventually* fires even in the presence of non-termination. The property is essential in the secrecy analysis in § 4. Another interesting liveness property comes from existential channel types.

THEOREM 3.2. (hiding by existential names) *Suppose* $\Gamma \vdash P : \Delta$ *and* $\mathsf{fv}(P) = \emptyset$ *with* Δ *closed and* $\Gamma \vdash u{:}(\exists[c{:}(\mathtt{nat})^{\uparrow_{\mathrm{L}}}]\ulcorner c{:}(\mathtt{nat})^{\uparrow_{\mathrm{L}}}\urcorner)^{!_{\mathrm{L}}}$. *Then* $P \stackrel{u\langle\ulcorner \overline{c}\langle n\rangle\urcorner\rangle}{\longrightarrow} P'$ *implies there exists* Q *such that* $P' \longrightarrow\!\!\!\!\to Q \stackrel{\overline{c}\langle n\rangle}{\longrightarrow}$.

This theorem means that an acknowledgement bound by existential name is eventually returned to the sender without being touched by the receiver. Recall Example 2.5. If τ in $\mathtt{Server}(l)$ is typed as

$$\tau \;=\; \exists[e{:}(\mathtt{nat})^{\uparrow_{\mathrm{L}}}]\Pi(x{:}\sigma_x)\{x{:}\sigma_x, e{:}(\mathtt{nat})^{\uparrow_{\mathrm{L}}}\}$$

with $\sigma_x = ((\mathtt{nat})^{\downarrow_{\mathrm{L}}})^{?_{\mathrm{L}}}$, then even if R' contains non-terminating behaviour, the remote call to the server can always succeed:

$$\mathtt{Server}(l) \,|\, \mathtt{let}\ x = \overline{l}\langle \mathtt{twice}\rangle\ \mathtt{in}\ R \,|\, R' \longrightarrow\!\!\!\!\to (\mathtt{Server}(l) \,|\, R\{2n/x\} \,|\, R'')$$

Now recall the example of (6) in §. 1. This term is not typable by assigning $()^{\uparrow_{\mathrm{L}}}$ to `ack`. To accept outputs, *run* y in $\mathsf{Sever}(a)$ should have type $\{\mathtt{ack}{:}()^{\uparrow_{\mathrm{L}}}, \Delta\}$ which contains the *free* linear output; hence it cannot be prefixed by the replication $!a(x)$ because if we do so, it means a linear channel repeatedly appear, cf. [28, 5, 53]. On the other hand, the server in (7) is typable since the channel $\mathtt{ack}{:}()^{\uparrow_{\mathrm{L}}}$ in the type $\{\mathtt{ack}{:}()^{\uparrow_{\mathrm{L}}}, \Delta\}$ of *run* y is going to be *bound* by $\exists$. Hence channel a can have type $(\exists[x{:}()^{\uparrow_{\mathrm{L}}}][x{:}()^{\uparrow_{\mathrm{L}}}, \Delta])^{!_{\mathrm{L}}}$. Note such argument is unconceivable without the fine-grained process type.

3.2 Branching and References

We augment the $\lambda\pi^{\mathsf{LA}}$ with (1) a constant reference agent $\mathsf{ref}\langle xv\rangle$ (for representing stateful computation); and (2) branching and selection (for representing conditionals, methods for objects and atomic data), as done for the π-calculus in [5, 50, 53, 20]. The integration is straightforward, extending the grammar as follows.

$$P \quad ::= \quad \cdots \mid u[\&_{i\in I}(\vec{y}_i).P_i] \mid \overline{u}\mathtt{in}_i\langle \vec{V}\rangle \mid \mathsf{ref}\langle xv\rangle$$

We often omit the indexing set I of $u[\&_{i\in I}(\vec{y}_i).P_i]$. $u[\&_{i\in I}(\vec{y}_i).P_i]$ is *branching*, while $\overline{x}\mathtt{in}_i\langle \vec{V}\rangle$ is *selection*. The reduction is defined as:

$$\begin{array}{rcl}
u[\&_i(\vec{y}_i).P_i] \,|\, \overline{u}\mathtt{in}_j\langle \vec{V}_j\rangle & \longrightarrow & P_j\{\vec{V}_j/\vec{y}_j\} \\
!\,u[\&_i(\vec{y}_i).P_i] \,|\, \overline{u}\mathtt{in}_j\langle \vec{V}_j\rangle & \longrightarrow & !\,u[\&_i(\vec{y}_i).P_i] | P_j\{\vec{V}_j/\vec{y}_j\} \\
\mathsf{ref}\langle xv\rangle \,|\, \overline{x}\mathtt{in}_1\langle c\rangle & \longrightarrow & \mathsf{ref}\langle xv\rangle \,|\, \overline{c}\langle v\rangle \\
\mathsf{ref}\langle xv\rangle \,|\, \overline{x}\mathtt{in}_2\langle wc\rangle & \longrightarrow & \mathsf{ref}\langle xw\rangle \,|\, \overline{c}
\end{array}$$

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

for the reference represents *reading*, the second one *writing*. As an example of branching, 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.

For types, the following action modes are additionally used.

$!_{\text{LM}}$	Linear stateful server	$?_{\text{LM}}$	Linear stateful client
$!_{\text{M}}$	Affine stateful server	$?_{\text{M}}$	Affine stateful output

These modes are stateful versions of $!_{\text{L}}$, $?_{\text{L}}$, $!_{\text{A}}$, and $?_{\text{A}}$, respectively. $!_{\text{LM}}$ and $?_{\text{LM}}$ are mutually dual, while $!_{\text{M}}$ and $?_{\text{M}}$ are mutually dual. Using the added action modes, the channel types are extended by the following syntax.

$$\begin{array}{rcl}\sigma & ::= & \cdots \mid [\&_{i\in I}\,\tau_i]^{p_{\text{I}}} \mid [\oplus_{i\in I}\,\tau_i]^{p_{\text{O}}} \mid (\tau)^{!_{\text{M}}} \mid (\tau)^{?_{\text{M}}} \\ & \mid & \mathsf{ref}\,\langle\sigma\rangle \mid \mathsf{rw}\,\langle\sigma\rangle \mid \mathsf{r}\langle\sigma\rangle \mid \mathsf{w}\langle\sigma\rangle \mid \overline{\mathsf{r}\langle\sigma\rangle} \mid \overline{\mathsf{w}\langle\sigma\rangle}\end{array}$$

with $\tau = \Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau}$ and $\tau_i = \Pi[\vec{x}_i{:}\vec{\sigma}_i]\vec{\tau}_i$ and $\mathsf{ref}\,\langle\sigma\rangle = [(\sigma)^{\uparrow_{\text{L}}}\&\overline{\sigma}()^{\uparrow_{\text{L}}}]^{!_{\text{LM}}}$, $\mathsf{r}\langle\sigma\rangle = [\oplus_{i=1}(\overline{\sigma})^{\downarrow_{\text{L}}}]^{?_{\text{LM}}}$, $\mathsf{w}\langle\sigma\rangle = [\oplus_{i=2}\sigma()^{\downarrow_{\text{L}}}]^{?_{\text{LM}}}$ and $\mathsf{rw}\,\langle\sigma\rangle = \overline{\mathsf{ref}\,\langle\sigma\rangle}$. $p_{\text{I}}/p_{\text{O}}$ denotes an input/output mode. $[\&\tau_i]^{p_{\text{I}}}/[\oplus\tau_i]^{p_{\text{O}}}$ is called a branching/selection type. We also define the subtyping on selection as:

$$\Gamma \vdash [\tau_1 \oplus \tau_2 \ldots \oplus \tau_n]^{p_{\text{O}}} \leq [\tau_1' \oplus \tau_2' \ldots \oplus \tau_n' \ldots \oplus \tau_{n+k}]^{p_{\text{O}}}$$

with $\Gamma \vdash \tau_i \leq \tau_i'$. The subtyping of branching type is defined dually. Intuitively if $\tau \leq \tau'$, then τ has more restricted behaviour than τ'. The same subtyping relation is applied for references, hence we have, for example, $\mathsf{r}\langle\sigma\rangle \leq \mathsf{rw}\,\langle\sigma\rangle$, $\mathsf{w}\langle\sigma\rangle \leq \mathsf{rw}\,\langle\sigma\rangle$ etc. (note σ is invariant since it occurs as both input and output). The (sub)typing rules are left to [54]. The basic properties such as subject reduction and the minimum interface theorems, as well as the same liveness properties on linear channels, are proved in this system.

We call the resulting typed calculus $\lambda\pi^{\mathsf{LAM}}_{<}$.

4. TWO SECURITY ANALYSES

4.1 Secrecy and Higher-Order Mobility

In systems with mobile code, secrecy analysis means ensuring that coordination of pieces of code — which can be distributed over different nodes, communicated and instantiated in the type-safe manner — does not reveal sensitive information to low-level outlets. Following [17, 20], we model flow of information as transformation of behaviour by interaction. Levels of information become levels attached to channel types. When carrying out this programme in $\lambda\pi^{\mathsf{LAM}}_{<}$, two subtleties arise:

1. *Interplay between linear/affine modes and secrecy analyses.*
2. *The added typability relative to trust on other code/channels.*

The point (1) is already observed in [17, 20] as well as many earlier work [2, 44, 42]. Here we merely point out that, for example, the secrecy analysis of such a simple command as $x^{\mathbb{H}} := y^{\mathbb{H}}; z^{\mathbb{L}} := w^{\mathbb{L}}$ demands distinction between linearity and affinity (i.e. if we only use affine channels, this obviously safe command becomes unsafe).

To illustrate the point (2), we use a simple example, which is a variation of the secrecy example in Introduction. Consider the following client process which wishes to download an application for calculating a sensitive datum.

$$\mathtt{client} \stackrel{\text{def}}{=} \overline{a}(c)c(X).(\nu\, c')(((Xn^{\mathbb{H}})c')\,|\,c'(y).P)$$

The process asks for the code via a, receives it as X, instantiates it with a local secret datum n and a local channel c', and receives the result via c'. As a corresponding server we consider:

$$\mathtt{server} \stackrel{\text{def}}{=} !a(c).\overline{c}\langle P\rangle \quad \text{with } P \stackrel{\text{def}}{=} \lambda z.\lambda c'.(\overline{c'}\langle z+z\rangle\,|\,\overline{b}^{\mathbb{L}}\langle z\rangle)$$

The behaviour of the carried code P is best illustrated using its higher-order process type:

$$\mathtt{nat}_{\mathbb{H}} \Rightarrow \Pi(c' : (\mathtt{nat}_{\mathbb{H}})^{\uparrow}_{\mathbb{H}})\{c' : (\mathtt{nat}_{\mathbb{H}})^{\uparrow}_{\mathbb{H}}, b : (\mathtt{nat}_{\mathbb{H}})^{\uparrow}_{\mathbb{L}}\}$$

Does this type represent secure behaviour? At first sight, the answer seems no, since it sends a high-level datum to a low-level channel. However, as illustrated in Introduction, our secrecy analysis can take into account not only senders but also receivers. In fact, suppose, at b, the following process is located:

$$b(y : \mathtt{nat}_{\mathbb{H}}).\overline{e}^{\mathbb{H}}\langle y\rangle$$

The low-level datum is just reported to a high-level channel, so no real low-level effects come out. If, on the other hand, a process at b uses the received datum insecurely (e.g. transforms it to a low-level datum), or there are eavesdroppers at b, secrecy is compromised. Note when a complex data structure (including mobile code) is exchanged, its part may contain a datum too sensitive for a receiver. The above discussion suggests this would be safe as far as the receiver is ensured to behave well. We can summarise these observations as follows.

A For a relatively low-level channel, if (1) each code listening to it is secrecy-safe, and (2) no other principals have access to it, one can safely send a high-level datum to the channel.

B If these assumptions are not met, then sending a high-level datum to the channel should be prohibited.

The type-based secrecy analysis outlined below can type a wide range of behaviours following the principle (**A**).

4.2 Non-Interference

We now outline the instrumentation of $\lambda\pi^{\mathsf{LAM}}_{<}$ with secrecy concerns. Let $\mathcal{L}$ be a complete lattice of secrecy levels (higher means more secret) and let $s, s', \ldots$ range over $\mathcal{L}$. Then we define:

$$\begin{array}{rcl}\sigma & ::= & (\tau)^{p}_{s} \mid [\&_{i\in I}\,\tau_i]^{p_{\text{I}}}_{s} \mid [\oplus_{i\in I}\,\tau_i]^{p_{\text{O}}}_{s} \mid \langle\sigma,\sigma'\rangle \\ & \mid & \mathsf{ref}_s\langle\sigma\rangle \mid \mathsf{rw}_s\langle\sigma\rangle \mid \mathsf{r}_s\langle\sigma\rangle \mid \mathsf{w}_s\langle\sigma\rangle \mid \overline{\mathsf{r}_s\langle\sigma\rangle} \mid \overline{\mathsf{w}_s\langle\sigma\rangle}\end{array}$$

with $\tau = \Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau}$ and $\tau_i = \Pi[\vec{x}_i{:}\vec{\sigma}_i]\vec{\tau}_i$. Note function types are not annotated. The duality is defined preserving secrecy levels. Pair types are as before. $\mathsf{sec}(\tau)$ denotes τ's outermost secrecy level.

Following [20], we take the approach that the level of effects of a process is calculated from its process type. For this purpose we define the *tamper level* $\mathsf{tamper}(\tau)$ of τ. $\mathsf{tamper}(\tau)$ is $\mathsf{sec}(\tau)$ if τ is *immediately tampering*, which means either $\tau = [\oplus_i\vec{\tau}_i]^{\uparrow_{\text{L}}}_{s}$, $\mathsf{md}(\tau) \in \{\uparrow_{\text{A}}, ?_{\text{M}}, ?_{\text{LM}}\}$ or $\mathtt{nat}_s$ — τ emits non-trivial information at the time of interaction. $\mathsf{tamper}(\tau)$ is $\top$ if τ is *innocuous*, i.e. $\tau = \mathtt{unit}$ or $\mathsf{md}(\tau) \in \{?_{\text{L}}, ?_{\text{A}}, \updownarrow\}$ (see [54] for the rest). As an example, the tamper level of code $\ulcorner\overline{b}^{\mathbb{L}}\urcorner$ in (8) in § 1 is $\mathbb{L}$ because b's tamper level is $\mathbb{L}$. While the tamper level of P in (8) is $\mathbb{H}$, because a is immediately tampering, we can ignore the level of b.

The secrecy-enhancement of the typing rules is straightforward integration of the rules of $\pi^{\mathsf{LAM}}_{\text{sec}}$ in [20]; we only have to change those rules which receive non-trivial information, saying the tamper level of the body is no lower than that of the received information [54].

The principle (**B**) is also simply formalised by precluding output types of the form $(\tau)^{p_{\text{O}}}_{s}$ such that $\mathsf{tamper}(\tau) \sqsubseteq s$ where τ is a higher-order value type. This paper focuses on the principle (**A**) from the typability view point.

Proving Non-Interference. For proving the non-interference (i.e. a high-level behaviour never affects low-level behaviour) of securely typed higher-order processes, a natural idea would be to use

a type-preserving translation into the corresponding securely typed π-calculus. But a known translation from HOπ into π by [38, 39] cannot be directly used since it is *not* type preserving. In [38], a message which carries a value $\lambda(x).P$ is translated into

$$[\![\overline{a}\langle\lambda(x).P\rangle]\!] \stackrel{\text{def}}{=} \overline{a}(c)!c(x).[\![P]\!]$$

where a process actually used in a receiver stays in the sender side. Hence if P has a type Δ, then the mapping of the sender *gains* the accessibility Δ, unlike the source process. Symmetrically for a receiver: for example, assuming Δ is a type of P,

$$\Gamma \vdash \overline{a}\langle\ulcorner P \urcorner\rangle : a:\sigma_a \quad \text{and} \quad \Gamma \vdash !a(x:\ulcorner\Delta\urcorner).run\,x : \{a:\overline{\sigma}_a, \Delta\},$$

with $\Gamma = \Delta, a:\langle\sigma_a, \overline{\sigma}_a\rangle$ and $\sigma_a = (\ulcorner\Delta\urcorner)^{\mathsf{O}}$. However using a mapping in [38], they are translated into

$$\Gamma^\circ \vdash [\![\overline{a}\langle\ulcorner P \urcorner\rangle]\!] : a:\sigma_a^\circ, \Delta^\circ \quad \text{and} \quad \Gamma^\circ \vdash !a(x).\overline{x} : a:\overline{\sigma}_a$$

where $\sigma_a^\circ = ((\,)^{\mathtt{I}})^{\mathsf{O}}$ and Δ° is a translation of P's process type. Basic notions as linearity and affinity are not preserved either.

Fortunately, the fine-grained process types in $\lambda\pi^{\mathsf{LAM}}_<$ lead to a radically new encoding of higher-order mobile processes into the first order ones, which is indeed *type-preserving*. In the new translation, names in process type in P above initially occur bound, and are eventually instantiated when x finally migrates from the receiver. Note x *knows* free names of P because of its assigned type $\ulcorner\Delta\urcorner$. Thus assuming $\{\vec{b}\} = \mathsf{dom}(\Delta)$, the new translation becomes:

$$[\![\overline{a}\langle\ulcorner P \urcorner\rangle]\!] \stackrel{\text{def}}{=} \overline{a}(c)!c(\vec{b}).[\![P]\!] \text{ and } [\![!a(x).run\,x]\!] \stackrel{\text{def}}{=} !a(x).\overline{x}\langle\vec{b}\rangle$$

where Δ° is recorded in the receiver side this time. Note this encoding is unconceivable without the fine-grained process type. Combined with Milner's encoding of the call-by-value λ-calculus [20, Figure 7], we can now translate each $\lambda\pi^{\mathsf{LAM}}_<$-process into a securely typed π-process in [20]. The translation preserves not only typability but action modes, termination and tampering levels. Thus the security-sensitive contextual congruence for $\lambda\pi^{\mathsf{LAM}}_<$ (see below) is soundly embedded in the corresponding congruence for the secure π-calculus in [20].

Define $\cong_s$ on securely-typed $\lambda\pi^{\mathsf{LAM}}_<$-processes as the maximum typed congruence which satisfies: (i) it is reduction-closed, i.e. $P \cong_s Q$ and $P \longrightarrow P'$ implies $Q \twoheadrightarrow Q'$ and $P' \cong_s Q'$; and (ii) if $\vdash P_1 \cong_s P_2 : x:()^{\uparrow_{\mathrm{A}}}_s$ and $P_1 \Downarrow_{x^{\uparrow_{\mathrm{A}}}}$ then $P_2 \Downarrow_{x^{\uparrow_{\mathrm{A}}}}$ [18]. The $[\![\,]\!]$ and $(\,)^\circ$ are embeddings of terms/types from $\lambda\pi^{\mathsf{LAM}}_<$ to the $\pi^{\mathsf{LAM}}_{\mathrm{sec}}$-calculus.

PROPOSITION 4.1. *1. (adequacy of the translation)* Suppose $\Gamma \vdash P : \Delta$ with $\mathsf{fv}(P) = \emptyset$. Then $\Gamma^\circ \vdash [\![P]\!] : \Delta^\circ$ and $\mathsf{tamper}(\Delta) = \mathsf{tamper}(\Delta^\circ)$. Moreover $P \Downarrow_{a^p}$ iff $[\![P]\!] \Downarrow_{a^p}$.
2. (soundness) $P \cong_s Q$ if $[\![P]\!] \cong_s [\![Q]\!]$.

The noninterference says that two processes with a secrecy level incompatible with s can always be equated by $\cong_s$.

THEOREM 4.2. (non-interference in $\lambda\pi^{\mathsf{LAM}}_<$) *If* $\Gamma \vdash P_{1,2} : \Delta$ *such that* $\mathsf{tamper}(\Delta) = s$ *and* $s \not\sqsubseteq s'$, *then* $\Gamma \vdash P_1 \cong_{s'} P_2 : \Delta$.

4.3 Access Control in $\lambda\pi^{\mathsf{LAM}}_<$

Suppose a portion of a memory (e.g. a large array) is shared by a resident high-level program (P) and a visiting "medium"-level program (Q). Assume the write access level of the array is set to "high", while the read access level, and practically the secrecy level, can be set to "medium". Then Q cannot write on the array, but can read from it. Note that even if P is secrecy-wise safe, it can still write on the array: as far as it does not write in the way its previous reads at neither lower nor equal level than "medium" are reflected, no insecure information transfer to Q occurs. Thus secrecy and access control are in principle distinct concerns; this subsection discusses how the latter can also be modelled cleanly in $\lambda\pi^{\mathsf{LAM}}_<$.

In the context of extensible architecture such as SPIN [12], Java [13, 25] and CLR [8], a couple of key mechanisms to smoothly administer access control have been identified (and implemented). Some of them are:

1. *Safe access to resources through well-typed references, which should be maintained through dynamic linkage (possibly combining name-space control).*

2. *Dynamic, and often automatic, update of current access rights when crossing a boundary of protection domains (i.e. pieces of code from distinct origins and static authority levels).*

3. *Primitives to alter one's own current access right, as far as it does not exceed its static authority level.*

Variations of these and other mechanisms and their combinations are surveyed in [12, 36, 3, 11]. Our aim in the subsection is to use type structures of $\lambda\pi^{\mathsf{LAM}}_<$ for proposing a powerful modelling framework of these access control mechanisms.

Modelling Authorities. We represent access rights by authority levels, written $\mathsf{a}, \mathsf{a}', ..$, which form a complete lattice. They can be, for example, the set of subsets of all principals, ordered by reverse inclusion. We write $\mathsf{a}_1 \sqsubseteq \mathsf{a}_2$ if a_1 is lower than a_2. The first step to model running code with an authority level is to augment $\lambda\pi^{\mathsf{LAM}}_<$ with a framed process following [3, 11, 12], which works quite effectively in the setting of $\lambda\pi$. A framed process, written with different fonts $\mathsf{P}, \mathsf{Q}, \mathsf{R}, \ldots$, is simply a process enclosed by an authority level boundary. For example,

$$\mathsf{P} \stackrel{\text{def}}{=} \{(\nu c)(\overline{b}\mathtt{in}_2\langle V_2 c\rangle \,|\, c(x).P')\}_{\mathsf{alice}}$$

is a thread[4] which tries to output via b at the second branch, running at the level "alice". We assume the boundary as above distributes over parallel composition and restriction as expected ($\{P \,|\, Q\}_{\mathsf{a}} \equiv \{P\}_{\mathsf{a}} \,|\, \{Q\}_{\mathsf{a}}$ and $\{(\nu a{:}\sigma)P\}_{\mathsf{a}} \equiv (\nu a{:}\sigma)\{P\}_{\mathsf{a}}$).

Given a thread, we next model *resources*, which include data structures, objects and code bases (linked classes). They in general have static authorities. One of the key observations underlying our modelling framework is that all these resources are uniformly representable as *replicated processes*. For example, consider:

$$\mathsf{R} \stackrel{\text{def}}{=} \{!b[^{\mathtt{bob}}(yz).S_1 \;\&\; ^{\mathtt{guest}}(yz).S_2]\}_{\mathtt{bob}}$$

R is an object with two methods: one can be invoked by those whose current levels are higher than bob; the other for those higher than guest. Assuming $\mathtt{guest} \sqsubseteq \mathtt{bob}$, the code has the static level bob since it may run as bob. If we have $\mathtt{guest} \sqsubseteq \mathtt{alice}$ too, then the access by P to R is safe. As other instances, access control for object creation and variables can be represented uniformly by annotating replicated processes with authority levels (see § 5).

Modeling Authority Change. Both stack-inspection [11, 36] and the history-based access control [3] offer a mechanism to automatically update a current authority level. $\lambda\pi^{\mathsf{LAM}}_<$ can elegantly

[4] In the linear/affine type discipline, a process with an active output or an internal τ-action can be considered as one thread [5].

model both of these mechanisms via simple instrumentation of operational semantics. First we focus on the stack inspection.

$$\textbf{(rep)}\quad \{!u[\&_i^{\mathsf{a}_i}(\vec{y}_i:\vec{\mathsf{t}}_i).P_i]\}_{\mathsf{a}} \mid \{\overline{u}\mathtt{in}_j\langle\vec{V}_j\rangle\}_{\mathsf{a}'} \longrightarrow \{!u[\&_i(\vec{y}_i:\vec{\mathsf{t}}_i).P_i]\}_{\mathsf{a}} \mid \{P_j\{\vec{V}_j/\vec{y}_j\}\}_{\mathsf{a}\sqcap\mathsf{a}'}$$

$$\textbf{(com)}\quad \{u[\&_i(\vec{y}_i:\vec{\mathsf{t}}_i).P_i]\}_{\mathsf{a}} \mid \{\overline{u}\mathtt{in}_j\langle\vec{V}_j\rangle\}_{\mathsf{a}'} \longrightarrow \{P_j\{\vec{V}_j/\vec{y}_j\}\}_{\mathsf{a}}$$

Let us see how it works using P and R above. Assume "bob" and "alice" have the meet "usr", and let $S_2\{V_2c/yz\} \longrightarrow^* \overline{c}\langle W\rangle$.

$$\begin{array}{rcl} \mathsf{P}|\mathsf{R} & \equiv & (\nu c)(\mathsf{R} \mid \{\overline{b}\mathtt{in}_2\langle V_2c\rangle\}_{\mathsf{alice}} \mid \{c(x).P'\}_{\mathsf{alice}}) \\ & \longrightarrow & (\nu c)(\{S_2\{V_2c/yz\}\}_{\mathsf{usr}} \mid \{c(x).P'\}_{\mathsf{alice}}) \mid \mathsf{R} \\ & \longrightarrow^* & (\nu c)(\{\overline{c}\langle W\rangle\}_{\mathsf{usr}} \mid \{c(x).P'\}_{\mathsf{alice}}) \mid \mathsf{R} \\ & \longrightarrow & \{P'\{W/x\}\}_{\mathsf{alice}} \mid \mathsf{R} \end{array}$$

Thus the rules simulate the principle of stack inspection (note the final reduction restores the original level when the value is returned, corresponding to discarding all frames when the value is obtained in the operational semantics of λ_{sec} [36]). This is because linear/affine type structure enables a precise simulation of stack discipline (cf. bracketing condition of game semantics [4, 21, 19]), even though $\lambda\pi_<^{\mathsf{LAM}}$ includes non-determinism and higher-order code passing. Further the framework is extensible: if we lower the level of the resulting process in **(com)** as in **(rep)** (i.e. a to $\mathsf{a}\sqcap\mathsf{a}'$), then the automatic update mechanism matches the history-based [3].

Both stack-based and history-based approaches also have a mechanism to freely change (usually raise) the authority level. This is incorporated by the following rule.

$$\textbf{(set)}\qquad \{\mathtt{set}(\mathsf{a})\{P\}\}_{\mathsf{a}'} \longrightarrow \{P\}_{\mathsf{a}}$$

The following example illustrates this rule in action.

$$\{\mathtt{set}(\mathsf{b})\{(\mathtt{set}(\mathsf{a})\{P\} \mid Q)\}\}_{\mathsf{c}} \longrightarrow \{\mathtt{set}(\mathsf{a})\{P\}\}_{\mathsf{b}} \mid \{Q\}_{\mathsf{b}} \longrightarrow \{P\}_{\mathsf{a}} \mid \{Q\}_{\mathsf{b}}$$

Note the authority change occurs only when the "set" construct (which freezes the body as input prefix) is precisely wrapped by an authority boundary. This is essential for having consistent operational semantics.

Static Typing for Access Control. We can instrument the type discipline of $\lambda\pi_<^{\mathsf{LAM}}$ to statically ensure the lack of access error. While access levels to resources may be dynamically decided in practice, static checking offers potential for optimisation and suggests flexible combination with dynamic checking [12]. We start with defining a notion of access errors $\{P\}_{\mathsf{a}} \stackrel{Aerr}{\longrightarrow}$ (read: "$\{P\}_{\mathsf{a}}$ does a dangerous thing"), closing the following rule under parallel composition and restriction.

$$\{!b[\&_i^{\mathsf{a}_i}(\vec{y}_i).P_i]\}_{\mathsf{a}} \mid \{\overline{b}\mathtt{in}_k\langle\vec{V}\rangle\}_{\mathsf{a}'} \stackrel{Aerr}{\longrightarrow} \text{ if } \mathsf{a}_k \not\sqsubseteq \mathsf{a}',$$

We outline how we can prevent processes from "doing dangerous things" by static typing. Typewise, there are two places we add annotations: (1) each branch in the replicated type (for its invoking level); and (2) a process type Δ, becoming a pair Δ,a. The composition $\odot$ is redefined as $\Delta_1,\mathsf{a}_1 \odot \Delta_2,\mathsf{a}_2 = \Delta_1 \odot \Delta_2, \mathsf{a}_1 \sqcup \mathsf{a}_2$. Further we have a covariant subtyping $\Delta,\mathsf{a} \le \Delta',\mathsf{a}'$ iff $\Delta \le \Delta'$ and $\mathsf{a} \sqsubseteq \mathsf{a}'$. Two judgements are involved:

$$\Gamma \vdash P : \Delta,\mathsf{a}, \qquad \text{and} \qquad \Gamma \vdash \{P\}_{\mathsf{a}} : \Delta$$

The former says the highest level of P's actions (calculated under Γ, which records the access levels of resources) is a. After $\Gamma \vdash P : \Delta,\mathsf{a}$ is statically checked, we can move to the second judgement, safely enclosing it with a-boundary. For the rest, $\mathtt{set}(\mathsf{a})\{P\}$ is typable only when the highest level of P is a, and the result is the lowest authority. ?-mode actions raise the current level. Finally, at the replicated-input, we ensure that the level of the body of each branch is the same as, or lower than, its invocation level: the resulting level should be more than the join of all invocation levels of its branches. For all other rules the authority level does not change.

We list the key properties of the typing system.

THEOREM 4.3. (subject reduction) *Suppose* $\Gamma \vdash \mathsf{P} : \Delta$. *Then* $\mathsf{P} \longrightarrow\!\!\!\!\!\rightarrow \mathsf{P}'$ *implies* $\Gamma \vdash \mathsf{P}' : \Delta$.

COROLLARY 4.4. (Access Safety) *Suppose* $\Gamma \vdash \mathsf{P} : \Delta$. *Then for all* P' *such that* $\mathsf{P} \longrightarrow\!\!\!\!\!\rightarrow \mathsf{P}'$, *we have* $\mathsf{P}' \stackrel{Aerr}{\not\longrightarrow}$.

We give one example of a typable process. Assume r (resp. g) has the level root (resp. guest) in Γ and let $\mathsf{guest} \sqsubseteq \mathsf{alice} \sqsubseteq \mathsf{root}$. Then

$$\Gamma \vdash !b[^{\mathsf{alice}}(y).\mathtt{set}(\mathsf{root})\{\overline{r}\langle y\rangle\} \;\&\; {}^{\mathsf{guest}}(y').\overline{g}\langle y'\rangle] : \Delta,\mathsf{a}$$

is type-safe iff $\mathsf{a} \sqsupseteq \mathsf{alice}$ (as far as given authority levels go), noting the sensitive operation at r is protected by set (we may raise a to root to mimic a signed code). From this we can derive $\{!b[^{\mathsf{alice}}(y).\mathtt{set}(\mathsf{root})\{\overline{r}\langle y\rangle\} \,\&\, {}^{\mathsf{guest}}(y').\overline{g}\langle y'\rangle]\}_{\mathsf{a}}$, obtaining the process with level a.

5. TYPE-BASED SECURITY FOR A LANGUAGE WITH CODE DISTRIBUTION

5.1 A Distributed Class Based Language

We consider a configuration consisting of multiple hosts (VMs) for a single high-level class-based language. Inter-host communication is done via remote method invocation, which may carry marshaled (serialised) code. Marshalling is done explicitly for arbitrary expressions, carrying local code (classes) as necessary, and are explicitly unmarshaled. Remote references will also be imported and exported dynamically in communication. For simplicity the language does not contain inheritance and exception, nor do we model distributed failure. The language is terse, but demonstrates some of the key aspects of code distribution in e.g. Java [25] and CLR [8], making its security analyses non-trivial.

Below we present the grammar of the language.

$$\begin{array}{lll} \text{(value)} & V & ::= 1,2,\ldots \mid x \mid o \mid \ulcorner e\,\mathtt{with}\,\mathtt{CT}\urcorner \\ \text{(exp)} & e & ::= V \mid \mathtt{succ}(e) \mid \mathtt{pred}(e) \mid \mathtt{skip} \\ & & \quad \mid \mathtt{new}\,\mathtt{C}(\vec{e}) \mid e.\mathtt{m}(\vec{e}) \mid e_1;\mathtt{return}\,e_2 \\ & & \quad \mid \mathtt{serialize}(e) \mid \mathtt{deserialize}(V) \\ & & \quad \mid x := V \mid e_1;e_2 \mid \mathtt{if}\,e\,\mathtt{then}\,e_1\,\mathtt{else}\,e_2 \\ & & \quad \mid \mathtt{while}\,e_1\,\mathtt{do}\,e_2 \mid \mathtt{let}\,x = e_1\,\mathtt{in}\,e_2 \\ & & \quad \mid \mathtt{let}\,x = !\,y\,\mathtt{in}\,e \mid \mathtt{newvar}\,x \mapsto V\,\mathtt{in}\,e \\ \text{(method)} & \mathtt{M} & ::= \mathtt{T}\,\mathtt{m}\,(\vec{\mathtt{T}}\,\vec{x})\{e;\} \\ \text{(class)} & \mathtt{CL} & ::= \mathtt{class}\,\mathtt{C}\,\{\vec{\mathtt{T}}\,\vec{x}\,;\,\vec{\mathtt{M}}\} \\ \text{(thread)} & \mathtt{P} & ::= \mathbf{0} \mid e \mid \mathtt{P}_1 \mid \mathtt{P}_2 \mid (\nu u)\mathtt{P} \\ \text{(host)} & \mathtt{H} & ::= l[\mathtt{P},\,\sigma,\,H,\,\mathtt{CT}] \\ \text{(network)} & \mathtt{N} & ::= \mathbf{0} \mid \mathtt{N}_1 \mid \mathtt{N}_2 \mid \mathtt{H} \mid (\nu u)\mathtt{N} \end{array}$$

The syntactic categories from (value) to (class) are for writable code (except o and $\ulcorner e\,\mathtt{with}\,\mathtt{CT}\urcorner$ illustrated later), while (thread), (host) and (network) are for run-time configurations. $x, y, \ldots$ range over both function and imperative (instance) variables.[5] $o, ..$ denote

[5] In expressions, y in $\mathtt{let}\,x = y\,\mathtt{in}\,e$ is a function variable, while y in $\mathtt{let}\,x = !y\,\mathtt{in}\,e$ and $\mathtt{newvar}\,y \mapsto V$ .. is an imperative variable.

Location (1)

```
Class MathCal
{
double z;
double twice{alice}(int x){
  z= x^x;
  return z;
}
double thrice{root}(int x){
  z= x^x^x;
  return z;
}
}
```

```
..
t = serialize(A);
w = r.run(t);
```

Code A:

```
double y = new MathCal(0);
myobj.ack();
double sum = 0.0;
for (int i=0; i<10000;i++)
  sum = y.twice(3)+sum;
return sum/1000;
```

Location (2)

```
double run{alice}(Thunk z)
{
return deserialize(z);
}
```

Figure 4: Code Passing with Class

object ids used only in operational semantics. `m` denotes a methods name; `C`, `D` are class names; `T` denotes either a primitive type (say `nat`) or a class name. l denotes a location name of a host. σ denotes an environment, a finite maps from imperative variables to values. H is an object heap, which is a finite map from object ids to $\vec{\mathtt{M}}$.

Expressions use `lets` and `newvar` for simpler static/dynamic semantics, to which standard syntax can be mapped. For example, "$x = \mathtt{new}\ \mathtt{C}(e)$" in Java becomes "$\mathtt{let}\ y = e\ \mathtt{in}\ \mathtt{let}\ z = \mathtt{new}\ C(y)\ \mathtt{in}\ x := z$". For legibility, however, we often use familar syntax in programs. Other expressions are standard and are illustrated by examples. The method decralation $\mathtt{T}\ \mathtt{m}\ (\vec{\mathtt{T}}\ \vec{x})\{e;\}$ declares a method `m` with the result type `T`, with functional variables $\vec{x}$ of types $\vec{\mathtt{T}}$ bound in e ($\vec{\mathtt{T}}\vec{x}$ means $\mathtt{T}_1 x_1 .. \mathtt{T}_n x_n$). The class declaration $\mathtt{class}\ \mathtt{C}\{\vec{\mathtt{T}}\ \vec{x}\ ;\ \vec{\mathtt{M}}\}$ introduces a class named `C` with instance variables $\vec{x}$ of types $\vec{\mathtt{T}}$, with $\vec{x}$ bound in $\vec{\mathtt{M}}$. We omit primitives not relevant to our present objective (see § 6).

The syntax is based on a core part of FJ [24] and MJ [7], using fine-grained constructs from [20]. The sole primitive for communication is the method invocation $e_1.\mathtt{m}(e_2)$: when e_1 evaluates to a remote object reference, this becomes inter-host communication. Further there are two primitives for packing and unpacking code.

***Serialisation*:** $\mathtt{serialize}(e)$ creates a thunk of an arbitrary expression e. The resulting thunk is written $\ulcorner e\ \mathtt{with}\ \mathtt{CT}\urcorner$ where `CT` is part of the local class table needed to execute e at a remote site.

***Deserialisation*:** $\mathtt{deserialize}(V)$ deserialises a thunk V, which in general immediately initiates its execution, using the runtime environment of a new host.

These primitives, combined with remote method invocation, offer both simplicity and generality. To illustrate their usage, Figure 4 shows an example of dynamic code passing, using a sugared syntax. A piece of code is sent from Location (1) to Location (2), with Location (1)'s local class called `MathCal`. The code is tiny, but the calculation is very costly so is better done in Location (2) (which e.g. has a high speed machine). In the commands in Location (1):

```
t = serialize(A); w = r.run(t);
```

The first line serializes code A (boxed below) and stores it in t. The second line invokes the method `run` of a remote object r at Location (2), sending out the newly created thunk. If the calculation finishes, the result is sent back to Location (1) and stored in w. `MathCal` should be sent along (or dynamically loaded) for the execution of the code. Also note Code A has a call `ack` to `myobj`, sending back an acknowledgement when it starts calculation, where we assume `myobj` is a variable bound by "`let`" preceding the given code. Annotations alice and root are discussed later.

5.2 Operational Semantics

The dynamics of programs/configurations is defined in two stages. First we have a judgement for reduction of a thread in a host.

$$(\nu\vec{u})(e,\ \sigma,\ H,\ \mathtt{CT}) \longrightarrow (\nu\vec{u}')(e',\ \sigma',\ H',\ \mathtt{CT}').$$

where $\vec{u}$ are newly generated object ids and references. σ and H are already explained. The class table `CT` at each host contains: (1) a common, fixed *foundation class table*, written `FCT`, and (2) a non-fixed local one. We first look at the rule for `new`.

$$\text{[NewC]}\ \frac{\mathtt{C} \mapsto \mathtt{class}\ \mathtt{C}\ \{\vec{\mathtt{T}}\ \vec{y}\ ;\ \vec{\mathtt{M}}\} \in \mathtt{CT} \qquad o, \vec{y}\ \text{fresh}}{\begin{array}{l}(\mathtt{let}\ x = \mathtt{new}\ \mathtt{C}(\vec{V})\ \mathtt{in}\ e_2, \sigma, H, \mathtt{CT}) \\ \quad \longrightarrow (\nu o\vec{y})(e_2\{o/x\}, \sigma \cup [\vec{y} \mapsto \vec{V}], H \cup [o \mapsto \vec{\mathtt{M}}], \mathtt{CT})\end{array}}$$

`new` creates a heap from a fresh object id o to methods $\vec{\mathtt{M}}$, and environments from $\vec{y}$ to values $\vec{V}$, which are referred from $\vec{\mathtt{M}}$; note unicity of o and y_i are ensured by the ν-operator. Next we look at how serialization and deserialization work. Below $\mathsf{fc}(e_1)$ returns free class names in e_1.

$$\text{[Ser]}\ \frac{\mathtt{CT} \subset (\mathtt{CT}' \setminus \mathtt{FCT}) \quad \mathsf{dom}(\mathtt{CT}) = \mathsf{fc}(e_1) \setminus \mathsf{dom}(\mathtt{FCT})}{\begin{array}{l}(\mathtt{let}\ x = \mathtt{serialize}(e_1)\ \mathtt{in}\ e_2, \sigma, H, \mathtt{CT}') \\ \quad \longrightarrow (e_2\{\ulcorner e_1\ \mathtt{with}\ \mathtt{CT}\urcorner/x\}, \sigma, H, \mathtt{CT}')\end{array}}$$

$$\text{[Deser]}\ (\mathtt{let}\ x = \mathtt{deserialize}(\ulcorner e_1\ \mathtt{with}\ \mathtt{CT}\urcorner)\ \mathtt{in}\ e_2, \sigma, H, \mathtt{CT}') \longrightarrow (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2, \sigma, H, \mathtt{CT}' \cup \mathtt{CT})$$

[Ser] creates a thunk of an expression after extracting free class names from it such as `MathCal` in Figure 4 (in the present syntax they are only used in `new`). [Deser] adds the carried classes to its class table, defreezes the code, and starts its evaluation. Here we assume class names are globally unique (which can be practically implementable using e.g. class fingerprints [25]).

Another judgement, written $\mathtt{N} \longrightarrow \mathtt{N}'$, represents communication among hosts as well as parallel execution of threads. The central rules for this judgement are method invocation, formulated so as to capture local and remote operations uniformly.

[Method Go]
$\mathtt{let}\ x = o.\mathtt{m}(\vec{V})\ \mathtt{in}\ e \longrightarrow (\nu c)(o.\mathtt{m}(\vec{V})\ \mathtt{to}\ c \mid \mathtt{wait}\ x\ \mathtt{at}\ c\ \mathtt{in}\ e) \quad (c\ \text{fresh})$

[Method Invocation]
$$\frac{o \mapsto \vec{\mathtt{M}} \in H \qquad \mathtt{M}_i = \mathtt{T}\ \mathtt{m}(\vec{\mathtt{T}}\ \vec{x})\{e;\}}{(o.\mathtt{m}(\vec{V})\ \mathtt{to}\ c, \sigma, H) \longrightarrow (e\{\vec{V}/\vec{x}\}\{\mathtt{return}\ e\ \mathtt{to}\ c/\ \mathtt{return}\ e\}, \sigma, H)}$$

[Return] $(\mathtt{return}\ V\ \mathtt{to}\ c \mid \mathtt{wait}\ x\ \mathtt{at}\ c\ \mathtt{in}\ e) \longrightarrow e\{V/x\}$

In [Method Go], a method call creates a message to that object and a continuation code; When the former meets the object, invocation takes place in [Method Invocation]. After calculation, the result may return to the initial continuation code in [Return]. For remote invocation, we combine these rules with migration.

[Method Migration]
$l_1[o.\mathtt{m}(\vec{V})\ \mathtt{to}\ c \mid \mathsf{P}_1,\sigma_1,H_1,\mathtt{CT}_1] \mid l_2[\mathsf{P}_2,\sigma_2,H_2,\mathtt{CT}_2]$
$\longrightarrow l_1[\mathsf{P}_1,\sigma_1,H_1,\mathtt{CT}_1] \mid l_2[o.\mathtt{m}(\vec{V})\ \mathtt{to}\ c \mid \mathsf{P}_2,\sigma_2,H_2,\mathtt{CT}_2] \quad (o \in \mathsf{dom}(H_2))$

[Remote Return]
$l_1[\mathtt{return}\ V\ \mathtt{to}\ c \mid \mathsf{P}_1,\sigma_1,H_1,\mathtt{CT}_1] \mid l_2[\mathsf{P}_2,\sigma_2,H_2,\mathtt{CT}_2]$
$\longrightarrow l_1[\mathsf{P}_1,\sigma_1,H_1,\mathtt{CT}_1] \mid l_2[\mathtt{return}\ V\ \mathtt{to}\ c \mid \mathsf{P}_2,\sigma_2,H_2,\mathtt{CT}_2] \quad (c \in \mathsf{fn}(\mathsf{P}_2))$

where a message/return migrates to a target if the latter is not local.

5.3 Types and Typing

Assuming the signature (interfaces) for remotely exported classes are shared by all hosts in the network, the type discipline for the language is standard [24] except for (inferred) types representing access to objects and serialised code.

Let us illustrate these two ideas using the example in Figure 4. Assume `myobj` in Location (1) is a method of class `Ack` and `run` belongs that of class `Defreeze`. We can then assign a fine-grained interface $\Delta = \{\mathtt{myobj.ack}:\mathtt{Ack}\}$ to Code A, describing its possible access to objects. Then we can type the thunk of code A as given by the existentially abstracted triple:

$$\exists[\mathtt{myobj}:\mathtt{Ack}]\ulcorner\mathtt{double};\Delta;\mathtt{MathCal}\urcorner$$

This type indicates: (1) it returns (evaluates to) `double`; (2) its access to other objects are at most $\{\mathtt{myobj.ack}:\mathtt{Ack}\}$ where (by quantification) `myobj` is anonymous; and (3) it carries a local class `MathCal`, offering clean abstraction of access of the code to other resources. To illustrate the type discipline for driving such types, first we introduce the syntax of value types.

$$\mathtt{T} \quad ::= \quad \mathtt{C} \mid \mathtt{nat} \mid \mathtt{void} \mid \exists[\vec{x}:\vec{\mathtt{C}}]\ulcorner\mathtt{T};\{\vec{u}:\vec{\mathtt{C}}\};\vec{\mathtt{D}}\urcorner$$

Types are either *class names*, *base types* or *thunk types*. In a thunk type, whose syntax is deliberately close to channel existential types in $\lambda\pi^{\mathsf{LAM}}_{<:}$, u_j in $\vec{u}$ has the form either x_i or $x_i.\mathtt{m}$, and each x_i is bound by $\exists$, so that $\mathsf{dom}(\Delta)/\{\vec{x}\} = \emptyset$.[6] $\vec{\mathtt{D}}$ denotes the names of the classes sent with code. The main typing judgements are:

$$E \vdash e : \rho,\Delta \qquad \text{and} \qquad E \vdash \mathtt{N} : \mathtt{net},\Delta$$

where E is a mapping from identifiers to either value types, reference types (of the form $\mathtt{ref}(T)$, for `newvar` and instance variables), a pair type for `deserialize` and channel types (for `return`). ρ is either the value type `T` or the command type `cmd`. Δ represents an effect of e, that is a finite mapping from x to a reference type or u to class names. The typing rules follow, on the one hand, the standard class-based languages [24, 34, 7] and, on the other, the rules for $\lambda\pi$ in Figure 3 in Section 2. Since the former is based on structural typing (in the sense of [34]) while the latter on nominal one, we bridge them by using a fixed signature for class tables. Leaving the full presentation to [54], we list the rules for serialization and deserialization.

$$[\mathit{Seri}]\ \frac{E \vdash e : \mathtt{T},\Delta \quad e \neq o \quad \mathsf{fv}(\Delta) = \emptyset \qquad E \vdash \mathtt{CT\ OK} \quad E,x{:}\exists[\vec{x}:\vec{\mathtt{C}}]\ulcorner\mathtt{T};\Delta;\vec{\mathtt{D}}\urcorner \vdash e' : \rho,\Delta' \quad \mathsf{dom}(\mathtt{CT}) = \mathsf{fc}(e) \setminus \mathsf{dom}(\mathtt{FCT})}{E \vdash \mathtt{let}\ x = \mathtt{serialize}\ (e)\ \mathtt{in}\ e' : \rho,\ \Delta'/x}$$

$$[\mathit{Deser}]\ \frac{E \vdash V : \exists[\vec{x}:\vec{\mathtt{C}}]\ulcorner\mathtt{T};\Delta;\vec{\mathtt{D}}\urcorner \quad E,\{\vec{x}:\vec{\mathtt{C}},y:(\mathtt{T},\Delta)\} \vdash e_2 : \rho,\Delta_2,\vec{x}:\vec{\mathtt{C}}}{E \vdash \mathtt{let}\ y = \mathtt{deserialize}\ (V)\ \mathtt{in}\ e_2 : \rho,\ \Delta_2/y}$$

[6] Describing access interface to non-anonymous objects may not make sense for static type inference. See [54] for more discussions.

Location (1) alice

```
..
t = serialize(B);
w = r.run(t);
```

Code B:

```
double y = new MathCal(0);
double[alice] sum = 0.0;
for (int i=0; i<10000;i++)
  sum = y.twice(3)+sum;
return sum/1000;
```

Location (2)

```
double run{alice}(Thunk x)
{
temp = deserialize(x);
file.check(temp);
return temp;
}

void check{alice}(double x)
 {nat y[guest] = 0;
  if x = 0.0 then
     y = 1;
  return;}
```

Figure 5: Unsafe Code Passing (1)

Location (1) Alice

```
..
t = serialize(C);
w = r.run(t);
```

Code C:

```
y = new NaiveLibrary();
y.cleanup(''..\\passwd'');
return;
```

Location (2) Root

```
double run{alice}(Thunk x)
{
return deserialize(x);
}

class NaiveLibrary{root}{
void cleanup(string s)
{(new File).delete(s);
 return;}}
```

Figure 6: Unsafe Code Passing (2)

[*Seri*] creates a thunk of e with a well-formed class table `CT` which is the part of the local class table needed to execute e. [*Deser*] deserialises the thunk V with its locally bound names $\vec{x}$ by opening the scope of y. $\{\ \}$ denotes an existential packing [54]. It would be useful to look at the correspondence of this rule with (In) in Figure 3, considering, for example, the following (simplified) encoding of $(\nu o)(\mathtt{let}\ y = \mathtt{deserialize}\ (V)\ \mathtt{in}\ y,\ \emptyset,[o \mapsto \vec{\mathtt{M}}],\emptyset)$ into $\lambda\pi^{\mathsf{LAM}}_{<:}$.

$$(\nu c)((\nu o)(\overline{c}\langle\ulcorner\vec{e}\urcorner\rangle \mid !o[\&_i(z_i).R_i]) \mid c(y:\tau).\mathit{run}\,y) \qquad (13)$$

where "$!o[\&_i(z_i).R_i]$" represents a heap $[o \mapsto \vec{\mathtt{M}}]$, while "$\mathit{run}\,y$" represents y; o is hidden assuming it is generated by `new`. To accept V, c in "$c(x:\tau).\mathit{run}\,x$" should have type "$(\exists[x:\sigma]\ulcorner\Delta,x:\sigma\urcorner)^{\mathtt{I}}$", which can be derived from:

$$\Gamma,\{x:\sigma,y:\ulcorner\Delta,x:\sigma\urcorner\} \vdash \mathit{run}\,y : \Delta,x:\sigma$$

Here "$\{x:\sigma,y:\ulcorner\Delta,x:\sigma\urcorner\}$" represents a pack of name x and variable y; and x should appear only in τ, cf.§ 2.4. We note that the reduction of this encoding is in tight correspondence with the original command with respect to the rule [*Deser*] in the previous subsection, which extend to the full encoding of the language into $\lambda\pi^{\mathsf{LAM}}_{<:}$ (which follows [20], see [54]). A notable point is that, in the encoding, the class names of `FCT` (interpreted as channel names in $\lambda\pi^{\mathsf{LAM}}_{<:}$) are ν-bound at each host, which are instantiated into the code whenever the latter is received at a host, using function dependent types.

5.4 Authority and Secrecy

For instrumenting the syntax and type discipline for capturing the security concerns along the line of our discussion in Section 4, we first add slight annotations to program texts as follows. Below we

treat both secrecy and authority simultaneously.

$$\begin{array}{lcl} e & ::= & \ldots \mid \texttt{newvar}\ x\{\mathsf{a}_r,\mathsf{a}_w\}[s] \mapsto V\ \texttt{in}\ e \\ & \mid & \texttt{set}(\mathsf{a})\{e\} \mid \{e\}_{\mathsf{a}} \\ \texttt{P} & ::= & \ldots \mid \{e\}_{\mathsf{a}} \\ \texttt{M} & ::= & \texttt{T m}\{\mathsf{a}\}[s]\ \texttt{return}\,[s']\ \ (\vec{\texttt{T}}\,\vec{x})\{e;\} \\ \texttt{CL} & ::= & \texttt{class C}\,\{\mathsf{a}\}[s]\ \{\vec{\texttt{T}}\{\vec{\mathsf{a}}_r,\vec{\mathsf{a}}_w\}[\vec{s}]\ \ \vec{x}\ ;\ \vec{\texttt{M}}\} \end{array}$$

$s, s', \ldots$ range over secrecy levels; $\mathsf{a}, \mathsf{a}', \ldots$ range over authority levels. In `newvar` and instance variables, we annotate them by read/write authority levels $\{\mathsf{a}_r, \mathsf{a}_w\}$, and the secrecy level, $[s]$ (written $\texttt{int}\{s\}\,x$ in Jflow [31]). The authority/secrecy levels of the methods are declared in $\{\mathsf{a}\}[s]$ (like `caller(a)` in [31]), together with the secrecy level for the final return value, $\texttt{return}\,[s']$. For classes, if `C` is annotated by a, then `C` can be created by the authority higher than a. The authority and security level annotations are optional, and are set to the lowest levels as default. $\texttt{set}(\mathsf{a})\{e\}$ sets e to the level a.

Secrecy annotations aim to control the levels of received and emitted information, while authority annotations aim to control the access levels of significant operations assuming the mechanism of stack inspection. We illustrate them using two simple examples, Figures 5 (for secrecy) and 6 (for authority). Assume $\mathsf{guest} \sqsubseteq \mathsf{alice} \sqsubseteq \mathsf{root}$ for both secrecy and authority levels. Then the code in Figure 5 does not violate access control annotations but leaks the sensitive data from alice to guest, assuming y in `check` has level guest. In this example, first alice calls `twice`, which does not violate the guard in `MathCal`. Figure 6 is a classical example (taken from [3, § 4.1]) which shows how a call of trusted code by an untrusted code results in access violation by performing a sensitive operation (i.e. delete a file). Combined with authority annotations for object creation and references, many of those examples reported in [3] as demonstrating the limitations of the current stack inspection framework may also be treated (see [54]).

For access control, we also need to instrument operational semantics. An expression (thread) (resp. object, resp. class) is now surrounded by a frame. The reduction of expressions is given following λ_{sec} [11, 36] and that of $\lambda\pi^{\mathsf{LAM}}_{<}$.

$$\texttt{set}(\mathsf{a})\{e\} \longrightarrow \{e\}_{\mathsf{a}} \qquad \{V\}_{\mathsf{a}} \longrightarrow V \quad \{\texttt{skip}\}_{\mathsf{a}} \longrightarrow \texttt{skip}$$

Access errors are formalised as:

$$(\{\texttt{let}\ x = \texttt{new}\ \texttt{C}(\vec{V})\ \texttt{in}\ e_2\}_{\mathsf{a}}, \sigma, H, \texttt{CT}) \xrightarrow{Aerr} \quad \text{with } \texttt{C} \mapsto \texttt{class C}\,\{\mathsf{a}'\}\,\{\vec{\texttt{T}}\,\vec{x}\,;\,\vec{\texttt{M}}\} \in \texttt{CT} \text{ and } \mathsf{a}' \not\sqsubseteq \mathsf{a}$$

$$(\{o.\texttt{m}(\vec{V})\ \texttt{to}\ c\}_{\mathsf{a}}, \sigma, H) \xrightarrow{Aerr} \quad \text{with } o \mapsto \vec{\texttt{M}} \in H,\ \texttt{M}_i = \texttt{T m}\{\mathsf{a}'\}(\vec{\texttt{T}}\,\vec{x}) \text{ and } \mathsf{a}' \not\sqsubseteq \mathsf{a}$$

We also add the corresponding rules for read/write of a reference (these rules follow the rule given in § 4.3 by regarding classes, objects and references as replicated processes).

Type Inference. The inferable value types are annotated with secrecy and authority levels as follows.

$$\texttt{T} \ ::= \ \texttt{C} \mid \texttt{nat}_s \mid \texttt{void} \mid \exists[\vec{x}{:}\vec{\texttt{C}}]^{\lceil}\texttt{T}; \{\vec{u}{:}\vec{\texttt{C}}, \mathsf{a}\}; \vec{\texttt{D}}^{\rceil}$$

where a in the thunk type is the authority level of effects (which corresponds to a of "Δ, a" in $\lambda\pi^{\mathsf{LAM}}_{<}$ in § 4.3). The secrecy typing, whose typability is written $E \vdash e : \rho, \Delta, \mathsf{a}$ and $E \vdash \texttt{N} : \texttt{net}, \Delta$, follows [20] (with simplification for the sequential part exploiting the recorded effects represented by interface type). The authority typing follows the framework in § 4.3, again treating objects and classes as if they were replicated process in $\lambda\pi$ (e.g. the authority level of each body in the method should be less than that of its guard). The full typing rules are given in [54]. Here we illustrate the key ideas by revisiting Figures 5 and 6. In Figure 5, the body of method is immediately untypable by a non-trivial flow from x to y. The program in Figure 6 is also untypable by the guard of the authority level root in `run`. For the root to accept and defreezing this malicious code, the annotated type `Thunk` should have Δ, root for some Δ, so the method `run` is untypable since the authority level of $\texttt{deserialize}(\texttt{x})$ should be lower than alice.

The following results underpin the presented type-based analyses. The first result is about access control via subject reduction, which in turn is verified via tight correspondence of each reduction rule between the source program and the encoding.

THEOREM 5.1. (Access Safety) *Let* $E \vdash \texttt{N} : \texttt{net}, \Delta$. *Then for all* $\texttt{N}'$ *such that* $\texttt{N} \twoheadrightarrow \texttt{N}'$, *we have* $\texttt{N}' \not\xrightarrow{Aerr}$.

For secrecy, to define an observable at the level of network, we take a return of void type. Below s at a return channel c comes from the return value annotation in program text.

$$\texttt{N} \Downarrow_{\langle c,s\rangle} \stackrel{\text{def}}{\Leftrightarrow} \texttt{N} \twoheadrightarrow (\nu\,\vec{u})(l[(\texttt{return to}\ c^s)\,|\texttt{P},\ \sigma, H,\ \texttt{CT}]\,|\texttt{N}')$$

with $c \notin \{\vec{u}\}$. Further write $\sigma_1 \sim_s \sigma_2$ for $\mathsf{dom}(\sigma_1) = \mathsf{dom}(\sigma_2)$ if $\sigma_1(x^{s'}) = \sigma_2(x^{s'})$ for each $x \in \mathsf{dom}(\Delta)$ such that $s' \sqsubseteq s$, assuming each instance variable is annotated with the declared secrecy level. Below $\texttt{N}[\sigma@l]$ denotes a network N in which σ is substituted for the environment of a host in N named l. We assume reduction does not α-convert bound names in the environment.

THEOREM 5.2. (Noninterference) *Let* $E \vdash \texttt{N}[\sigma_i@l] : \texttt{net}, \Delta$ *such that* $\sigma_1 \sim_s \sigma_2$. *Then* $\texttt{N}[\sigma_1@l] \Downarrow_{\langle c,s\rangle}$ *iff* $\texttt{N}[\sigma_2@l] \Downarrow_{\langle c,s\rangle}$.

The proof is essentially direct from Theorem 4.2 via the encoding.

6. RELATED WORK/FURTHER TOPICS

There have been a wide range of studies of mobile and distributed computation on process calculi [37, 38, 41, 43, 9]. Among those closely related to the present work, Dπ [37] proposed a framework where partial type checking prevents migration of code from untrusted locations using location-based types. M-calculus [43] is a typed version of the higher-order distributed join calculus with hierarchical locations where the type represents access rights of a process as a set of channels, which is more closely related to our early work in [52]; the channel types in [43] can control higher-order communication and guarantee the unicity of local names (which is essentially similar condition guaranteed by our replicated affine channels). The typing framework with functional/channel dependent and existential types in the present work allows more detailed specification of access rights (as shown in § 4 and 5) and a more general form of parameterised code with fine-grained process types. [15] shows the new dependent/existential types in this paper can statically obtain the dynamic program detection enabled by partial type checking in [37] in a non-trivial higher-order extension of Dπ. See also [15] for more comparisons with other typed distributed calculi.

The boxed π-calculus [41] studies a causality based typing system to control dynamic composition of untrusted components; they annotate output processes with a set of principals and their causal flow is analysed by coloured lts. The work does not include higher-order process passing, which is the main aim of using the $\lambda\pi$ in the present work. In distributed mobility, [9, 27] proved the noninterference result with May equivalence (which is a weaker property than our noninterference stated in Theorem 4.2) in the boxed ambient and Facile based language, respectively. Our framework differs

from them in that they do not directly pursue an integration of information flow analysis and authorised access control, while we aim at offering an integrated modelling framework which can precisely capture language semantics for distributed networked computation. The typability of our system is larger than that in [9, 27] due to the distinction between linearity and affinity (for example, see § 4.1), which is crucial to preserve the semantics of programming phrases and secrecy analysis (also see §.8 in [20] for more comparison with other work).

The SLam-calculus [14] is one of the first work which integrates secrecy and integrity concerns in a single formalism. The sequential part of SLam is encodable into Dependency Core Calculus [2] which in turn can be encoded into the linear/affine π-calculus [20] (a subcalculus of the $\lambda\pi^{\mathsf{LAM}}_{<}$). The concurrency framework in SLam is based on shared variables without communication on channels, while, in π and $\lambda\pi$, shared variables are simulated using interaction. The development in § 4 and 5 may suggest that our approach would scale more easily to the modelling of realistic distributed languages than the approach based on shared-variable concurrency.

Jflow [31] designs a secrecy type discipline for a major subset of Java based on decentralised labels [30] and studies its distributed implementation involving possibly untrusted hosts [55]. We believe parameterising classes by authority levels in [31] would be easy to integrate into our framework by representing class types by dependent types. [31] also uses a non-trivial mechanism, *declassification* (which allows lowering of a secrecy level of a datum in a restricted situation) to allow programmers to write realistic code while ensuring effective secrecy. Incorporation of their primitives, as well as investigation for automatic elaboration of secrecy/authority level annotations would be practically important and is a desirable further study.

The access control mechanisms for extensible language architectures such as Java [25] and CLR [8] are studied in [3, 12, 36, 45, 46]. In comparison with the studies based on the λ-calculus [11, 36], $\lambda\pi$ can uniformly treat access control in the presence of, among others, name/code-parameterised code, channel-based interaction and concurrency, which would be difficult in the function-based model. The present work has also shown that the use of linear/affine type structure allows us to faithfully model stack inspection mechanism (we expect we have a fully abstract encoding of λ_{sec} in a sequential subset of $\lambda\pi^{\mathsf{LAM}}_{<}$ [5, 6, 20, 49, 50, 53]). The presented type discipline captures access to resources as static interface. On its basis, incorporation of more refined behavioural specifications, such as treatment of dynamic resource usage patterns studied by [22], would be feasible. The use of higher-order mobile processes would lead to an integrated framework for specifying, controlling and verifying behavioural safety in the presence of mobile code.

The class-based language we have embedded in the present work does not include such language features as recursively defined classes, inheritance, interface and casting (all of which are found, for example, in FJ [24] and MJ [7]), as well as polymorphism, synchronisation and reflection. Some of these features can be represented by straightforward enrichment of the presented type structure such as branching subtyping and recursive types. We also believe incorporation of second-order polymorphism along the line of [6] would not be technically difficult. For modelling object-oriented features in general, we can effectively use preceding studies on objects and their types cf.[34] in the context of the λ-calculus via the linear/affine type structures.

In the foregoing studies of the higher-order mobile processes [26, 38, 39], tractable higher-order behavioural equivalences are defined via encodings of a higher-order calculus into the first-order one. The type-preserving encoding proposed in § 4.2 shows this methodology can be inherited in the setting of strong type discipline, with an added benefit of more faithful operational correspondence including preserving localities and effects of mobile code [41, 47], which may be useful for various applications beyond what we have treated in this paper.

7. REFERENCES

[1] Abadi, M., Secrecy by Typing in Security Protocols, *TACS'97*, LNCS 1281, pp.611-638, Springer-Verlag, 1997.

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

[3] Abadi, M. and Fornet, C., Access Control based on Execution History, *NDSS'03*, California, 2003.

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

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

[6] Berger, M., Honda, K. and Yoshida, N., Genericity and the π-Calculus, *FoSSaCs'03 (ETAPS'03)*, LNCS, Springer, 2003.

[7] Bierman, G.M., Parkinson, M.J and Pitts, A.M., MJ: an imperative core calculus for Java and Java with effects, UCAM-CL-TR-563, Cambridge, 2003.

[8] Box, D., *Essential .Net Volume I: The Common Language Runtime*. Addison Wesley, 2002.

[9] Crafa, S., Bugliesi, M. and Castagna, G., Information Flow Security in Boxed Ambients, ENTCS 66 No. 3, 2003.

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

[11] Fornet, C. and Gordon, A.D., Stack Inspection: theory and variants, POPL, ACM, 2002.

[12] Grimm, R. and Bershad, B., Separating Access Control Policy, Enforcement, and Functionality in Extensible Systems, ACM Tra. on Comp. Sys., 19(1), Feb., 36–70, 2001.

[13] Gong, L., *Inside Java TM 2 Platform Security*, Addison Wesley, 1999.

[14] Heintze, N. and Riecke, J., The SLam Calculus: Programming with Secrecy and Integrity, *POPL'98*, pp.365-377. ACM Press, 1998.

[15] Hennessy, M., Rathke, J. and Yoshida, N, SafeDpi: a language for controlling mobile code, 2003. To appear as COGS Technical Report, 2003.

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

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

[18] Honda, K. and Yoshida, N., On Reduction-Based Process Semantics. *TCS*, pp.437–486, No.151, North-Holland, 1995.

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

[20] Honda, K. and Yoshida, N., A Uniform Type Structure for Secure Information Flow, *POPL'02*, 81–92, ACM Press, 2002. The full version is to appear as a DOC technical report, Department of Computing, Imperial College, University of London, August, 2002. Available at: `www.doc.ic.ac.uk/ yoshdia`.

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

[22] Igarashi, A. and Kobayashi, N., Resource Usage Analysis, POPL'02, 331-342, Portland, OR, ACM, 2002.

[23] Igarashi, A. and Kobayashi, N., A Generic Type System for the Pi-Calculus, POPL'01, 128-141, ACM, 2001.

[24] Igarashi, A., Pierce, B. and Wadler, P., Featherweight Java: A Minimal Core Calculus for Java and GJ. TOPLAS, 23(3):396-450, May 2001.

[25] Java, Sun Microsystems Inc., http://www.javasoft.com/.

[26] Jeffrey, A. and Rathke, J., Contextual equivalence for higher-order pi-calculus revisited, MFPS XIX, Montreal, 2003.

[27] Kirli, D., Confined Mobile Functions, 14th CSFW, 2001.

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

[29] Microsoft Corporation, .NET Framework Developer's Guide, http://msdn.microsoft.com, 2003.

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

[31] Myers, M., JFlow: Practical mostly-static information flow control. POPL'99, 228–241, 1999.

[32] Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes. *Infor. & Comp.*, 100(1), pp.1–77, 1992.

[33] Necula, G., Proof-carrying code. *POPL'96*, ACM, 1996.

[34] Pierce, B.C., *Types and Programming Languages*, MIT, 2002.

[35] Pierce, B.C. and Sangiorgi. D, Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.

[36] Pottier, F., Skalka, C. and Smith, S., A systematic approach to access control. *ESOP01*, LNCS 30–45, Springer, 2001.

[37] Riely, J. and Hennessy, M., Trust and partial typing in open systems of mobile agents. *POPL'99*, ACM Press, 1999.

[38] Sangiorgi, D., *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms*. Ph.D. Thesis, University of Edinburgh, CST-99-93, 1993.

[39] Sangiorgi, D., Asynchronous process calculi: the first-order and higher-order paradigms, TCS, 1999.

[40] Sangiorgi, D., Termination of Processes, CONCUR'02, 2002.

[41] Sewell, P. and Vitek, J., Secure Composition of Untrusted Code: Box pi, Wrappers, and Causality Types. 13th CSFW, 2000. To appear in Journal of Computer Security.

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

[43] Schmitt, A. and Stefani, J.-B., The M-calculus: A Higher-Order Distributed Process Calculus, POPL03, ACM, 2003.

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

[45] Wallach, D.S, et al., Extensible security architectures for Java. SOSP, 116–128, IEEE, 1997.

[46] Wallach, D.S, et al., Safkasi: A security mechanism for language-based systems. *ACM Tra. Sof. Eng. Meth.*, 9(4):341–378, 2000.

[47] Unypoth, A. and Sewell, P., Nomadic Pict: Correct Communication Infrastructure for Mobile Computation, POPL01, ACM, 2001.

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

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

[50] Yoshida, N., Berger, M. and Honda, K., Strong Normalisation in the π-Calculus, *LICS'01*, 311–322, IEEE, 2001. The revised full version as DOC technical report, 2003-01, Imperial College London, 2003. . Available at www.dos.ic.ac.uk/~yoshida.

[51] Yoshida, N. and Hennessy, M., Subtyping and Locality in Distributed Higher Order Processes. *CONCUR'99*, LNCS 1664, pp.557–573, Springer, 1999. CS Technical Report 01/99, University of Sussex, Available at: http://www.cogs.susx.ac.uk, April, 1999.

[52] Yoshida, N. and Hennessy, M., Assigning Types to Processes, *Info. &. Comp.*, Volume 174, Issue 2, 143-179, Academic Press, 2002. Extended abstract appeared in *LICS'00*.

[53] Yoshida, N., Honda, K. and Berger, M. Linearity and Bisimulation, FoSSaCs 2002, LNCS 2303, pp.417–433, Springer, 2002.

[54] Appendix of this paper. Available at www.doc.ic.ac.uk/~yoshida.

[55] Zdancewic, S. Zheng, L., Nystrom N., Myers, A., Untrusted Hosts and Confidentiality: Secure Program Partitioning, *SOSP01*, ACM, 2001.

(Reduction)

$$(\beta)\quad (\lambda(x{:}\tau)P)\,V \longrightarrow P\{V/x\}$$

$$(\mathrm{app}_r)\ \frac{P \longrightarrow P'}{P\,Q \longrightarrow P'\,Q} \qquad (\mathrm{app}_l)\ \frac{Q \longrightarrow Q'}{V\,Q \longrightarrow V\,Q'}$$

$$(\mathrm{com})\ a(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).P \mid \overline{a}\langle V_1,\ldots,V_n\rangle \longrightarrow P\{V_1,\ldots,V_n/x_1,\ldots,x_n\}$$

$$(\mathrm{rep})\ !a(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).P \mid \overline{a}\langle V_1,\ldots,V_n\rangle \longrightarrow P\{V_1,\ldots,V_n/x_1,\ldots,x_n\} \mid !a(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).P$$

$$(\mathrm{par})\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad (\mathrm{res})\ \frac{P \longrightarrow P'}{(\nu\, a{:}\sigma)P \longrightarrow (\nu\, a{:}\sigma)P'}$$

$$(\mathrm{str})\ \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

(Structure Equivalence)

- $P \equiv Q$ if $P \equiv_\alpha Q$.
- $P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad P \mid \mathbf{0} \equiv P$
- $(\nu\, a{:}\sigma)\mathbf{0} \equiv \mathbf{0}$
  $(\nu\, a{:}\sigma)P \mid Q \equiv (\nu\, a{:}\sigma)(P \mid Q)$ if $a \notin \mathsf{fn}(Q)$
  $(\nu\, a{:}\sigma)(\nu\, b{:}\sigma')P \equiv (\nu\, b{:}\sigma')(\nu\, a{:}\sigma)P$ if $a \notin \mathsf{fn}(\sigma')$ and $b \notin \mathsf{fn}(\sigma)$

Figure 7: Reduction

APPENDIX

A. SECTION 2

A.1 Reduction and Types

This subsection lists the full definitions corresponding to § 2.1 and § 2.2. First we define the reduction relation of $\lambda\pi$ in Figure 7 and the set of free names and variables in Figure 8. Then following [5, 20, 17, 16, 20, 50], we define duality of types and a partial algebra on modes/types in order to control composability of processes. By refining these operators, we can give more expressive notions of type constraints. We define:

(m-1) $\overline{p}$, the *dual* of p, is given by: $\overline{\mathtt{I}} = \mathtt{O}$ and $\overline{\mathtt{O}} = \mathtt{I}$.

(m-2) A partial commutative operator $\odot$ is generated from: $p \odot p = p$ and $\mathtt{I} \odot \mathtt{O} = \mathtt{IO}$ where $\mathtt{IO}$ denotes input and output mode.

The *dual of* σ, denoted by $\overline{\sigma}$, is the result of dualising all action modes of channel types as follows.

$$\overline{(\Pi[x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n]\tau_1,\ldots,\tau_m)^p} = (\Pi[x_1{:}\overline{\sigma_1},\ldots,x_n{:}\overline{\sigma_n}]\tau_1,\ldots,\tau_m)^{\overline{p}}$$

We set *the mode of* σ, denoted by $\mathsf{md}(\sigma)$, by:

$$\mathsf{md}((\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^p) = p \qquad \mathsf{md}(\langle\sigma,\sigma'\rangle) = \mathsf{md}(\sigma) \odot \mathsf{md}(\sigma')$$

Then a partial commutative operator $\odot$ on types is defined as:

(t-1) if $\mathsf{md}(\sigma) = \mathtt{O}$, then $\sigma \odot \overline{\sigma} = \langle\sigma,\overline{\sigma}\rangle = \langle\overline{\sigma},\sigma\rangle$ and;

(t-2) $\sigma \odot \sigma = \sigma$, $\sigma \odot \langle\sigma,\sigma'\rangle = \sigma \odot \langle\sigma',\sigma\rangle = \langle\sigma,\sigma'\rangle = \langle\sigma',\sigma\rangle$.

We extend this operation to channel environments point-wise: $\Delta_1 \asymp \Delta_2$ if $x{:}\sigma_1 \in \Delta_1$ and $x{:}\sigma_2 \in \Delta_2$ imply $\sigma_1 \odot \sigma_2$ is defined. If $\Delta_1 \asymp \Delta_2$, then we set:

$$\Delta_1 \odot \Delta_2 \stackrel{\text{def}}{=} \{x{:}\sigma \mid x{:}\sigma_1 \in \Delta_1, x{:}\sigma_2 \in \Delta_2, \sigma = \sigma_1 \odot \sigma_2\} \cup (\Delta_1 \setminus \mathsf{dom}(\Delta_2)) \cup (\Delta_2 \setminus \mathsf{dom}(\Delta_1))$$

Note this operation $\odot$ corresponds to the least upper bound operator $\sqcup$ in [52]. Using this operator, the substitution is formally defined in Figure 9.

(Free Variables) We assume $\mathsf{fv}(x) = \{x\}$, $\mathsf{fv}(a) = \emptyset$.

$\mathsf{fv}(\mathtt{unit}) = \mathsf{fv}(\mathtt{nat}) = \emptyset$

$\mathsf{fv}(\tau \to \tau') = \mathsf{fv}(\tau) \cup \mathsf{fv}(\tau')$

$\mathsf{fv}(\Pi(x{:}\sigma)\tau) = \mathsf{fv}(\sigma) \cup \mathsf{fv}(\tau) \setminus \{x\}$

$\mathsf{fv}(\Delta) = \{\mathsf{fv}(u) \cup \mathsf{fv}(\tau) \mid u{:}\tau \in \Delta\}$

$\mathsf{fv}((\Pi[x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n]\tau_1,\ldots,\tau_m)^p) = (\mathsf{fv}(\sigma_1)\ldots\cup\mathsf{fv}(\sigma_n)\cup\mathsf{fv}(\tau_1)\ldots\cup\mathsf{fv}(\tau_m)) \setminus \{x_1,\ldots,x_n\}$

$\mathsf{fv}(\langle\sigma,\sigma'\rangle) = \mathsf{fv}(\sigma) \cup \mathsf{fv}(\sigma')$

(Free Names) We assume $\mathsf{fn}(x) = \emptyset$, $\mathsf{fn}(a) = \{a\}$.

$\mathsf{fn}(\Pi(x{:}\sigma)\tau) = \mathsf{fn}(\sigma) \cup \mathsf{fn}(\tau)$

$\mathsf{fn}((\Pi[x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n]\tau_1,\ldots,\tau_m)^p) = \mathsf{fn}(\sigma_1)\ldots\cup\mathsf{fn}(\sigma_n)\cup\mathsf{fn}(\tau_1)\ldots\cup\mathsf{fn}(\tau_m)$

Other rules are given by replacing $\mathsf{fv}(\ )$ by $\mathsf{fn}(\ )$.

(Free Names and Variables) $\mathsf{fnv}(\tau) = \mathsf{fn}(\tau) \cup \mathsf{fv}(\tau)$

Figure 8: Free Variables and Names on Types

A.2 Typing

We define the notions of well-formedness for types and environments by the formal system in Figure 10 where we use the following four forms of judgements:

$\Gamma \vdash \mathtt{Env}$	a well-formed environment
$\Gamma \vdash \tau : \mathtt{tp}_H$	a well-formed HO-value type in Γ
$\Gamma \vdash \tau : \mathtt{tp}_P$	a well-formed process type in Γ
$\Gamma \vdash \sigma : p$	a well-formed channel type (mode p) in Γ

$\mathtt{tp}$ stands for either $\mathtt{tp}_H$ or $\mathtt{tp}_P$ and $\mathtt{J}$ stands for any judgement. For simplicity, we omit subtyping rules from [52], but (Weak) in the main typing rule in Figure 3 essentially plays the same rôle as process subsumption. As in [52], the first judgement $\Gamma \vdash \mathtt{Env}$ is designed to ensure that an identifier can only be used in the construction of a type if it has already been *declared* in the environment. The formation rules for functional types are the same as [52]; (t-base) is for constant and (t-abs$_H$) is for higher-order values and (t-abs$_N$) for the functional name dependent type. See [52, § 3.1] for more explanation.

The formulation rules for process types are the same as in (t-proc) in [52]. The second condition $\Gamma \vdash u : \sigma$ (which means either $\Gamma(u) = \langle\sigma,\sigma'\rangle$ or $\Gamma(u) = \sigma$ together with (Chan,Pair) in Figure 3) ensures that a process always has a channel environment which does not exceed the capability specified by the global environment Γ. In channel types, $x_1,\ldots,x_n$ bind τ_j and $x_1,\ldots,x_{i-1}$ bind σ_i from the formation rule. A pair type consists of input and its dual output.

A.3 Proofs for § 2.3

Theorem 2.4. First, as in [52, Lemma 3.1], we prove the same basic consistency lemmas such as renaming closure. We list only different lemmas from [52]. The proof follows the same routine in [52].

LEMMA A.1. *1.* (bounded weakening) *Assume* $\Gamma, u{:}\sigma, \Gamma' \vdash \mathtt{J}$. *Then* $\Gamma, u{:}\langle\sigma,\overline{\sigma}\rangle, \Gamma' \vdash \mathtt{J}$.

2. (narrowing) *Assume* $\Gamma, u{:}\sigma, \Gamma' \vdash \mathtt{J}$ *with* $u \notin \mathsf{fnv}(\Gamma') \cup \mathsf{fnv}(\mathtt{J})$. *Then* $\Gamma, \Gamma' \vdash \mathtt{J}$.

3. (name substitution) *Suppose* $\Gamma \vdash u : \sigma$. *Then* $\Gamma, x{:}\sigma, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma, \Gamma'\{u/x\} \vdash \mathtt{J}\{u/x\}$.

(Well-Formed Environment)

$$\text{(e-nil)}\quad \emptyset \vdash \mathtt{Env} \qquad \text{(e-val)}\quad \frac{\Gamma \vdash \tau : \mathtt{tp}_H \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x{:}\tau \vdash \mathtt{Env}} \qquad \text{(e-chan)}\quad \frac{\Gamma \vdash \sigma : p \quad u \notin \mathsf{dom}(\Gamma)}{\Gamma, u{:}\sigma \vdash \mathtt{Env}}$$

(Well-Formed Types)

$$\text{(t-base)}\ \frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash \mathtt{unit} : \mathtt{tp}_H} \qquad \text{(t-abs}_H\text{)}\ \frac{\Gamma \vdash \tau_1 : \mathtt{tp}_H \quad \Gamma \vdash \tau_2 : \mathtt{tp}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathtt{tp}_H} \qquad \text{(t-abs}_N\text{)}\ \frac{\Gamma, x{:}\sigma \vdash \tau : \mathtt{tp}}{\Gamma \vdash \Pi(x{:}\sigma)\tau : \mathtt{tp}_H}$$

$$\text{(t-emp)}\ \frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash \emptyset : \mathtt{tp}_P} \qquad \text{(t-pro)}\ \frac{\Gamma \vdash \Delta : \mathtt{tp}_P \quad \Gamma \vdash u : \sigma \quad u \notin \mathsf{dom}(\Delta)}{\Gamma \vdash \Delta, u{:}\sigma : \mathtt{tp}_P} \qquad \text{(t-input)}\ \frac{\Gamma, x_1{:}\sigma_1, .., x_n{:}\sigma_n \vdash \tau_j : \mathtt{tp}_H}{\Gamma \vdash (\Pi[x_1{:}\sigma_1, ..., x_n{:}\sigma_n]\tau_1, ..., \tau_m)^{\mathtt{I}} : \mathtt{I}}$$

$$\text{(t-dual)}\ \frac{\Gamma \vdash \sigma : \mathtt{I}}{\Gamma \vdash \overline{\sigma} : \mathtt{O}} \qquad \text{(t-pair)}\ \frac{\Gamma \vdash \sigma : \mathtt{O}}{\Gamma \vdash \langle \sigma, \overline{\sigma} \rangle : \mathtt{IO}}$$

Figure 10: Well-Formed Higher-Order IO Types

(Types)

$$\mathtt{unit}\{V/x\} = \mathtt{unit} \qquad \mathtt{nat}\{V/x\} = \mathtt{nat}$$
$$(\tau \to \tau')\{V/x\} = \tau\{V/x\} \to \tau'\{V/x\}$$
$$(\Pi(y{:}\sigma)\tau)\{V/x\} = \Pi(y{:}\sigma\{V/x\})\tau\{V/x\} \quad (x \neq y)$$
$$\{u_1{:}\sigma_1, ..., u_n{:}\sigma_n\}\{V/x\} = \{u_1\{V/x\}{:}\sigma_1\{V/x\}\} \odot \cdots \odot \{u_n\{V/x\}{:}\sigma_n\{V/x\}\}$$
$$(\Pi[x_1{:}\sigma_1, ..., x_n{:}\sigma_n]\tau_1, ..., \tau_m)^p\{V/x\} = (\Pi[x_1{:}\sigma_1\{V/x\}, ..., x_n{:}\sigma_n\{V/x\}]\tau_1\{V/x\}, ..., \tau_m\{V/x\})^p \quad (x \neq x_i)$$
$$\langle \sigma, \sigma' \rangle\{V/x\} = \langle \sigma\{V/x\}, \sigma'\{V/x\} \rangle$$

(Terms)

$$(u(x_1{:}\tau_1, \ldots x_n{:}\tau_n).P)\{V/x\} = u\{V/x\}?(x_1 : \tau_1\{V/x\}, \ldots x_n : \tau_n\{V/x\}).P\{V/x\} \quad (x \neq x_i)$$
$$\cdots$$

Figure 9: Substitution into Types and Terms

4. (process types) $\Gamma \vdash \Delta_{1,2} : \mathtt{tp}_P$ *implies* $\Gamma \vdash \Delta_1 \odot \Delta_2 : \mathtt{tp}_P$.

LEMMA A.2. 1. (narrowing) *Suppose* $\Gamma, u{:}\sigma, \Gamma' \vdash P : \tau$ *with* $u \notin \mathsf{fnv}(\Gamma', P)$. *Then there exists* τ' *such that* $u \notin \mathsf{fnv}(\tau')$ *and* $\Gamma, \Gamma' \vdash P : \tau'$.
2. (name occurrence) *Suppose* $\Gamma, a{:}\sigma \vdash P : \Delta, a{:}\sigma$. *Then* $a \notin \mathsf{fn}(\Delta)$.
3. (substitution) *Suppose* $\Gamma \vdash V : \tau$. *Then* $\Gamma, x{:}\tau, \Gamma' \vdash P : \tau'$ *implies* $\Gamma, \Gamma'\{V/x\} \vdash P\{V/x\} : \tau'\{V/x\}$.

Using the above Narrowing and the standard Weakening, we have:

PROPOSITION A.3. (struct) *Suppose* $\Gamma \vdash P : \tau$ *and* $P \equiv P'$. *Then* $\Gamma \vdash P' : \tau$.

Then by this proposition and the above substitution lemma, we conclude Theorem 2.4.

Proof for Proposition 2.5. The proof is by constructing the minimum typing system where we apply weakening rules just before (Res) and (In) as much as we need to obtain the latter. See [51, 17] for the similar proofs to construct the minimum typing. (2) is obvious by (Weak) together with (1). Note that Corollary 2.6 (1) and (2) are straightforward from the minimum interface property (1) and the maximum one (2), respectively.

A.4 Existential Typing

Typing. First we extend the syntax of types in Figure 2 as follows.

$$\tau \ ::= \ \ldots \mid \exists[x{:}\sigma]\tau$$

$\exists[x{:}\sigma]\tau$ is an existential type with name x and value type τ (its binding/free names/variables are defined as $\Pi[x{:}\sigma]\tau$). We also use $\{v, V\}$, a pair of name v and value V, which appears only during type inference.[7] The extended rules for well-formedness and typing are listed in Figure 11. In the environment, $\{x{:}\sigma, y{:}\tau\}$ guarantees that x only appears in τ, i.e. its scope is in τ. In the typing rule, (Tupl$\exists$)/(Val$\exists$) is an introduction of a pair of variables/a variable. (Pack) is a packing construction rule. (In$\exists$) and (Out$\exists$) can be regarded as a special case of (In) and (Out) of the dependent typing system in Figure 3 where scope of x is restricted in τ.

We list typing for an example in the main section as follows.

EXAMPLE A.4. (existential typing) We will type the term

$$c(y).(y().\overline{a}\langle y \rangle) \mid (\nu v)(\overline{c}\langle \ulcorner \overline{v}\langle 2 \rangle \urcorner \rangle \mid !v(z).R)$$

in (10). First let $\sigma_x = (\mathtt{nat})^{\mathtt{O}}$, $\tau = \ulcorner x{:}\sigma_x \urcorner$, $\sigma_a = (\exists[x{:}\overline{\sigma_x}]\ulcorner x{:}\sigma_x \urcorner)^{\mathtt{O}}$, and $\sigma_c = \langle \sigma_a, \overline{\sigma_a} \rangle$. Then by (Var$\exists$) and (App^H), we have:

$$\Gamma, \{x{:}\sigma_x, y{:}\tau\} \vdash y() : x{:}\sigma_x$$

with $\Gamma = c{:}\sigma_c, a{:}\sigma_a$. Then by (Tupl$\exists$) and (Out$\exists$), we have:

$$\Gamma, \{x{:}\sigma_x, y{:}\tau\} \vdash \overline{a}\langle y \rangle : \{a{:}\sigma_a, x{:}\sigma_x\}$$

Since a name in σ_a is existential quantified, we know $x \notin \mathsf{fv}(\sigma_a)$. Hence after (Par), we can apply (In$\exists$) to obtain:

$$\Gamma \vdash c(y{:}\exists[x{:}\sigma_x]\tau).(y().\overline{a}\langle y \rangle) \ : \ \{c{:}\overline{\sigma_a}, a{:}\sigma_a\}$$

Similarly, by (Pack), we have $\{v, \ulcorner \overline{v}\langle 2 \rangle \urcorner\} : \exists[x{:}\sigma]\tau$. Hence we have:

$$\Gamma \vdash (\nu v)(\overline{c}\langle \ulcorner \overline{v}\langle 2 \rangle \urcorner \rangle \mid !v(z).R) : c{:}\sigma_a$$

Thus P has a type $\{c{:}\langle \sigma_a, \overline{\sigma_a} \rangle, a{:}\sigma_a\}$, as required.

Properties with Existential Types. We first list basic lemmas for existential types which are different from the channel dependency types. Note in (5), existential name v is only substituted to type τ, not to term P (cf. Lemma A.2 (3)).

[7] For simplicity, in the existential typing system here, we only consider a single name and a single type, which can be easily generalised to the arbitrary number of names and/or types as $\exists[\vec{x}{:}\vec{\sigma}]\vec{\tau}$. We also only consider value V in $\{v, V\}$; it can be also generalised to "$\{v, P\}$ `as` $\exists[x{:}\sigma]\tau$" and `let` $\{y, z\} = P$ `in` P by adding the reduction rules for tuples and `let` as defined in the standard existential typing system [34].

(Well-Formed Environment and Types)

$$\text{(t-exist)}\ \frac{\Gamma, x{:}\sigma \vdash \tau : \mathtt{tp}_H}{\Gamma \vdash \exists[x{:}\sigma]\tau : \mathtt{tp}_\exists} \qquad \text{(t-pack)}\ \frac{\Gamma \vdash \exists[x{:}\sigma]\tau : \mathtt{tp}_\exists \quad x,y \notin \mathsf{dom}(\Gamma)}{\Gamma, \{x{:}\sigma, y{:}\tau\} \vdash \mathtt{Env}} \qquad \text{(t-in}\exists\text{)}\ \frac{\Gamma \vdash \tau : \mathtt{tp}_\exists}{\Gamma \vdash (\tau)^{\mathtt{I}} : \mathtt{I}}$$

(Typing System)

$$\text{(Tpl}\exists\text{)}\ \frac{\Gamma, \{x{:}\sigma, y{:}\tau\}, \Gamma' \vdash \mathtt{Env}}{\Gamma, \{x{:}\sigma, y{:}\tau\}, \Gamma' \vdash \{x,y\} : \exists[x{:}\sigma]\tau} \qquad \text{(Pack)}\ \frac{\Gamma \vdash v : \sigma \quad \Gamma \vdash V : \tau\{v/x\}}{\Gamma \vdash \{v, V\} : \exists[x{:}\sigma]\tau} \qquad \text{(Var}\exists\text{)}\ \frac{\Gamma, \{x{:}\sigma, y{:}\tau\}, \Gamma' \vdash \mathtt{Env}}{\Gamma, \{x{:}\sigma, y{:}\tau\}, \Gamma' \vdash y : \tau}$$

$$\text{(In}\exists\text{)}\ \frac{\Gamma \vdash u : (\exists[x{:}\sigma]\tau)^{\mathtt{I}} \quad \Gamma, \{x{:}\sigma, y{:}\tau\} \vdash P :\ \Delta, x{:}\sigma}{\Gamma \vdash u(y{:}\exists[x{:}\sigma]\tau).P :\ u{:}(\exists[x{:}\sigma]\tau)^{\mathtt{I}}, \Delta} \qquad \text{(Out}\exists\text{)}\ \frac{\Gamma \vdash u : (\exists[x{:}\overline{\sigma}]\tau)^{\mathtt{O}} \quad \Gamma \vdash \{v, V\} : \exists[x{:}\sigma]\tau}{\Gamma \vdash \overline{u}\langle V\rangle\ :\ u{:}(\exists[x{:}\overline{\sigma}]\tau)^{\mathtt{O}}, v{:}\sigma}$$

Other rules are from Figure 3.

Figure 11: Existential Typing System

LEMMA A.5. *1. Suppose* $\Gamma, \{x{:}\sigma, y{:}\tau\}, \Gamma' \vdash \mathtt{J}$. *Then* $x \notin \mathsf{fv}(\Gamma, \Gamma', \mathtt{J})$.

2. *Suppose* $\Gamma, \{x{:}\sigma, y{:}\tau\}, \Gamma' \vdash P : \tau'$. *Then* $x \notin \mathsf{fv}(\Gamma, \Gamma', P)$.
3. (renaming for a package) *Suppose* $\Gamma, \{x_1{:}\sigma, y_1{:}\tau\}, \Gamma', \{x_2{:}\sigma, y_2{:}\tau\{x_2/x_1\}\}, \Gamma'' \vdash \{x_2, y_2\} : \exists[x_1{:}\sigma]\tau$. *Then we have* $\Gamma, \{x_1{:}\sigma, y_1{:}\tau\}, \Gamma', \Gamma'' \vdash \{x_1, y_1\} : \exists[x_1{:}\sigma]\tau$.
4. (renaming) *Suppose* $\Gamma, \{x_1{:}\sigma, y_1{:}\tau\}, \Gamma', \{x_2{:}\sigma, y_2{:}\tau\{x_2/x_1\}\}, \Gamma'' \vdash P : \tau'$. *Then* $\Gamma, \{x_1{:}\sigma, y_1{:}\tau\}, \Gamma', \Gamma'' \vdash P\{y_1/y_2\}\{x_1/x_2\} : \tau'\{x_1/x_2\}$.
5. (substitution lemma for a package) *Suppose* $\Gamma \vdash \{w, W\} : \exists[x{:}\sigma]\tau$ *and* $\Gamma, \{x{:}\sigma, y{:}\tau\}, \Gamma' \vdash \{v, V\} : \exists[z{:}\sigma']\tau'$. *Then* $\Gamma, \Gamma' \vdash \{v\{w/x\}, V\{W/y\}\} : \exists[z{:}\sigma'\{w/x\}]\tau'\{w/x\}$.
6. (substitution lemma) *Suppose* $\Gamma, \{x{:}\sigma, y{:}\tau\}, \Gamma' \vdash P : \tau'$ *and* $\Gamma \vdash \{v, V\} : \exists[x{:}\sigma]\tau$. *Then* $\Gamma, \Gamma' \vdash P\{V/y\} : \tau'\{v/x\}$.

PROOF. (1) and (2) are straightforward from the rules. (3–6) are proved simultaneously. All are easy by regarding (5) as a special case of Substitution Lemma of the dependent types, where $x \notin \mathsf{fv}(P)$. □

THEOREM A.6. (subject reduction) $\Gamma \vdash P : \tau$ *and* $P \twoheadrightarrow P'$ *imply* $\Gamma \vdash P' : \tau$.

PROOF. By the same routine as Subject Reduction Theorem of the dependency types using the above lemma. □

The same basic propositions in the previous section (Propositions 2.5, 2.6,2.7) hold for existential types. In the following, we prove the type erasure theorem. The syntax and operational semantics of untyped λπ-calculus are defined as in Figure 7 erasing type annotations in bound variables and names.

DEFINITION A.7. *The* erasure *of an IO-typed term is defined as follows.*

$$\mathsf{Erase}(\mathbf{0}) = \mathbf{0} \quad \mathsf{Erase}(P \,|\, Q) = \mathsf{Erase}(P) \,|\, \mathsf{Erase}(Q)$$
$$\mathsf{Erase}((\nu\, a{:}\sigma)P) = (\nu\, a)\mathsf{Erase}(P)$$
$$\mathsf{Erase}(\overline{u}\langle V_1, ..., V_n\rangle) = \overline{u}\langle \mathsf{Erase}(V_1), ..., \mathsf{Erase}(V_n)\rangle$$
$$\mathsf{Erase}(u(x_1{:}\tau_1, ..., x_n{:}\tau_n).P) = u(x_1, ..., x_n).\mathsf{Erase}(P)$$
$$\mathsf{Erase}(!\,u(x_1{:}\tau_1, ..., x_n{:}\tau_n).P) = !\,u(x_1, ..., x_n).\mathsf{Erase}(P)$$
$$\mathsf{Erase}(\lambda(x{:}\tau).P) = \lambda x.\mathsf{Erase}(P)$$
$$\mathsf{Erase}(P\,Q) = \mathsf{Erase}(P)\mathsf{Erase}(Q) \quad \mathsf{Erase}(u) = u \quad \mathsf{Erase}(()) = ()$$

Note that the existential bound names are erased whereas the dependent ones are not. If u has a type $(\exists[x{:}\sigma]\tau)^{\mathtt{I}}$, then $\mathsf{Erase}(u(y : \exists[x : \sigma]\tau).P) = u(y).\mathsf{Erase}(P)$, while if u has a type $(\Pi[x{:}\sigma]\tau)^{\mathtt{I}}$, then $\mathsf{Erase}(u(x : \sigma, y : \tau).P) = u(x,y).\mathsf{Erase}(P)$.

PROPOSITION A.8. *1. If* $P \equiv P'$, *then* $\mathsf{Erase}(P) \equiv \mathsf{Erase}(P')$.

2. *Suppose* $\Gamma \vdash P : \Delta$ *and* $P \longrightarrow P'$. *Then* $\mathsf{Erase}(P) \longrightarrow \mathsf{Erase}(P')$.
3. *Suppose* $\mathsf{Erase}(P) \longrightarrow Q'$. *Then there exists a typed term* P' *such that* $P \longrightarrow P'$ *and* $\mathsf{Erase}(P') = Q'$.

PROOF. By induction of $\equiv$ and $\longrightarrow$.

Example: Forwarding and Filtering. To demonstrate the power of dependent and existential channel types, we type remote-forwarding and filtering, which are similar to those formalised by combining dynamic and static type-checking in [37]. Our fine-grained types can provide a powerful and flexible host-based protection mechanism, retaining a static checking framework.

Suppose thunked process $\ulcorner P \urcorner$ is passed to remote site l. This can be simply written as $\overline{l}\langle \ulcorner P \urcorner \rangle$ in λπ, which corresponds to `goto` $l.P$ using the notation in [37]. Using our process types, the receiver site can restrict the usage of migrating code as:

$$!\,l(x{:}\ulcorner \Delta_l \urcorner).\,run\,y \;\mid\; !a(x).R_a \,|\, !b(x).R_b \,|\, ...$$

Here $!\,l(x{:}\ulcorner\Delta_l\urcorner).\,run\,x$ behaves as a filter of location l against which incoming code is compared. We write this filter as $l\langle\!\langle \Delta_l \rangle\!\rangle$ following the notation in [37]. If $\Delta = \{a{:}(\mathtt{nat})^{\mathtt{O}}\}$, accepted code can access only resource $!a(x).R_a$ but has no permission to access $!b(x).R_b$.

Our fine-grained types can provide arbitrary higher-order behaviour by functional arrow types, hence we can provide a nested specification. For example, the following agent

$$\mathtt{goto}\, k.(\mathtt{goto}\, l.\overline{a}\langle v\rangle \,|\, \overline{b}\langle v\rangle)$$

is translated into $\overline{k}\langle \ulcorner \overline{l}\langle \ulcorner \overline{a}\langle v\rangle \urcorner\rangle \,|\, \overline{b}\langle v\rangle \urcorner\rangle$. Then a filter of site k, $k\langle\!\langle \Delta_k \rangle\!\rangle$, has a nested type Δ_k as:

$$\{l{:}(\ulcorner a{:}(\mathtt{nat})^{\mathtt{O}}\urcorner)^{\mathtt{O}},\ b{:}(\mathtt{nat})^{\mathtt{O}}\}$$

Then incoming code via k can only access the resource b, then moves to the site l then access only resource a.

The next example shows our channel dependency type can represent the similar mechanism ensured by the location dependency type in [37]. Let us consider the following simple forwarder

$$b(x,y).\mathtt{goto}\ y.\overline{x}\langle 1\rangle$$

which inputs two names x and y and goes to the location y at which x is located. Then b has a channel dependent type as follows:

$$(\Pi[x:(\mathtt{nat})^{\mathtt{O}}](\ulcorner x:(\mathtt{nat})^{\mathtt{O}\urcorner})^{\mathtt{O}})^{\mathtt{I}}$$

Here $\ulcorner x:(\mathtt{nat})^{\mathtt{O}\urcorner}$ is a type of a filter of location y, in which type of x is declared.

Next let us consider another forwarder

$$P \stackrel{\text{def}}{=} \mathtt{goto}\ k.\ b(x).\mathtt{goto}\ l.\overline{a}\langle x\rangle$$

which inputs a value from port b at location k and forwards it to port a at l. Now we consider the following example:

$$(\nu a)(\mathtt{goto}\ k.\ b(x).\mathtt{goto}\ l.\overline{a}\langle x\rangle \mid l\langle\!\langle \Delta_l \rangle\!\rangle \mid a(x).R_a)$$

The scope restriction ensures that a message at a can always return to local resource a at l via filter $l\langle\!\langle \Delta_l \rangle\!\rangle$. P is translated into $\overline{k}\langle \urcorner !b(x).\overline{l}\langle \ulcorner \overline{a}\langle x\rangle^{\urcorner}\rangle^{\urcorner}\rangle$ in $\lambda\pi$. Then the channel type of k of the filter $k\langle\!\langle \Delta_k \rangle\!\rangle$ can be declared by using existential quantifier over hidden name a:

$$(\ulcorner b:(\mathtt{nat})^{\mathtt{I}},\ l:(\exists[y:(\mathtt{nat})^{\mathtt{I}}]\ulcorner y:(\mathtt{nat})^{\mathtt{O}\urcorner})^{\mathtt{O}\urcorner})^{\mathtt{I}}$$

This type means that location k can accept a process which interacts port b and location l which contains *anonymous* resource which carries `nat` type. Another example which is not statically typable in the system of [37] is:

$$\mathtt{goto}\ k.\ !b(x).\mathtt{goto}\ l.\overline{x}\langle 1\rangle$$

To accept this agent, a type of k should be able to declares:

$$(\ulcorner b:((\mathtt{nat})^{\mathtt{O}})^{\mathtt{I}},\ l:(\exists[x:(\mathtt{nat})^{\mathtt{I}}]\ulcorner x:(\mathtt{nat})^{\mathtt{O}\urcorner})^{\mathtt{O}\urcorner})^{\mathtt{I}}$$

Note l has a type which contains an output of `nat` type with *anonymous* name x. [15] also shows functional and channel dependent types as well as existential types can provide a static host protection mechanism in a non-trivial higher-order extension of Dπ.

B. INTEGRATION WITH LINEAR/AFFINE TYPES

This section defines the types and typing systems for $\lambda\pi^{\mathsf{A}}$ in § B.1, $\lambda\pi^{\mathsf{L}}$ in § B.2, $\lambda\pi^{\mathsf{LA}}$ in § B.3 and $\lambda\pi^{\mathsf{LAM}}_{<}$ in § B.4, respectively, demonstrating the fine-grained-typing system makes it possible to integrate different type structures essentially (1) by refining the algebraic laws of compositions of modes and channels, i.e. $\odot$, and (2) by replacing the typing systems for processes. Since the natural numbers can be encoded into branching and selection introduced in $\lambda\pi^{\mathsf{LAM}}_{<}$ [5, 20, 50, 53], hereafter we only consider `unit` as a base type.

B.1 Affine Type Discipline

The affine typing system is defined by integrating π^{A} introduced in [5] into which PCF/PCFv are fully abstractly embeddable. With this refinement, we can guarantee availability of server resources and linear usage of channels, together with locality [51, 39]. We call this calculus the *affine* $\lambda\pi$*-calculus*, denoted by $\lambda\pi^{\mathsf{A}}$.

Types. The syntax of terms and the types remains as the same as Figures 1 and 2 except the modes p in channel types. We first extend the set of modes as follows (we omit script A from modes until we introduce the mixed calculus, $\lambda\pi^{\mathsf{LA}}$ in § B.3).

$\downarrow$	Affine input	$\uparrow$	Affine output
!	Unique Server	?	Client requests to !

$$\text{(t-abs}_N\text{)}\ \frac{\Gamma, x:\sigma_{\mathtt{O}} \vdash \tau:\mathtt{tp}}{\Gamma \vdash \Pi(x:\sigma_{\mathtt{O}})\tau:\mathtt{tp}_H} \qquad \text{(t-pros)}\ \frac{\Gamma \vdash ?\Delta:\mathtt{tp}_P \quad \Gamma \vdash u:\sigma_{\mathtt{O}} \quad u \notin \mathsf{dom}(\Delta)}{\Gamma \vdash \Delta, u:\sigma_{\mathtt{O}}:\mathtt{tp}_{PS}}$$

$$\text{(t-}\downarrow\text{)}\ \frac{\mathsf{md}(\sigma_i) = ? \quad \Gamma, x_1:\sigma_1, .., x_n:\sigma_n \vdash \tau_j:\mathtt{tp}_H}{\Gamma \vdash (\Pi[x_1:\sigma_1, ..., x_n:\sigma_n]\tau_1, ..., \tau_m)^{\downarrow}:\downarrow}$$

$$\text{(t-!)}\ \frac{\mathsf{md}(\sigma_i) = ?\ (i \neq n) \quad \mathsf{md}(\sigma_n) = \uparrow \quad \Gamma, x_1:\sigma_1, .., x_n:\sigma_n \vdash \tau_j:\mathtt{tp}_H}{\Gamma \vdash (\Pi[x_1:\sigma_1, ..., x_n:\sigma_n]\tau_1, ..., \tau_m)^{!}:!}$$

Others are from Figure 10 (in (t-dual,pair), we replace `I` by $p_{\mathtt{I}}$, `O` by $p_{\mathtt{O}}$ and `IO` by $p_{\mathtt{I}} \odot p_{\mathtt{O}}$; `tp` stands for either $\mathtt{tp}_H$ or $\mathtt{tp}_{PS}$).

Figure 12: Well-Formed Affine Higher-Order Types

as well as $\updownarrow$, which indicates uncomposability at affine channels. $\downarrow, !$ are input modes, denoted by p_{I}, while $\uparrow, ?$ are output modes, denoted by p_{O}. As a small example, a channel which outputs a natural number at most once has type $(\mathtt{nat})^{\uparrow}$ and $(\mathtt{nat})^{\downarrow}$ is its dual. On the other hand, a channel which can repeatedly output natural numbers has type $(\mathtt{nat})^{?}$, and its dual $(\mathtt{nat})^{!}$ denotes the dual unique server. We often denote $\sigma_{\mathtt{I}}$ for a type σ s.t. $\mathsf{md}(\sigma) = p_{\mathrm{I}}$, similarly for $\sigma_{\mathtt{O}}$ and $\sigma_{\mathtt{IO}}$.

Next the rules (m-1) and (m-2) in § A.1 are refined to:

(a-m-1) $\overline{\downarrow} = \uparrow$, $\overline{!} = ?$ and $\overline{\overline{p}} = p$.

(a-m-2) $\odot$ is generated from: (1) $\downarrow \odot \uparrow = \updownarrow$, (2) $! \odot ? = !$ and (3) $? \odot ? = ?$.

Intuitively (1) in the above definition of $\odot$ says an affine input can only be composed with an affine output, with no further connection, while (2-3) say a server can be composed with zero or more clients. The definition of duality of channels is given as before. Then we define $\odot$ on types such that:

(a-t-1) as the same as **(t-1)** by replacing `O` by p_{O}, and

(a-t-2) the same rule **(t-2)** is applied to σ if $\mathsf{md}(\sigma) = ?$.

Note other compositions of types are undefined. The definition of the operator $\Delta_1 \odot \Delta_2$ remains as the same.

The new judgements and typing rules are listed in Figures 12 and 13. Here $\vec{p}\Delta$ indicates $\mathsf{md}(\Delta) \subseteq \{\vec{p}\}$ and Δ^{-u} means $u \notin \mathsf{fnv}(\Delta)$. For well-formedness, we add the following three main constraints;

Locality channels/values without input capability are instantiated and sent around via channels [39, 51] (guaranteed by $\sigma_{\mathtt{O}}$ in (t-abs$_N$) and (t-proc)).

IO-alternation input carries only output and vice verser (guaranteed by (t-!) and (t-$\uparrow$)); and

Unique answer per thread only a replicated input channel carries unique affine output and vise verser (guaranteed by (t-!)) and each input prefix/λ-abstraction contains at most one affine output (guaranteed by $?\Delta$ in (t-proc))

The first constraint is now standard in the literature for a natural representation of distributed computation [39, 43, 51]. The last two conditions, which were inherited from [5], had come form game semantics [4, 19, 21]. The resulting calculus is an integration of the local higher-order π-calculus [39, 51] and the affine π-calculus [5]. Since each original calculus can embed the λ-calculus, it does not lose expressiveness.

(Processes)

$$\frac{\Gamma \vdash P_{1,2} : \Delta_{1,2} \quad \Delta_1 \asymp \Delta_2}{\Gamma \vdash P_1 \,|\, P_2 : \Delta_1 \odot \Delta_2}\ (\text{Par}) \qquad \frac{\Gamma, a:\sigma \vdash P : \Delta, a:\sigma \quad \mathsf{md}(\sigma) \in \{\updownarrow, !\}}{\Gamma \vdash (\nu a{:}\sigma)P : \Delta}\ (\text{Res}) \qquad \frac{\Gamma \vdash a:\sigma \quad \mathsf{md}(\sigma) \in \{\updownarrow, ?\} \quad \Gamma \vdash P : \Delta \asymp a:\sigma}{\Gamma \vdash P : \Delta \odot a:\sigma}\ (\text{Weak})$$

$$(\mathsf{In}^{\downarrow})\ \frac{\Gamma \vdash u : (\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^{\downarrow} \quad \Gamma, \vec{x}:\vec{\sigma}, \vec{y}:\vec{\tau} \vdash P :\ \vec{x}:\vec{\sigma}, \uparrow ?\Delta^{-u}}{\Gamma \vdash u(\vec{x}:\vec{\sigma}, \vec{y}:\vec{\tau}).P :\ u : (\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^{\downarrow}, \Delta}$$

$$(\mathsf{In}^{!})\ \frac{\Gamma \vdash u : (\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^{!} \quad \Gamma, \vec{x}:\vec{\sigma}, \vec{y}:\vec{\tau} \vdash P :\ \vec{x}:\vec{\sigma}, ?\Delta^{-u}}{\Gamma \vdash\ !u(\vec{x}:\vec{\sigma}, \vec{y}:\vec{\tau}).P :\ u : (\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^{!}, \Delta}$$

Others are from Figure 3 (in (Out), we replace $\mathtt{0}$ by $p_{\mathtt{0}}$)

Figure 13: Affine Higher-Order Typing System

Typing. The typing system for the affine $\lambda\pi$ is defined in Figure 13. We just need to replace **(Process)** in IO-rules in Figure 3 to those of the π^{A}-calculus [5]. In (Par), we need to check the compositionality of P_1 and P_2 by checking $\Delta_1 \asymp \Delta_2$ by using the laws **(a-t-1,2)**. Note that the definition of $\asymp$ allows the parallel composed processes can have many free different affine outputs. (Weak) can weaken only $\updownarrow$ and ?-nodes since they allow the possibility of no action. In (Res), we do not allow $\uparrow$, ? or $\downarrow$-channel to be restricted since these actions expect their dual actions exist in the environment. ($\mathsf{In}^{\downarrow}$) ensures that x occurs precisely once (by Δ^{-u}) and that no free input is suppressed under the prefix (by $\uparrow ?\Delta$). ($\mathsf{In}^{!}$) is the same as ($\mathsf{In}^{\downarrow}$) except that no free $\uparrow$-channels are suppressed (if a $\uparrow$-channel is under replication, then it can be used more than once). Note that the common and functional rules remain as the same.

We list the key name substitution lemma. Note that Lemma A.1 (4) no longer holds for the affine $\lambda\pi$. We prove (1,2,3) simultaneously.

LEMMA B.1. *1.* (?-name substitution) *Suppose* $\Gamma \vdash u{:}\sigma$ *with* $\mathsf{md}(\sigma) = ?$. *Then* $\Gamma, x:\sigma, \Gamma' \vdash P : \tau$ *implies* $\Gamma, \Gamma'\{u/x\} \vdash P\{u/x\} : \tau\{u/x\}$.

2. ($\uparrow$-name substitution into process types) *Suppose* $\Gamma \vdash u{:}\sigma$ *with* $\mathsf{md}(\sigma) = \uparrow$, $\Gamma, x:\sigma, \Gamma' \vdash P : \Delta, x:\sigma$ *and* $u \notin \mathsf{dom}(\Delta)$. *Then* $\Gamma, \Gamma'\{u/x\} \vdash P\{u/x\} : \Delta\{u/x\}, u:\sigma$.

3. ($\uparrow$-name substitution into well-formed types) *Suppose* $\Gamma \vdash u{:}\sigma$ *and* $\Gamma, x:\sigma, \Gamma' \vdash P : \tau$ *with* $\Gamma \vdash \tau : \mathtt{tp}$. *Then* $\Gamma, \Gamma'\{u/x\} \vdash P\{u/x\} : \tau\{u/x\}$.

PROOF. (1-3) are proved simultaneously. (1) is proved just as Lemma A.1 (3). For (2) and (3), we use the substitution lemma for well-formed channel and value types. □

By the above substitution lemma, one can prove the subject reduction theorem. All properties listed in Propositions 2.5, 2.6 and 2.7, as well as basic properties in [5] hold in this calculus. We can show the following more refined properties in $\lambda\pi^{\mathsf{A}}$.

PROPOSITION B.2. (resource availability) *Suppose* $\Gamma \vdash P : \Delta$. *If* $a \in \mathsf{dom}(\Delta)$ *with* $\mathsf{md}(\Delta(a)) \in \{\downarrow, !\}$, *then* $P \Downarrow a^{\mathtt{I}}$.

Affine Existential Type. The existential type of the affine type discipline is defined by regarding $(\exists[x:\sigma]\tau)^{p}$ as a special case of $(\Pi[x:\sigma]\tau)^{p}$. For example, $(\exists[x:\sigma]\tau)^{\downarrow}$ is well typed if $\mathsf{md}(\sigma) = ?$.

$$(\mathsf{In}^{\downarrow})\ \frac{\Gamma \vdash u : (\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^{\downarrow} \quad \Gamma, \vec{x}:\vec{\sigma}, \vec{y}:\vec{\tau} \vdash P :\ \vec{x}:\vec{\sigma}, \uparrow \Delta^{-u}, ?\Delta'}{\Gamma \vdash u(\vec{x}:\vec{\sigma}, \vec{y}:\vec{\tau}).P : u : (\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^{\downarrow} \to \Delta, \Delta'}$$

$$(\mathsf{In}^{!})\ \frac{\Gamma \vdash u : (\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^{!} \quad \Gamma, \vec{x}:\vec{\sigma}, \vec{y}:\vec{\tau} \vdash P :\ \vec{x}:\vec{\sigma}, ?\Delta^{-u}}{\Gamma \vdash\ !u(\vec{x}:\vec{\sigma}, \vec{y}:\vec{\tau}).P :\ u : (\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^{!} \to \Delta}$$

Well-formed types are defined as in Figure 12. Others are from Figure 13.

Figure 14: Linear Higher-Order Typing System

We also need to add one following condition to the assumption of (t-exist) and (Pack).

$$\text{if } \mathsf{md}(\sigma) = \uparrow \text{ then } \mathsf{tail}(\tau) \vdash x{:}\sigma$$

where $\mathsf{tail}(\ )$ is defined as: $\mathsf{tail}(\mathtt{unit}) = \emptyset$, $\mathsf{tail}(\tau \to \tau') = \mathsf{tail}(\tau')$, $\mathsf{tail}(\Pi(x:\sigma)\tau) = \mathsf{tail}(\tau)/x$ and $\mathsf{tail}(\Delta) = \Delta$. This condition means if the linear name is existentially quantifiered, then it should appear in the final process type. If this condition is dropped, the typing system is not closed under the scope opening rule since the existential name is erased. To see this, let us consider the following pack where $a \notin \mathsf{fnv}(V)$.

$$\{a, \ulcorner \overline{b} \urcorner\} : \exists[x:()^{\uparrow}]\ulcorner b:()^{\uparrow}\urcorner$$

Then the following process is typable since $a:()^{\uparrow}$ has the opponent process with dual type $a:()^{\downarrow}$.

$$P \;=\; (\nu a{:}\sigma)(\overline{u}\langle \ulcorner \overline{b} \urcorner \rangle \,|\, a.\mathbf{0})$$

where $\sigma = \langle ()^{\downarrow}, ()^{\uparrow} \rangle$. However $P \equiv \overline{u}\langle \ulcorner \overline{b} \urcorner \rangle \,|\, (\nu a{:}\sigma)(a.\mathbf{0})$ is not typable. This problem does not appear in the dependency type because bound name a in a message is not erased, i.e.

$$(\nu a{:}\sigma)(\overline{u}\langle a, \ulcorner \overline{b} \urcorner \rangle \,|\, a.\mathbf{0}) \not\equiv \overline{u}\langle a, \ulcorner \overline{b} \urcorner \rangle \,|\, (\nu a{:}\sigma)(a.\mathbf{0})$$

B.2 Linear Type Discipline

The second refinement is to construct a strong normalising (SN) typing system for $\lambda\pi$ by which we can precisely embed program constructions with secrecy and access control enforcement in the main sections. We call this calculus the *linear* $\lambda\pi$*-calculus*, denoted by $\lambda\pi^{\mathsf{L}}$.

Types and Typing. The syntax of types is as the same as $\lambda\pi^{\mathsf{A}}$. Further, we write $x:\tau \to x':\tau'$ when x' causally depends on x. $x:\tau \to x':\tau'$ is allowed if either (1) $\mathsf{md}(\tau) = \downarrow$ and $\mathsf{md}(\tau) = \uparrow$ or (2) $\mathsf{md}(\tau) = !$ and $\mathsf{md}(\tau) = ?$. For example, a simple agent $x.\overline{y}$ would be typed by $x:()^{\downarrow} \to y:()^{\uparrow}$. The algebras for modes **(a-m-1/2)** and for types **(a-t-1/2)** remain as the same. Then $\Delta_1 \asymp \Delta_2$ is defined by preventing circularity between channels. For example, $x_1:\tau_1 \to x_2:\overline{\tau_2} \not\asymp x_2:\tau_2 \to x_1:\overline{\tau_1}$. For the formal definition of $\asymp$, see Section 2 in [50]. From the affine typing system, we only have to modify input rules as listed in Figure 14. In these rules, $x:\tau \to \Delta$ denoting the result of adding a new edge from $x:\tau$ to each maximal node in Δ. Note when composing P_1 and P_2 by (Par), we check circularity of names (by $\Delta_1 \asymp \Delta_2$). Other rules remain the same.

Proving SN and Linear Liveness. All basic properties for $\lambda\pi^{\mathsf{A}}$ listed in the previous subsection are proved in $\lambda\pi^{\mathsf{L}}$. In this paragraph, we prove the strong normalisation and the linear liveness property using the type preserving translation whose idea has been

$\tau^\bullet \stackrel{\text{def}}{=} (\tau^\circ)^\uparrow$ if τ is a HO-type

$(\Delta, u{:}\sigma^\uparrow)^\bullet \stackrel{\text{def}}{=} \sigma^\circ$

$\mathtt{unit}^\circ \stackrel{\text{def}}{=} ((\,)^\uparrow)!$ $\quad (\tau_1 \to \tau_2)^\circ \stackrel{\text{def}}{=} (\overline{\tau_1^\circ}\tau_2^\bullet)!$

$$(\Pi(y{:}\sigma)\tau)^\circ \stackrel{\text{def}}{=} \begin{cases} (\sigma^\circ\tau^\bullet)! & (\mathsf{md}(\sigma) = ?) \\ (\tau^\bullet)! & (\mathsf{md}(\sigma) = \uparrow) \end{cases}$$

$((\Pi[x_1{:}\sigma_1, ..., x_n{:}\sigma_n]\tau_1, ..., \tau_m)^{p_1})^\circ \stackrel{\text{def}}{=} (\sigma_1^\circ, ..., \sigma_n^\circ, \overline{\tau_1^\circ}, ..., \overline{\tau_m^\circ})^{p_1}$

$((\Pi[x_1{:}\sigma_1, ..., x_n{:}\sigma_n]\tau_1, ..., \tau_m)^{p_O})^\circ \stackrel{\text{def}}{=} (\sigma_1^\circ, ..., \sigma_n^\circ, \tau_1^\circ, ..., \tau_m^\circ)^{p_O}$

$\langle \sigma_1, \sigma_2 \rangle^\circ \stackrel{\text{def}}{=} \langle \sigma_1^\circ, \sigma_2^\circ \rangle$

$\emptyset^\circ = \emptyset$ $\quad (\Gamma, y{:}\tau)^\circ = \Gamma^\circ, y{:}\overline{\tau^\circ}$ (τ is a HO type) $\quad (\Gamma, u{:}\sigma)^\circ = \Gamma^\circ, u{:}\sigma^\circ$

Figure 15: Mapping of Linear Types

discussed in § 4.2. For simplicity, here we propose the *linear-name preserving encoding*, which is enough to prove the main results (because channels with mode ? does not affect to the environment in $\lambda\pi^{\mathsf{A}}$, $\lambda\pi^{\mathsf{L}}$ and $\lambda\pi^{\mathsf{LA}}$).

The encoding of types and terms are listed in Figures 15 and 16, respectively. The key idea here is in order to preserve the linearity, we translate process type Δ into the unique answer at $\uparrow$-type as found in the mapping $\Delta^\bullet$ in Figure 15 (σ^p means $\mathsf{md}(\sigma) = \uparrow$). The translation of the terms is defined by combining this idea and Milner's encoding of the call-by-value simply typed λ-calculus [20, Figure 7] (we omit the mapping of $\exists[x{:}\sigma]\tau$ which is defined as a special case of $\Pi[x{:}\sigma]\tau$).

For the operational correspondence, we use the extended reduction $\mapsto$ from [50] which contracts the redex under prefixes.

DEFINITION B.3. (extended reductions) We define $\mapsto$ as the compatible relations on processes modulo $\equiv$ respectively generated by the following rules.

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

Here $C[\,]$ is an arbitrary context not binding x. Then $\mapsto$ is called the *extended reduction* relation. □

The following is proved by induction on P. In (6), $\cong$ is defined as $\cong_s$ in §. 4.2 ignoring the secrecy level.

PROPOSITION B.4. *1.* (type preserving property for functions) *Suppose* $\Gamma \vdash P : \tau$ *with* $\mathsf{fv}(P) = \emptyset$ *and* $\Gamma \vdash \tau : \mathtt{tp}_H$. *Then we have* $\Gamma^\circ, u{:}\tau^\bullet \vdash [\![P]\!]_u : u{:}\tau^\bullet, \vec{u}{:}\vec{\sigma}^\circ$ *with* $\Gamma \vdash u_i : \sigma_i^?$.

2. (type preserving for processes) *Suppose* $\Gamma \vdash P : \Delta$ *and* $\mathsf{fv}(P) = \emptyset$. *Then* $\Gamma^\circ \vdash [\![P]\!]_u : \Delta^\circ, \vec{u}{:}\vec{\sigma}^\circ$ *with* $\Gamma \vdash u_i : \sigma_i^?$.

3. (operational correspondence) *Suppose* $\Gamma \vdash P : \tau$ *and* $P \longrightarrow P'$. *Then* $[\![P]\!]_u \mapsto^+ [\![P']\!]_u$ *or* $[\![P]\!] \mapsto^+ [\![P']\!]$.

4. (computational adequacy 1) *Suppose* $\Gamma \vdash P : \Delta$ *with* $x \notin \mathsf{fv}(P)$. *Then* $P \Downarrow_{a^p}$ *iff* $[\![P]\!] \Downarrow_{a^p}$.

5. (computational adequacy 2) *Suppose* $\Gamma \vdash P : \mathtt{unit}$ *with* $x \notin \mathsf{fv}(P)$. *Then* $[\![P]\!]_u \Downarrow_{u^\uparrow}$.

6. (soundness) $[\![P]\!] \cong [\![Q]\!]$ *implies* $P \cong Q$.

Then the properties in Theorem 3.1 are direct from the result in Theorem 3.2 and Proposition 7.1 in [50] by using the above proposition.

B.3 Linear-Affine Type Discipline

In this subsection, we start with coherently mixing the two typing systems, linear and affine ones, then prove the linear liveness property. This property means the answer always return to the sender in the presence of non-terminating and code passing computation. From now on we distinguish linear and affine channels by annotating modes with A for affine and L for linear ($\uparrow$ as $\uparrow_{\mathrm{A}}$ for affine output, $\uparrow_{\mathrm{L}}$ for linear output, etc.). The composition rules for modes are changed with respect to A and L as:

(la-m-1) $\overline{\downarrow_{\mathrm{L}}} = \uparrow_{\mathrm{L}}$, $\overline{!_{\mathrm{L}}} = ?_{\mathrm{L}}$, $\overline{\downarrow_{\mathrm{A}}} = \uparrow_{\mathrm{A}}$, $\overline{!_{\mathrm{A}}} = ?_{\mathrm{A}}$, and $\overline{\overline{p}} = p$.

(la-m-2) (1) $\downarrow_{\mathrm{L}} \odot \uparrow_{\mathrm{L}} = \downarrow_{\mathrm{A}} \odot \uparrow_{\mathrm{A}} = \updownarrow$ and (2) and (3) are the same by replacing ? by $\mathcal{M}_?$.

where $\mathcal{M}_p$ is defined as $\mathcal{M}_\downarrow = \{\downarrow_{\mathrm{L}}, \downarrow_{\mathrm{A}}\}$, $\mathcal{M}_\uparrow = \{\uparrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}\}$, $\mathcal{M}_! = \{!_{\mathrm{L}}, !_{\mathrm{A}}\}$, and $\mathcal{M}_? = \{?_{\mathrm{L}}, ?_{\mathrm{A}}\}$. Then the composition for types are defined as the same as **(a-t-1)/(a-t-2)** by replacing ? by $\mathcal{M}_?$.

The rules for the well-formed channel types and typing are defined in Figure 17. These rules are the same as the affine-linear rules presented in [20]. We construct the typing system where a linear output is only carried by a linear replication; and a non-linear input never suppresses linear outputs. We shall call this calculus, the *linear/affine* $\lambda\pi$*-calculus*, denoted by $\lambda\pi^{\mathsf{LA}}$.

As we already discussed in the main section (§ 3.1), the liveness property is proved via translation in Figure 16. Here we only have to modify the mapping of types as:

$$\tau^\bullet \stackrel{\text{def}}{=} (\tau^\circ)^{\uparrow_{\mathrm{L}}} \quad \tau \text{ is a HO-type} \qquad (\Delta, u{:}\sigma^\uparrow)^\bullet \stackrel{\text{def}}{=} \sigma^\circ$$

$$\mathtt{unit}^\circ \stackrel{\text{def}}{=} ((\,)^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}} \qquad (\tau_1 \to \tau_2)^\circ \stackrel{\text{def}}{=} \begin{cases} (\overline{\tau_1^\circ}\tau_2^\bullet)^{!_{\mathrm{L}}} & \tau_2 \text{ is total} \\ (\overline{\tau_1^\circ}\tau_2^\bullet)^{!_{\mathrm{A}}} & \tau_2 \text{ is partial} \end{cases}$$

$$(\Pi(y{:}\sigma)\tau)^\circ \stackrel{\text{def}}{=} \begin{cases} (\sigma^\circ\tau^\bullet)^p & (\mathsf{md}(\sigma) \in \mathcal{M}_?) \\ (\tau^\bullet)^p & (\mathsf{md}(\sigma) \in \mathcal{M}_\uparrow) \end{cases} \quad p = \begin{cases} !_{\mathrm{L}} & \tau \text{ is total} \\ !_{\mathrm{A}} & \tau \text{ is partial} \end{cases}$$

where τ is partial (non-terminating) if $\mathsf{tail}(\tau) = \Delta, u{:}\sigma^\uparrow$ and $\mathsf{md}(\sigma) = \uparrow_{\mathrm{A}}$. Others are total.

Proof for Theorem 3.2. We first define the typed transition relation essentially as the same as [38, Table 2.3] mixing the condition in [50, § 4.1]. We say Δ is closed if $\mathsf{md}(\Delta) \subseteq \mathcal{M}_! \cup \{\updownarrow\}$. W.o.l.g. we prove the case $P \equiv !u(y{:}\tau).R$ with $\tau = (\exists[c{:}(\mathtt{nat})^{\uparrow_{\mathrm{L}}}]\ulcorner c{:}(\mathtt{nat})^{\uparrow_{\mathrm{L}}}\urcorner)^{!_{\mathrm{L}}}$. Assume $P \stackrel{u\langle\ulcorner\overline{c}\langle n\rangle\urcorner\rangle}{\longrightarrow} P'$. Then we can write $P' \equiv (R\{\ulcorner\overline{c}\langle n\rangle\urcorner/y\} \,|\, P)$. Let $R' \stackrel{\text{def}}{=} R\{\ulcorner\overline{c}\langle n\rangle\urcorner/y\}$. Then we know $\Gamma \vdash R' : c : (\mathtt{nat})^{\uparrow_{\mathrm{L}}}$ (cf. [50, Proposition 4.1]). Hence we know $R' \Downarrow c^{\uparrow_{\mathrm{L}}}$ by Theorem 3.1 (3). Here we also need to check R' emits n via c. Then by $(\mathsf{In}^\exists)$ (adapting to $(\mathsf{In}^{!_{\mathrm{L}}})$), we know $\Gamma, \{x{:}(\mathtt{nat})^{\uparrow_{\mathrm{L}}}, y{:}\tau'\} \vdash R : x{:}(\mathtt{nat})^{\uparrow_{\mathrm{L}}}$ with $\tau' = \ulcorner x{:}(\mathtt{nat})^{\uparrow_{\mathrm{L}}}\urcorner$. Note that by Proposition A.5 (2), $x \notin \mathsf{fv}(R)$. Hence $\neg R \Downarrow x^{\uparrow_{\mathrm{L}}}$. We also know $\neg R \Downarrow c^{\uparrow_{\mathrm{L}}}$ by definition of $\stackrel{l}{\longrightarrow}$. So if $R' \longrightarrow\!\!\!\!\to (\nu\,\vec{a})(\overline{c}\langle V\rangle \,|\, R'')$, then $\overline{c}\langle V\rangle$ is created from the input transition $l = u\langle\ulcorner\overline{c}\langle n\rangle\urcorner\rangle$, which means $V = n$.

B.4 Branching and Reference

In this subsection, we define the typing systems for $\lambda\pi^{\mathsf{LAM}}_{<}$. The composition rules for modes are the same as in $\lambda\pi^{\mathsf{LA}}$, adding $!_{\mathrm{LM}}, !_{\mathrm{M}}$ to $\mathcal{M}_!$ and $?_{\mathrm{LM}}, ?_{\mathrm{M}}$ to $\mathcal{M}_?$. We say type σ is *mutable* if either $\mathsf{md}(\sigma) \in \{?_{\mathrm{M}}, ?_{\mathrm{LM}}\}$, $\sigma = \mathsf{w}\langle\sigma'\rangle$ or $\sigma = \mathsf{rw}\langle\sigma'\rangle$. We list the subtyping rule in Figure 18. The composition of channel types is similarly defined; we need to add only the following composition rule for with ?-mode;

$$[\tau_1 ... \oplus \tau_n]^? \odot [\tau_1' ... \oplus \tau_n' ... \oplus \tau_{n+k}]^p \stackrel{\text{def}}{=} [\tau_1 \sqcup \tau_1' ... \oplus \tau_n \sqcup \tau_n' ... \oplus \tau_{n+k}]^p$$

$[\![\mathbf{0}]\!]_\Gamma \stackrel{\text{def}}{=} \mathbf{0} \quad [\![P \mid Q]\!]_\Gamma \stackrel{\text{def}}{=} [\![P]\!]_\Gamma \mid [\![Q]\!]_\Gamma \quad [\![(\nu a{:}\sigma)P]\!]_\Gamma \stackrel{\text{def}}{=} (\nu a{:}\sigma^\circ)[\![P]\!]_{\Gamma,a\sigma}$

$[\![u(x_1{:}\sigma_1,\ldots,y_m{:}\tau_m).P]\!]_\Gamma \stackrel{\text{def}}{=} u(x_1{:}\sigma_1^\circ,\ldots,y_m{:}\overline{\tau_m^\circ}).[\![P]\!]_{\Gamma,x_1\sigma_1,\ldots,y_m\tau_m}$

$[\![!\,u(x_1{:}\sigma_1,\ldots,y_m{:}\tau_m).P]\!]_\Gamma \stackrel{\text{def}}{=} !\,u(x_1{:}\sigma_1^\circ,\ldots,y_m{:}\overline{\tau_m^\circ}).[\![P]\!]_{\Gamma,x_1\sigma_1,\ldots,y_m\tau_m}$

$[\![\overline{u}\langle V_1,\ldots,V_n\rangle]\!]_\Gamma \stackrel{\text{def}}{=} \overline{u}(c_1,\ldots,c_n)\prod\langle c_j = V_j\rangle_\Gamma$

$[\![PQ]\!]_\Gamma \stackrel{\text{def}}{=} [\![PQ]\!]^w_\Gamma \qquad (\Gamma \vdash PQ : \Delta, w{:}\sigma^\uparrow)$

$[\![PQ]\!]^w_\Gamma \stackrel{\text{def}}{=} (\nu n)([\![P]\!]^n_\Gamma \mid n(y).(\nu m)([\![Q]\!]^m_\Gamma \mid m(x).\overline{y}\langle x,w\rangle)) \quad (\Gamma \vdash Q : \tau, \tau \neq \sigma)$

$[\![Pu]\!]^w_\Gamma \stackrel{\text{def}}{=} (\nu n)([\![P]\!]^n_\Gamma \mid n(y).\overline{y}\langle u,w\rangle) \quad (\Gamma \vdash u : \sigma^?)$

$[\![Pu]\!]^w_\Gamma \stackrel{\text{def}}{=} (\nu n)([\![P]\!]^n_\Gamma \mid n(y).\overline{y}\langle w\rangle) \quad (\Gamma \vdash u : \sigma^\uparrow)$

$[\![V]\!]^m_\Gamma \stackrel{\text{def}}{=} \overline{m}(c)\langle c = V\rangle_\Gamma$

$\langle c = u\rangle_\Gamma \stackrel{\text{def}}{=} [c \to u]^{\tau^\circ} \quad (\Delta \vdash u : \tau)$

$\langle c = ()\rangle_\Gamma \stackrel{\text{def}}{=} !c(x).\overline{x}$

Case $\Gamma, x{:}\tau \vdash P : \tau' \neq \Delta$

$$\langle c = \lambda x.P\rangle_\Gamma \stackrel{\text{def}}{=} \begin{cases} !\,c(x,m).[\![P]\!]^m_{\Gamma,x\tau} & (\tau \neq \sigma^\uparrow) \\ !\,c(m).[\![P]\!]^m_{\Gamma,x\tau} & (\tau = \sigma^\uparrow) \end{cases}$$

Case $\Gamma, x{:}\tau \vdash P : \Delta, y{:}\sigma_0^\uparrow$

$$\langle c = \lambda x.P\rangle_\Gamma \stackrel{\text{def}}{=} \begin{cases} !\,c(x,y).[\![P]\!]_{\Gamma,x\tau} & (\tau \neq \sigma^\uparrow) \\ !\,c(y).[\![P]\!]_{\Gamma,x\tau} & (\tau = \sigma^\uparrow) \end{cases}$$

Copy Cat

$$[u \to w]^{(\vec{\sigma})^p} \stackrel{\text{def}}{=} \begin{cases} u(\vec{y}).\overline{w}(\vec{y}')\Pi_i[y_i' \to y_i]^{\overline{\sigma_i}} & (p = \downarrow) \\ !u(\vec{y}).\overline{w}(\vec{y}')\Pi_i[y_i' \to y_i]^{\overline{\sigma_i}} & (p = !) \end{cases}$$

Figure 16: Encoding of $\lambda\pi^{\text{L}}$-Calculus into π^{L}-Calculus

(Well-Formed Types)

$$\text{(t-}\downarrow\text{)}\ \frac{p \in \mathcal{M}_\uparrow \quad \mathsf{md}(\sigma_i) \in \mathcal{M}_? \qquad \Gamma, x_1{:}\sigma_1,..,x_n{:}\sigma_n \vdash \tau_j : \mathtt{tp}_H}{\Gamma \vdash (\Pi[u_1{:}\sigma_1,\ldots,u_n{:}\sigma_n]\tau_1,\ldots,\tau_m)^p : p} \qquad \text{(t-!}_{\text{L}}\text{)}\ \frac{\mathsf{md}(\sigma_i) \in \mathcal{M}_? \ (i \neq n) \quad \mathsf{md}(\sigma_n) \in \mathcal{M}_\uparrow \qquad \Gamma, x_1{:}\sigma_1,..,x_n{:}\sigma_n \vdash \tau_j : \mathtt{tp}_H}{\Gamma \vdash (\Pi[u_1{:}\sigma_1,\ldots,u_n{:}\sigma_n]\tau_1,\ldots,\tau_m)^{!_{\text{L}}} : !_{\text{L}}} \qquad \text{(t-!}_{\text{A}}\text{)}\ \frac{\mathsf{md}(\sigma_i) \in \mathcal{M}_? \ (i \neq n) \quad \mathsf{md}(\sigma_n) = \uparrow_{\text{A}} \qquad \Gamma, x_1{:}\sigma_1,..,x_n{:}\sigma_n \vdash \tau_j : \mathtt{tp}_H}{\Gamma \vdash (\Pi[u_1{:}\sigma_1,\ldots,u_n{:}\sigma_n]\tau_1,\ldots,\tau_m)^{!_{\text{A}}} : !_{\text{A}}}$$

Others are from Figure 12.

(Typing Rules)

$$(\mathsf{In}^{\downarrow_{\text{L}}})\ \frac{\Gamma \vdash u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{\downarrow_{\text{L}}} \qquad \Gamma, \vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau} \vdash P :\ \vec{x}{:}\vec{\sigma}, \uparrow_{\text{L}}\Delta^{-u}, \uparrow_{\text{A}}?\Delta'}{\Gamma \vdash u(\vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau}).P : u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{\downarrow_{\text{L}}} \to \Delta, \Delta'} \qquad (\mathsf{In}^{!_{\text{L}}})\ \frac{\Gamma \vdash u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{!_{\text{L}}} \qquad \Gamma, \vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau} \vdash P :\ \vec{x}{:}\vec{\sigma}, ?_{\text{L}}\Delta^{-u}, ?_{\text{A}}\Delta'}{\Gamma \vdash !\,u(\vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau}).P :\ u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{!_{\text{L}}} \to \Delta, \Delta'}$$

$$(\mathsf{In}^{\downarrow_{\text{A}}})\ \frac{\Gamma \vdash u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{\downarrow_{\text{A}}} \qquad \Gamma, \vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau} \vdash P :\ \vec{x}{:}\vec{\sigma}, \uparrow_{\text{A}}?\Delta}{\Gamma \vdash u(\vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau}).P : u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{\downarrow_{\text{A}}}, \Delta} \qquad (\mathsf{In}^{!_{\text{A}}})\ \frac{\Gamma \vdash u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{!_{\text{A}}} \qquad \Gamma, \vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau} \vdash P :\ \vec{x}{:}\vec{\sigma}, ?_{\text{L}}?_{\text{A}}\Delta^{-u}}{\Gamma \vdash !\,u(\vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau}).P :\ u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{!}, \Delta}$$

Others are from Figure 13 by replacing ! by $\mathcal{M}_!$ in (Res) and ? by $\mathcal{M}_?$ in (Weak-?).

Figure 17: Linear-Affine Higher-Order Typing System

(Well-Formedness)

$$\text{(t-ref)}\ \frac{\Gamma \vdash \sigma : !}{\Gamma \vdash \mathsf{ref}\,\langle\sigma\rangle, \overline{\mathsf{w}\,\langle\sigma\rangle}, \overline{\mathsf{rw}\,\langle\sigma\rangle} : !_{\mathrm{LM}}} \quad \text{(t-rw)}\ \frac{\Gamma \vdash \sigma : !}{\Gamma \vdash \mathsf{r}\langle\sigma\rangle, \mathsf{w}\,\langle\sigma\rangle, \mathsf{rw}\,\langle\sigma\rangle : ?_{\mathrm{LM}}} \quad \text{(t-pair)}\ \frac{\Gamma \vdash \sigma_1 : p_{\mathrm{O}} \quad \Gamma \vdash \sigma_1 < \sigma_2}{\Gamma \vdash \langle\sigma_1, \overline{\sigma_2}\rangle : p_{\mathrm{O}} \odot \overline{p_{\mathrm{O}}}}$$

(Subtyping Rules)

$$\text{(s-base)}\ \frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \le \alpha} \qquad \text{(s-abs}_H\text{)}\ \frac{\Gamma \vdash \tau_{1i} \to \tau_{2i} : \mathtt{tp}_H \quad \Gamma \vdash \tau_{12} \le \tau_{11} \quad \Gamma \vdash \tau_{21} \le \tau_{22}}{\Gamma \vdash \tau_{11} \to \tau_{21} \le \tau_{12} \to \tau_{22}}$$

$$\text{(s-abs}_N\text{)}\ \frac{\Gamma \vdash \Pi(x{:}\sigma_i)\tau_i : \mathtt{tp}_H \quad \Gamma \vdash \sigma_1 \le \sigma_2 \quad \Gamma, x{:}\sigma_2 \vdash \tau_1 \le \tau_2}{\Gamma \vdash \Pi(x{:}\sigma_1)\tau_1 \le \Pi(x{:}\sigma_2)\tau_2} \qquad \text{(s-pro)}\ \frac{\mathsf{dom}(\Delta_1) = \mathsf{dom}(\Delta_2) \quad \Gamma \vdash \Delta_2 : \mathtt{tp}_P \quad \Gamma \vdash \Delta_1(a) \le \Delta_2(a)}{\Gamma \vdash \Delta_1 \le \Delta_2 \odot \Delta}$$

$$\text{(s-select)}\ \frac{\Gamma \vdash [\oplus_{j\in I_1} \tau_{ji}]^p : p \quad \Gamma \vdash \tau_{j1} \le \tau_{j2} \quad I_1 \subseteq I_2}{\Gamma \vdash [\oplus_{j\in I_1} \tau_{j1}]^p \le [\oplus_{l\in I_2} \tau_{l2}]^p} \qquad \text{(s-read)}\ \frac{\Gamma \vdash \sigma : !}{\Gamma \vdash \mathsf{r}\langle\sigma\rangle \le \mathsf{rw}\,\langle\sigma\rangle} \qquad \text{(s-write)}\ \frac{\Gamma \vdash \sigma : !}{\Gamma \vdash \mathsf{w}\,\langle\sigma\rangle \le \mathsf{rw}\,\langle\sigma\rangle}$$

$$\text{(s-dual)}\ \frac{\Gamma \vdash \sigma_1 \le \sigma_2}{\Gamma \vdash \overline{\sigma_2} \le \overline{\sigma_1}} \qquad \text{(s-pair-1)}\ \frac{\Gamma \vdash \langle\sigma_1, \sigma_2\rangle : p \quad \Gamma \vdash \sigma \le \sigma_1 \quad \text{or} \quad \sigma \le \sigma_2}{\Gamma \vdash \sigma \le \langle\sigma_1, \sigma_2\rangle} \qquad \text{(s-pair-2)}\ \frac{\Gamma \vdash \langle\sigma_{1i}, \sigma_{2i}\rangle : p \quad i = 1,2 \quad \Gamma \vdash \sigma_{j1} \le \sigma_{j2} \quad j = 1,2}{\Gamma \vdash \langle\sigma_{11}, \sigma_{21}\rangle \le \langle\sigma_{12}, \sigma_{22}\rangle}$$

(Typing Rules)

$$\text{(Sub}_H\text{)}\ \frac{\Gamma \vdash P : \tau \quad \Gamma \vdash \tau \le \tau'}{\Gamma \vdash P : \tau'} \qquad \text{(Sub}_N\text{)}\ \frac{\Gamma \vdash u : \sigma \quad \Gamma \vdash \sigma' \le \sigma}{\Gamma \vdash u : \sigma'} \qquad \text{(Bra}^{\downarrow_{\mathrm{A}}}\text{)}\ \frac{\Gamma \vdash u : [\&_{i\in I} \Pi[\vec{x}_i{:}\vec{\sigma}_i]\vec{\tau}_i]^{\downarrow_{\mathrm{A}}} \quad \Gamma, \vec{x}_i{:}\vec{\sigma}_i, \vec{y}_i{:}\vec{\tau}_i \vdash P_i : \vec{x}_i{:}\vec{\sigma}_i, \uparrow_{\mathrm{A}}?\Delta^{-u\vec{x}_i}}{\Gamma \vdash u[\&_{i\in I}(\vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau}).P_i] : u : [\&_{i\in I} \Pi[\vec{x}_i{:}\vec{\sigma}_i]\vec{\tau}_i]^{\downarrow_{\mathrm{A}}}, \Delta}$$

$$\text{(Sel)}\ \frac{\Gamma \vdash u : [\oplus_{l\in I} \Pi[\vec{x}_l{:}\overline{\vec{\sigma}_l}]\vec{\tau}_i]^{p_{\mathrm{O}}} \quad \Gamma \vdash v_i : \sigma_{il}\{\vec{v}/\vec{x}_l\} \quad \Gamma \vdash V_j : \tau_{jl}\{\vec{v}/\vec{x}_l\}}{\Gamma \vdash \overline{u}\,\mathtt{in}_l\langle\vec{v}, \vec{V}\rangle \ : u : [\oplus_l \Pi[\vec{x}_l{:}\vec{\sigma}_l]\vec{\tau}_l]^{p_{\mathrm{O}}}, \vec{v}{:}\vec{\sigma}_l\{\vec{v}/\vec{x}_l\}}$$

$$\text{(Ref)}\ \frac{\Gamma \vdash u : \mathsf{ref}\,\langle\sigma\rangle \quad \Gamma \vdash v : \overline{\sigma}}{\Gamma \vdash \mathsf{ref}\,\langle uv\rangle : \mathsf{ref}\,\langle\sigma\rangle, v{:}\overline{\sigma}} \qquad \text{(In}^{!_{\mathrm{A}}}\text{)}\ \frac{\Gamma \vdash u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{!_{\mathrm{A}}} \quad \Gamma, \vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau} \vdash P : \ \vec{x}{:}\vec{\sigma}, ?_{\mathrm{L}}?_{\mathrm{A}}?_r\Delta^{-u}}{\Gamma \vdash\ !u(\vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau}).P : \ u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{!}, \Delta} \qquad \text{(In}^{!_{\mathrm{M}}}\text{)}\ \frac{\Gamma \vdash u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{!_{\mathrm{M}}} \quad \Gamma, \vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau} \vdash P : \ \vec{x}{:}\vec{\sigma}, ?\Delta^{-u}}{\Gamma \vdash\ !u(\vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau}).P : \ u : (\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{!_{\mathrm{M}}}, \Delta}$$

For well-formedness, we assume the same constraint for branching/selection channel types for each summand as in Figure 17. Subtyping rule for the unary output is defined as (s-select) by setting $I_1 = I_2 = \{1\}$. ($\mathsf{Bra}^{\downarrow_{\mathrm{L}}, !_{\mathrm{L}}, !_{\mathrm{A}}}$) are defined as ($\mathsf{In}^{\downarrow_{\mathrm{L}}, !_{\mathrm{L}}, !_{\mathrm{A}}}$) in Figure 17. Δ^r in ($\mathsf{In}^{!_{\mathrm{A}}}$) means it does not contain mutable types. The subtyping of $\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau}$ is defined as (s-abs$_N$).

Figure 18: Higher-Order Branching, Reference and Subtyping System for $\lambda\pi^{\mathsf{LAM}}_{<}$

$\mathtt{unit} \sqcap \mathtt{unit} \stackrel{\text{def}}{=} \mathtt{unit} \sqcup \mathtt{unit} \stackrel{\text{def}}{=} \mathtt{unit}$

$(\tau_{11} \to \tau_{12}) \sqcap (\tau_{21} \to \tau_{22}) \stackrel{\text{def}}{=} \tau_{11} \sqcup \tau_{21} \to \tau_{12} \sqcap \tau_{22}$

$(\tau_{11} \to \tau_{12}) \sqcup (\tau_{21} \to \tau_{22}) \stackrel{\text{def}}{=} \tau_{11} \sqcap \tau_{21} \to \tau_{12} \sqcup \tau_{22}$

$\Pi(x{:}\sigma_1)\tau_1 \sqcap \Pi(x{:}\sigma_2)\tau_2 \stackrel{\text{def}}{=} \Pi(x{:}\sigma_1 \sqcap \sigma_2)\tau_1 \sqcap \tau_2$

$\Pi(x{:}\sigma_1)\tau_1 \sqcup \Pi(x{:}\sigma_2)\tau_2 \stackrel{\text{def}}{=} \Pi(x{:}\sigma_1 \sqcup \sigma_2)\tau_1 \sqcup \tau_2$

$[\oplus_{i\in I}\Pi[\vec{x}_i{:}\vec{\sigma}_{1i}]\vec{\tau}_{1i}]^p \sqcap [\oplus_{j\in J}\Pi[\vec{x}_j{:}\vec{\sigma}_{2j}]\vec{\tau}_{2j}]^p = [\oplus_{i\in I\cap J}\Pi[\vec{x}_i{:}\vec{\sigma}_{1i}\sqcap\vec{\sigma}_{2i}]\vec{\tau}_{1i}\sqcap\vec{\tau}_{2i}]^p$

$[\oplus_{i\in I}\Pi[\vec{x}_i{:}\vec{\sigma}_{1i}]\vec{\tau}_{1i}]^p \sqcup [\oplus_{j\in J}\Pi[\vec{x}_j{:}\vec{\sigma}_{2j}]\vec{\tau}_{2j}]^p = [\oplus_{i\in I\cup J}\Pi[\vec{x}_i{:}\vec{\sigma}_{1i}\sqcup\vec{\sigma}_{2i}]\vec{\tau}_{1i}\sqcup\vec{\tau}_{2i}]^p$

$[\&_{i\in I}\Pi[\vec{x}_i{:}\vec{\sigma}_{1i}]\vec{\tau}_{1i}]^p \sqcap [\&_{j\in J}\Pi[\vec{x}_j{:}\vec{\sigma}_{2j}]\vec{\tau}_{2j}]^p = [\&_{i\in I\cup J}\Pi[\vec{x}_i{:}\vec{\sigma}_{1i}\sqcap\vec{\sigma}_{2i}]\vec{\tau}_{1i}\sqcup\vec{\tau}_{2i}]^p$

$[\&_{i\in I}\Pi[\vec{x}_i{:}\vec{\sigma}_{1i}]\vec{\tau}_{1i}]^p \sqcup [\&_{j\in J}\Pi[\vec{x}_j{:}\vec{\sigma}_{2j}]\vec{\tau}_{2j}]^p = [\&_{i\in I\cap J}\Pi[\vec{x}_i{:}\vec{\sigma}_{1i}\sqcup\vec{\sigma}_{2i}]\vec{\tau}_{1i}\sqcap\vec{\tau}_{2i}]^p$

$\langle \sigma_{\mathrm{I}}, \sigma_0\rangle \sqcap \langle \sigma'_{\mathrm{I}}, \sigma'_0\rangle \stackrel{\text{def}}{=} \langle \sigma_{\mathrm{I}} \sqcap \sigma_{\mathrm{I}}, \sigma_0 \sqcap \sigma'_0\rangle$

$\langle \sigma_{\mathrm{I}}, \sigma_0\rangle \sqcup \langle \sigma'_{\mathrm{I}}, \sigma'_0\rangle \stackrel{\text{def}}{=} \langle \sigma_{\mathrm{I}} \sqcup \sigma'_{\mathrm{I}}, \sigma_0 \sqcup \sigma'_0\rangle$ $\sigma_{\mathrm{I}} \geq \sigma'_0$ and $\sigma'_{\mathrm{I}} \geq \sigma_0$ else undefined.

$\Gamma_1 \sqcap \Gamma_2 \stackrel{\text{def}}{=} \Gamma_1 \setminus \mathsf{dom}(\Gamma_2) \cup \Gamma_2 \setminus \mathsf{dom}(\Gamma_1) \cup \{u{:}(\Gamma_1(u) \sqcap \Gamma_2(u)) \mid u \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)\}$

$\Gamma_1 \sqcup \Gamma_2 \stackrel{\text{def}}{=} \{u{:}(\Gamma_1(u) \sqcup \Gamma_2(u)) \mid u \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)\}$

Unary output/input are defined as the case $I = J = \{1\}$ in selection/branching types. The reference types are defined as branching/selection types.

Figure 19: Partial Meet and Join Operators

where $\sqcup$ is a partial join operator defined in Figure 19. Formally we define:

(s-t-1) $\sigma_{\mathrm{I}} \odot \sigma_0 = \langle \sigma_{\mathrm{I}}, \sigma_0\rangle = \langle \sigma_0, \sigma_{\mathrm{I}}\rangle$ with $\overline{\mathsf{md}(\sigma_{\mathrm{I}})} = \mathsf{md}(\sigma_0)$.

(s-t-2) $\sigma_1 \odot \langle \sigma_2, \sigma\rangle = \sigma_1 \odot \langle \sigma, \sigma_2\rangle = \langle \sigma_1 \odot \sigma_2, \sigma\rangle = \langle \sigma, \sigma_1 \odot \sigma_2\rangle$ $\sigma_1 \odot \sigma_2 = \sigma_1 \sqcup \sigma_2$ with $p \in \mathcal{M}_?$.

In the above, the unary case is defined setting $I = J = \{1\}$. Note that we only have to check the composability of the outmost node. The definition of $\Delta_1 \odot \Delta_2$ remains as the same. As in [52, Proposition 3.1], we can easily check under an arbitrary well-formed environment, the subtyping relation over types is a partial order and finite bounded complete ([52, Definition 3.1]). We have:

PROPOSITION B.5. *1.* (decidability of $<$) *It is decidable whether, given an environment Γ, the judgement $\Gamma \vdash \tau < \tau'$ is derivable.*
2. Suppose $\Gamma \vdash \Delta_i : \mathtt{tp}_P$ $(i = 1, 2)$ and $\Delta_1 \asymp \Delta_2$. Then $\Delta_1 \odot \Delta_2$ is always defined and $\Delta_1 \odot \Delta_2 = \Delta_1 \sqcup \Delta_2$.

We list the typing rules in Figure 18. This is a straightforward integration with the system in [20, § 4]. See [20, § 4] for the detailed explanation. Due to the branching subtyping for the reference agent, we can simplify the rules in [20]. We then obtain the same basic properties as in the previous section in $\lambda\pi^{\mathsf{LA}}$ including the subject reduction theorem. Other properties such as the linear liveness property are proved via the correctness of the translation,[8] which is proved together with the following extended proposition.

PROPOSITION B.6. *Suppose $\Gamma \vdash \tau_1 \leq \tau_2$. Then $\Gamma^\circ \vdash \tau_1^\bullet \leq \tau_2^\bullet$ and $\Gamma^\circ \vdash \tau_1^\circ \leq \tau_2^\circ$.*

[8]The reference type/term are mapped homomorphically, e.g. $(\mathsf{ref}\langle\sigma\rangle)^\circ = \mathsf{ref}\langle\sigma^\circ\rangle$. The branching/selection are mapped as the same as the unary case.

We conclude with a small example of a branching subtyping.

EXAMPLE B.7. (subtyping) First we will show that the atomic subtyping such as $\mathtt{bool} < \mathtt{nat}$ can be simulated using the encoding into the $\lambda\pi^{\mathsf{LA}}_{<}$. First boolean agents are encoded as follows:

$$[\![\mathtt{true}]\!]_u \stackrel{\text{def}}{=} \overline{u}(c)!c(e).\overline{e}\mathtt{in}_1 \quad \text{and} \quad [\![\mathtt{false}]\!]_u \stackrel{\text{def}}{=} \overline{u}(c)!c(e).\overline{e}\mathtt{in}_2$$

u has an output type $\mathtt{bool}^\bullet \stackrel{\text{def}}{=} (([\ \oplus\ ]^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}})^{\uparrow_{\mathrm{L}}}$. These agents are the special case of the following natural number agent which has a type $\mathtt{nat}^\bullet \stackrel{\text{def}}{=} (([\oplus_{i\in\mathbb{N}}\ ]^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}})^{\uparrow_{\mathrm{L}}}$.

$$[\![n]\!]_u \stackrel{\text{def}}{=} \overline{u}(c)!c(e).\overline{e}\mathtt{in}_n$$

Note that for any well formed environment Γ, we have:

$$\Gamma \vdash \mathtt{bool}^\bullet \leq \mathtt{nat}^\bullet$$

To understand subtyping of process types, suppose the following receiver which provides the boolean branch:

$$\mathtt{if}\ u\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \stackrel{\text{def}}{=} u(c).\overline{c}(c')c'[.P_1\ \&\ .P_2]$$

Let $\tau = \ulcorner u{:}\mathtt{bool}^\bullet \urcorner$ and assume $\Gamma \vdash P_i : \Delta$. Then under the environment $\Gamma' = \Gamma, a{:}\langle \sigma_a, \overline{\sigma_a}\rangle, u{:}\langle \mathtt{bool}^\bullet, \overline{\mathtt{bool}^\bullet}\rangle$ with $\sigma_a = (\tau)^{?_{\mathrm{L}}}$, the following agent is typable.

$$P \stackrel{\text{def}}{=} !a(x{:}\tau).\mathit{run}\, x \,|\, \mathtt{if}\ u\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$$

Then one can check $P \,|\, \overline{a}\langle \ulcorner \mathfrak{V} \urcorner\rangle$, $P \,|\, \overline{a}\langle \ulcorner [\![\mathtt{true}]\!]_u \urcorner\rangle$ and $P \,|\, \overline{a}\langle \ulcorner [\![\mathtt{false}]\!]_u \urcorner\rangle$ are typable while $P \,|\, \overline{a}\langle [\![n]\!]_u\rangle$, which causes the run-time-error, is not typable, since

$$\Gamma' \vdash a{:}(\ulcorner \mathfrak{V} \urcorner)^{?_{\mathrm{L}}} \text{ and } a{:}(\ulcorner u{:}\mathtt{bool}^\bullet \urcorner)^{?_{\mathrm{L}}} \text{ but } \Gamma' \not\vdash a{:}(\ulcorner u{:}\mathtt{nat}^\bullet \urcorner)^{?_{\mathrm{L}}}$$

C. SECRECY AND ACCESS CONTROL

C.1 Secrecy

In this subsection, we show the definitions omitted from §.4.1 and §.4.2.

Tampering Levels. First we define the *tamper level* of types, at which the process may affect, or tamper, the environment.

DEFINITION C.1. τ is *immediately tampering* if either $\tau = [\oplus_i \vec{\tau}_i]^{\uparrow_{\mathrm{L}}}_s$ or $\mathsf{md}(\tau) \in \{\uparrow_{\mathrm{A}}, ?_{\mathrm{M}}, ?_{\mathrm{LM}}\}$. τ is *innocuous* if $\tau = \mathtt{unit}$ or $\mathsf{md}(\tau) \in \{?_{\mathrm{A}}, ?_{\mathrm{L}}, \updownarrow\}$. Then the *tampering level* of τ, denoted by $\mathsf{tamper}(\tau)$, is inductively given by:

$$\begin{array}{rcll}
\mathsf{tamper}(\tau) & = & \mathsf{sec}(\tau) & \tau \text{ is immediately tampering} \\
\mathsf{tamper}(\tau) & = & \top & \tau \text{ is innocuous.} \\
\mathsf{tamper}(\tau_1 \to \tau_2) & = & \mathsf{tamper}(\tau_2) & \\
\mathsf{tamper}(\Pi(x{:}\sigma)\tau) & = & \mathsf{tamper}(\tau) & \\
\mathsf{tamper}(\{u_i{:}\sigma_i\}) & = & \sqcap\{\mathsf{tamper}(\sigma_i)\} & \\
\mathsf{tamper}((\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^p_s) & = & \sqcap\{\mathsf{tamper}(\sigma_i)\} & (p \in \mathcal{M}_{!,\downarrow}) \\
\mathsf{tamper}((\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{\uparrow_{\mathrm{L}}}_s) & = & \sqcap\{\mathsf{tamper}(\sigma_i)\} \cup \{\mathsf{tamper}(\tau_j)\} & \\
\mathsf{tamper}([\&_i\Pi[\vec{x}_i{:}\vec{\sigma}_i]\vec{\tau}_i]^p_s) & = & \sqcap\{\mathsf{tamper}(\sigma_{ij})\} & (p \in \mathcal{M}_{!,\downarrow}) \\
\mathsf{tamper}(\langle \sigma, \sigma'\rangle) & = & \mathsf{tamper}(\sigma) & (\mathsf{md}(\sigma) \in \mathcal{M}_!)
\end{array}$$

Intuitively, a type is immediately tampering if it emits non-trivial information at the time of interaction; then a channel which carries the immediate tamperting channels at level s affects the environment at the same level; innocuous channels, on the other hand, never have such a latend effect. See [20, Example 3.3] for more examples and discussions.

(Types)

$$(u_1:\sigma_1,...,u_n:\sigma_n,u:\sigma^{\uparrow})^{\bullet}_{\Gamma} \stackrel{\text{def}}{=} (u_1:\sigma_1,...,u_n:\sigma_n,u:\sigma^{\uparrow})^{\bullet}_{\Gamma}$$
u_i appears in the order of $u_1,...,u_n$ in Γ

$$(\mathsf{ref}_s\langle\sigma\rangle)^{\circ}_{\Gamma} \stackrel{\text{def}}{=} \mathsf{ref}_s\langle(\sigma)^{\circ}_{\Gamma}\rangle \qquad (\mathsf{r}_s\langle\sigma\rangle)^{\circ}_{\Gamma} \stackrel{\text{def}}{=} \mathsf{r}_s\langle(\sigma)^{\circ}_{\Gamma}\rangle$$
$$(\mathsf{w}_s\langle\sigma\rangle)^{\circ}_{\Gamma} \stackrel{\text{def}}{=} \mathsf{w}_s\langle(\sigma)^{\circ}_{\Gamma}\rangle \qquad (\mathsf{rw}_s\langle\sigma\rangle)^{\circ}_{\Gamma} \stackrel{\text{def}}{=} \mathsf{rw}_s\langle(\sigma)^{\circ}_{\Gamma}\rangle$$
$$(\Pi(y:\sigma)\tau)^{\circ}_{\Gamma} \stackrel{\text{def}}{=} \begin{cases} (\sigma^{\circ}_{\Gamma}\tau^{\bullet}_{\Gamma,y:\sigma})^{p} & (\mathsf{md}(\sigma)\in\mathcal{M}_?) \\ (\tau^{\bullet}_{\Gamma,y:\sigma})^{p} & (\mathsf{md}(\sigma)\in\mathcal{M}_{\uparrow}) \end{cases}$$
$$((\Pi[x_1:\sigma_1,...,x_n:\sigma_n]\tau_1,...,\tau_m)^{p_1}_{s})^{\circ}_{\Gamma} \stackrel{\text{def}}{=} ((\sigma_1)^{\circ}_{\Gamma_1},...,(\sigma_n)^{\circ}_{\Gamma_n},\overline{(\tau_1)^{\circ}_{\Gamma_n}},...,\overline{(\tau_m)^{\circ}_{\Gamma_n}})^{p_1}_{s}$$
with $\Gamma_i = \Gamma,x_1:\sigma_1,...,x_{i-1}:\sigma_{i-1}$

The mapping for output channel type is similarly defined. Others are from §. B.4 by replacing $(\cdot)^{\circ}/(\cdot)^{\bullet}$ by $(\cdot)^{\circ}_{\Gamma}/(\cdot)^{\bullet}_{\Gamma}$.

(Terms)

$$[\![PQ]\!]_{\Gamma} \stackrel{\text{def}}{=} [\![PQ]\!]^{w}_{\Gamma} \stackrel{\text{def}}{=} [\![PQ]\!]^{w,\vec{v}}_{\Gamma} \qquad (\Gamma\vdash PQ:\vec{v}:\vec{\sigma},w:\sigma^{\uparrow})$$
$$[\![PQ]\!]^{w,\vec{v}}_{\Gamma} \stackrel{\text{def}}{=} (\nu n)([\![P]\!]^{n}_{\Gamma}\,|\,n(y).(\nu m)([\![Q]\!]^{m}_{\Gamma}\,|\,m(x).\overline{y}\langle x,\vec{v},w\rangle))$$
$$[\![Pu]\!]^{w,\vec{v}}_{\Gamma} \stackrel{\text{def}}{=} (\nu n)([\![P]\!]^{n}_{\Gamma}\,|\,n(y).\overline{y}\langle u,\vec{v},w\rangle) \qquad (\Gamma\vdash u:\sigma^{?})$$
$$[\![Pu]\!]^{w,\vec{v}}_{\Gamma} \stackrel{\text{def}}{=} (\nu n)([\![P]\!]^{n}_{\Gamma}\,|\,n(y).\overline{y}\langle\vec{v},w\rangle) \qquad (\Gamma\vdash u:\sigma^{\uparrow})$$

Case $\Gamma,x:\tau\vdash P:\vec{v}:\vec{\sigma},y:\sigma^{\uparrow}_0$

$$\langle u=\lambda x.P\rangle_{\Gamma} \stackrel{\text{def}}{=} \begin{cases} !\,c(x,\vec{v},y).[\![P]\!]_{\Gamma,x\tau} & (\tau\neq\sigma^{\uparrow}) \\ !\,c(\vec{v},y).[\![P]\!]_{\Gamma,x\tau} & (\tau=\sigma^{\uparrow}) \end{cases}$$

Others are from Figure 16. In the

Figure 20: Encoding of $\lambda\pi^{\text{LAM}}$-Calculus into π^{LAM}-Calculus

Typing. First we need to consider the subtyping relation for security levels on channel types. This is defined as:

$$\frac{\begin{array}{l}\text{(s-select)}\\ \Gamma\vdash[\oplus_{j\in I_i}\tau_{ji}]^{p}_{s_i}:p \quad s_1\sqsubseteq s_2\\ \Gamma\vdash\tau_{j1}\leq\tau_{j2} \quad I_1\subseteq I_2\end{array}}{\Gamma\vdash[\oplus_{j\in I_1}\tau_{j1}]^{p}_{s_1}\leq[\oplus_{l\in I_2}\tau_{l2}]^{p}_{s_2}}$$

The subtyping rules for reference is defined as the same as above (i.e. $\mathsf{rw}_s\langle\sigma\rangle < \mathsf{rw}_{s'}\langle\sigma\rangle$ if $s\sqsubseteq s'$). Other subtyping rules remain as the same as Figure 18. For the well-formedness, we add the *structural security* condition to the reference and $?_{\text{M}}$-types [20, Definition 6.3]. This is simply defined by adding $s\sqsubseteq\mathsf{tamper}(\sigma)$ in (t-ref,rw) in Figure 18 and $s\sqsubseteq\{\mathsf{tamper}(\sigma_{ij})\}\sqcap\{\mathsf{tamper}(\tau_{ij})\}$ in (t-$!_{\text{M}}$) in Figure 17.

The typing system is a straightforward integration with $\pi^{\text{LA}}_{\text{sec}}$ in [20]; we only have to change ($\mathsf{In}^{\downarrow_{\text{A}}}$), ($\mathsf{Bra}^{\uparrow_{\text{A}}}$) and ($\mathsf{Bra}^{\uparrow_{\text{L}}}$) saying we care the secrecy level of the body since the input directly receives information from the output which immediately tampers.

$$(\mathsf{In}^{\downarrow_{\text{A}}})\ \frac{\begin{array}{l}\Gamma\vdash u:(\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^{\downarrow_{\text{A}}}_{s}\\ \Gamma,\vec{x}:\vec{\sigma},\vec{y}:\vec{\tau}\vdash P:\ \vec{x}:\vec{\sigma},\uparrow_{\text{A}}?\Delta\end{array}}{\Gamma\vdash u(\vec{x}:\vec{\sigma},\vec{y}:\vec{\tau}).P:u:(\Pi[\vec{x}:\vec{\sigma}]\vec{\tau})^{\downarrow_{\text{A}}}_{s},\Delta} \quad (s\sqsubseteq\mathsf{tamper}(\Delta))$$

The detailed illustration of this rule is found in [20, §. 2,3]. Note we do not have to change other rules including functionals.

Proving Non-Interference. As already discussed in the main section, we use the type preserving mapping for the proofs. In order to preserve the tampering levels, we need to modify the linearity-preserving encoding used in the previous section. The definition of the modified encoding is found in Figure 20. The only difference is that all process names appeared in a process type are bound in the environment called since immediate tampering channels now contain $?_{\text{M}}$ and $?_{\text{LM}}$. Using this encoding, the proof of Proposition 4.1 is mechanical as done in Proposition B.4 using the following additional lemma.

LEMMA C.2. (tampering preserving for types)
Suppose $\Gamma\vdash\tau:\mathtt{tp}$. *Then* $\mathsf{tamper}(\tau)=\mathsf{tamper}(\tau^{\bullet})$.
Also we have: $\mathsf{tamper}(\tau_1)\sqsubseteq\mathsf{tamper}(\tau_2)$ *implies* $\mathsf{tamper}(\tau^{\bullet}_1)\sqsubseteq\mathsf{tamper}(\tau^{\bullet}_2)$ *and* $\mathsf{tamper}(\tau^{\circ}_1)\sqsubseteq\mathsf{tamper}(\tau^{\circ}_2)$.

Now we prove Theorem 4.2 as follows. The first implication is by Proposition 4.1 (1), the second implication is by Proposition 6.8 in [20] (non-interference theorem), and the third is by Proposition 4.1 (2). Below we assume $\mathsf{tamper}(\Delta)=s$.

$$\begin{array}{lll}\Gamma\vdash P_{1,2}:\Delta & \Rightarrow & \Gamma^{\circ}\vdash[\![P_{1,2}]\!]:\Delta^{\circ}\text{ with }\mathsf{tamper}(\Delta^{\circ})=s\\ & \Rightarrow & [\![P_1]\!]\cong_{s'}[\![P_2]\!]\text{ with }s\not\sqsubseteq s'\\ & \Rightarrow & P_1\cong_{s'}P_2\end{array}$$

hence done.

C.2 Access Control

In this subsection, we lists the definitions omitted from § 4.3. Assume a finite set of principals names $\mathcal{P}$ and $\mathsf{a},\mathsf{a}',..\subseteq\mathcal{P}$ are authority levels. Principal names are ranged over $\mathsf{p},\mathsf{p}',...$ Write $\mathsf{a}_1\sqsubseteq\mathsf{a}_2$ if a_1 has a lower authority level than a_2 (which is equivalent to $\mathsf{a}_2\subseteq\mathsf{a}_1$). The syntax and types are extended as:

$$\begin{array}{lcl} P & ::= & .. \mid \mathtt{set}(\mathsf{a})\{P\} \mid !\,u[\&^{\mathsf{a}_i}_i(\vec{y}_i).P_i] \mid \mathsf{ref}^{\mathsf{a}_1,\mathsf{a}_2}_s\langle uv\rangle \\ \mathsf{P} & ::= & \mathbf{0} \mid \{P\}_{\mathsf{a}} \mid \mathsf{P}_1\,|\,\mathsf{P}_2 \mid (\nu a:\sigma)\mathsf{P} \\ \sigma & ::= & .. \mid [\&^{\mathsf{a}_i}_{i\in I}\tau_i]^{p_1}_{s} \mid [\oplus^{\mathsf{a}_i}_{i\in I}\tau_i]^{p_{\text{O}}}_{s} \quad \text{with } p\in\mathcal{M}_! \\ & \mid & \mathsf{ref}^{\mathsf{a}_r,\mathsf{a}_w}_{s}\langle\sigma\rangle \mid \overline{\mathsf{r}^{\mathsf{a}}_{s}\langle\sigma\rangle} \mid \overline{\mathsf{w}^{\mathsf{a}}_{s}\langle\sigma\rangle} \mid \mathsf{rw}^{\mathsf{a}_1,\mathsf{a}_2}_{s}\langle\sigma\rangle \mid \mathsf{r}^{\mathsf{a}}_{s}\langle\sigma\rangle \mid \mathsf{w}^{\mathsf{a}}_{s}\langle\sigma\rangle \\ \tau & ::= & .. \mid \Delta,\mathsf{a} \end{array}$$

In the process syntax, "$\mathtt{set}(\mathsf{a})\{P\}$" sets P at the authority level a. The i-th branch of $!\,u[\&^{\mathsf{a}_i}_i(\vec{y}_i).P_i]$ only accepts the i-th selection emitted from a principal whose authority level is higher than a_i. Similarly a_1 and a_2 of $\mathsf{ref}^{\mathsf{a}_1,\mathsf{a}_2}_s\langle uv\rangle$ control a low level memory accessibility; a reference can be readable at a_1 and writable at a_2. The principal consists of threads $\{P_i\}_{\mathsf{a}_i}$ running at a_i on parallel. Our presentation needs not to frame a principal by another frame: each

$$\begin{array}{ll}\text{(set)} & \{\mathtt{set}(\mathsf{a})\{P\}\}_{\mathsf{a}'}\longrightarrow\{P\}_{\mathsf{a}}\\ \text{(com-\&)} & \{u[\&_i(\vec{y}_i:\vec{\tau}_i).P_i]\}_{\mathsf{a}}\,|\,\{\overline{u}\mathtt{in}_j\langle\vec{V}_j\rangle\}_{\mathsf{a}'}\longrightarrow\{P_j\{\vec{V}_j/\vec{y}_j\}\}_{\mathsf{a}}\\ \text{(rep-\&)} & \{!\,u[\&^{\mathsf{a}_i}_i(\vec{y}_i:\vec{\tau}_i).P_i]\}_{\mathsf{a}}\,|\,\{\overline{u}\mathtt{in}_j\langle\vec{V}_j\rangle\}_{\mathsf{a}'}\\ & \qquad\longrightarrow\{!\,u[\&_i(\vec{y}_i:\vec{\tau}_i).P_i]\}_{\mathsf{a}}\mid\{P_j\{\vec{V}_j/\vec{y}_j\}\}_{\mathsf{a}\sqcap\mathsf{a}'}\end{array}$$

$$\text{(par)}\ \frac{\mathsf{P}\longrightarrow\mathsf{P}'}{\mathsf{P}\,|\,\mathsf{Q}\longrightarrow\mathsf{P}'\,|\,\mathsf{Q}} \quad \text{(res)}\ \frac{\mathsf{P}\longrightarrow\mathsf{P}'}{(\nu a:\sigma)\mathsf{P}\longrightarrow(\nu a:\sigma)\mathsf{P}'} \qquad \text{(str)}\ \frac{\mathsf{P}\equiv\mathsf{P}'\longrightarrow\mathsf{Q}'\equiv\mathsf{Q}}{\mathsf{P}\longrightarrow\mathsf{Q}}$$

$\{\mathbf{0}\}_{\mathsf{a}}=\mathbf{0}$ $\{P\,|\,Q\}_{\mathsf{a}}=\{P\}_{\mathsf{a}}\,|\,\{Q\}_{\mathsf{a}}$ $\{(\nu a:\sigma)P\}_{\mathsf{a}}\equiv(\nu a:\sigma)\{P\}_{\mathsf{a}}$

The reduction for the reference is defined as (rep-&). Other structure rules are from Figure 7 (by adding the same axioms for principals).

Figure 21: Reduction for Access Control

$$\text{(Set)}\ \frac{\Gamma \vdash P : \Delta, \mathsf{a}}{\Gamma \vdash \mathtt{set}(\mathsf{a})\{P\} : \Delta, \bot}$$

$$\text{(Bra}^{!_{\mathrm{A}}}\text{)}\ \frac{\Gamma \vdash u : [\&_{i\in I}^{\mathsf{a}_i}\tau_i]^{!_{\mathrm{A}}} \quad \tau_i = \Pi[\vec{x}_i : \vec{\sigma}_i]\vec{\tau}'_i \quad \Gamma, \vec{x}_i : \vec{\sigma}_i, \vec{y}_i : \vec{\tau}'_i \vdash P_i : \vec{x}_i : \vec{\sigma}_i, ?\Delta^{-u\vec{x}_i}, \mathsf{a}_i}{\Gamma \vdash\ !u[\&_{i\in I}^{\mathsf{a}_i}(\vec{x}_i : \vec{\sigma}_i, \vec{y}_i : \vec{\tau}'_i).P_i] : u : [\&_{i\in I}\tau_i]_s^{!_{\mathrm{A}}}, \Delta, \sqcup \mathsf{a}_i}$$

$$\text{(Ref)}\ \frac{\Gamma \vdash u : \mathsf{ref}_s^{\mathsf{a}_r, \mathsf{a}_w}\langle\sigma\rangle \quad \Gamma \vdash v : \overline{\sigma}}{\Gamma \vdash \mathsf{ref}_s^{\mathsf{a}_r, \mathsf{a}_w}\langle uv\rangle : \mathsf{ref}_s\langle\sigma\rangle, v : \overline{\sigma}, \mathsf{a}_r \sqcup \mathsf{a}_w}$$

$$\text{(Sel)}\ \frac{\Gamma \vdash u : [\oplus_{l\in I}^{\mathsf{a}_l}\tau_l]^{po} \quad \tau_l = \Pi[\vec{x}_l : \overline{\vec{\sigma}_l}]\vec{\tau}'_l \quad \Gamma \vdash v_i : \sigma_{il}\{\vec{v}/\vec{x}_l\} \quad \Gamma \vdash V_j : \tau'_{jl}\{\vec{v}/\vec{x}_l\}}{\Gamma \vdash \overline{u}\mathtt{in}_l\langle\vec{v}, \vec{V}\rangle\ :\ u : [\oplus_l \tau_l]^{po}, \vec{v} : \vec{\sigma}_l\{\vec{v}/\vec{x}_l\}, \mathsf{a}_l}$$

$$\text{(Principal)}\ \frac{\Gamma \vdash P : \Delta, \mathsf{a}}{\Gamma \vdash \{P\}_{\mathsf{a}} : \Delta}$$

Other rules are from Figure 18 replacing Δ by Δ, a in each sequent in processes. For the principals, we also add (Par), (Res) and (Weak) from Figure 13.

Figure 22: Authority Typing System

process is surrounded only by a *single* frame which makes presentation and reasoning simple. Branching/selection types (as well as reference types) are now annotated by the authority levels; authority levels are always optional, hence omittable; similarly a process type is a pair of interface Δ and the current authority level a; we often omit a when the level of channels in Δ and a coincide together. Other types are not explicitly annotated by authority levels, including value types and unary replications.

Reduction rules for $\lambda\pi_{<}^{\mathsf{LAM}}$ are defined in Figure 21; (bra-&) says that the current authority level is always inserted after interaction [36, 11]; however (set) can freely change the authority level from a to a'. See § 4.3 for examples of reductions. Other reduction rules (such as (com-&) and unary rules) do *not* insert the current authority level of the output to the receiver. This rule simulates Stack Inspection mechanism where once an expression reduces to an outcome (i.e. value), all frames are discarded; in $\lambda\pi_{<}^{\mathsf{LAM}}$, the outcome is carried by a linear/affine output, hence P's level stays as the same in these rules; see the example in § 4.3.

For subtyping, the authority level on the process types is covariant, i.e. $\Delta, \mathsf{a} \leq \Delta', \mathsf{a}'$ with $\Delta \leq \Delta'$ and $\mathsf{a} \sqsubseteq \mathsf{a}'$. For the well-formedness of the pair type, the input authority level should be lower than that of the output authority level (this is opposite to the secrecy level). With respect to $\leq$, the composition $\odot$ is defined as $\Delta_1, \mathsf{a}_2 \odot \Delta_2, \mathsf{a}_2 = \Delta_1 \odot \Delta_2, \mathsf{a}_2 \sqcup \mathsf{a}_2$.

Figure 22 lists the extended typing system. (Set) means $\mathtt{set}(\mathsf{a})\{P\}$ is typable only when the highest level of P is a (which implies P only accesses resources guarded less than a as the principal a), then outside level can be set to the bottom. In ($\mathsf{Bra}^{?_{\mathrm{A}}}$), each body P_i should have less level than that of guarded level a_i. Further, we set the current level to that of u. In (Sel), we record the levels of u (note we do not record the level of carried channels/values). Other rules do not change the authority level.

THEOREM C.3. (subject reduction) *Suppose* $\Gamma \vdash \mathsf{P} : \Delta$. *Then* $\mathsf{P} \twoheadrightarrow \mathsf{P}'$ *implies* $\Gamma \vdash \mathsf{P}' : \Delta$.

PROOF. By analysing if $\Gamma \vdash P : \Delta, \mathsf{a}$ with $u \in \mathsf{dom}(\Gamma) \cap \mathsf{fnv}(P)$, $\Gamma(u) = [\oplus_l^{\mathsf{a}_l}\tau_l]$ with $\mathsf{a} \sqsubseteq \mathsf{a}_l$ implies $\overline{u}\mathtt{in}_l\langle\vec{V}\rangle$ is under $\mathtt{set}(P)\{\mathsf{b}\}$ for some $\mathsf{a}_l \sqsubseteq \mathsf{b}$. Similarly for input branching. □

Next we formulate absence of access errors as follows; define the unary predicate $\mathsf{P} \xrightarrow{Aerr}$ as:

$$\{!b[\&_i^{\mathsf{a}_i}(\vec{y}_i).P_i]\}_{\mathsf{a}} \mid \{\overline{b}\mathtt{in}_k\langle\vec{V}\rangle\}_{\mathsf{a}'} \xrightarrow{Aerr} \text{ if } \mathsf{a}_k \not\sqsubseteq \mathsf{a}'.$$
$$\{\mathsf{ref}^{\mathsf{a}_r, \mathsf{a}_w}\langle xv\rangle\}_{\mathsf{a}} \mid \{\overline{x}\mathtt{in}_1\langle c\rangle\}_{\mathsf{a}'} \xrightarrow{Aerr} \quad \text{if } \mathsf{a}_r \not\sqsubseteq \mathsf{a}'.$$
$$\{\mathsf{ref}^{\mathsf{a}_r, \mathsf{a}_w}\langle xv\rangle\}_{\mathsf{a}} \mid \{\overline{x}\mathtt{in}_2\langle wc\rangle\}_{\mathsf{a}'} \xrightarrow{Aerr} \quad \text{if } \mathsf{a}_w \not\sqsubseteq \mathsf{a}'.$$

$$\frac{\mathsf{P} \xrightarrow{Aerr}}{(\nu a : \sigma)\mathsf{P} \xrightarrow{Aerr}} \qquad \frac{\mathsf{P} \xrightarrow{Aerr} \text{ or } \mathsf{Q} \xrightarrow{Aerr}}{\mathsf{P} \mid \mathsf{Q} \xrightarrow{Aerr}}$$

We interpret "$\{P\}_{\mathsf{a}}$ does do dangerous things", to $\{P\}_{\mathsf{a}_i} \xrightarrow{Aerr}$. That is, the principal cannot access resources guarded by the higher level. The proof is straightforward by the Subject Reduction Theorem.

THEOREM C.4. (Access Safety) *Suppose* $\Gamma \vdash \mathsf{P} : \Delta$. *Then for all* P' *such that* $\mathsf{P} \twoheadrightarrow \mathsf{P}'$, *we have* $\mathsf{P}' \not\xrightarrow{Aerr}$.

Remark. In the history based access control [3], the current rights when a method terminates are lower than or equal to their value before the method call. This mechanism can be defined in $\lambda\pi_{<}^{\mathsf{LAM}}$ by changing (com-&) as well as unary input/output rules as (rep-&). I.e.:

$$\text{(com-\&)} \quad \{u[\&_i(\vec{y}_i : \vec{\tau}_i).P_i]\}_{\mathsf{a}} \mid \{\overline{u}\mathtt{in}_j\langle\vec{V}_j\rangle\}_{\mathsf{a}'} \longrightarrow \{P_j\{\vec{V}_j/\vec{y}_j\}\}_{\mathsf{a}\sqcap\mathsf{a}'}$$

The subject reduction and the access safety theorems can be also proved in this system. We use the stack based version for the application.

Finally we conclude with an example which treats both access control and secrecy.

C.3 Example: Access Control and Secrecy

Suppose we have an authority relation such that $\mathsf{applet} \sqsubseteq \mathsf{alice} \sqsubseteq \mathsf{root}$ and $\mathsf{root} \not\sqsubseteq \mathsf{alice} \not\sqsubseteq \mathsf{applet}$. Assume the following reference

$$\mathsf{ref}^{\mathsf{applet}, \mathsf{root}}\langle t2\rangle$$

This agent can be regarded as a small low level natural number memory readable by everybody and writable by any principal higher than root. A remote server which accepts client's request to access this memory is modelled as:

$$\mathsf{Serv}_1(l, 2) \stackrel{\text{def}}{=} (\nu t)(\mathsf{ref}^{\mathsf{applet}, \mathsf{root}}\langle t2\rangle \mid !l[^{\mathsf{root}}(x : \tau_1).xt \ \&\ ^{\mathsf{alice}}(x : \tau_2).xt \ \&\ ^{\mathsf{applet}}(x : \tau_3).xt]) \quad (14)$$

A channel t is hidden since the address of the memory should be invisible from the clients. $\mathsf{Serv}_1(l, 2)$ contains root capability, hence it should be running at the higher level than root. Note that each branching contains the same body "xt" except the type annotation, τ_i. We will demonstrate how the combination of the authority level and the type annotation τ_i provides the fine-grained access control.

Assume the following two kinds of code, the first is similar with (12) in Example 2.5, while the second writes the data before read it twice in order to use the remote memory (below we write `read` for $\mathtt{in}_1$ and `write` for $\mathtt{in}_2$ to the reference).

Code 1:

$\mathtt{return\ twice\ to}\ e \stackrel{\text{def}}{=} \lambda(t).\overline{t}\mathtt{read}(c)c(x).\overline{t}\mathtt{read}(c)c(x).\overline{e}\langle x + x\rangle$
with type $\Pi(t : \sigma_r)\{t : \sigma_r, e : (\mathtt{nat})^{\uparrow_{\mathrm{L}}}\}$ where $\sigma_r = \mathsf{r}^{\mathtt{applet}}\langle\mathtt{nat}\rangle$

Code 2:

$\mathtt{return\ write\ 3\ then\ twice\ to}\ e$
$\stackrel{\text{def}}{=} \lambda(t).\overline{t}\mathtt{write}\langle 3(c)\rangle c.\overline{t}\mathtt{read}(c)c(x).\overline{t}\mathtt{read}(c)c(x).\overline{e}\langle x + x\rangle$
with $\Pi(t : \sigma_{rw})\{t : \sigma_{rw}, e : (\mathtt{nat})^{\uparrow_{\mathrm{L}}}\}$ where $\sigma_{rw} = \mathsf{rw}^{\mathtt{applet}, \mathtt{root}}\langle\mathtt{nat}\rangle$.

Then the following is the method invocation to perform Code 2 at the remote location l.

$$\begin{array}{l}\texttt{let}\ x = \overline{l}\texttt{in}_i\langle\texttt{write 3 then twice}\rangle\ \texttt{in}\ R\\ \stackrel{\text{def}}{=}\ (\nu e)(\overline{l}\texttt{in}_i\langle\texttt{return write 3 then twice to}\ e\rangle \,|\, e(x).R)\end{array}$$

Assume alice tries to perform this code as follows:

$$\{\texttt{let}\ x = \overline{l}\texttt{in}_2\langle\texttt{write 3 then twice}\rangle\ \texttt{in}\ R\}_{\mathsf{alice}} \quad (15)$$

The server should not accept this code since it violates assessibility of its local memory. Hence he needs to put the protection as:

$$\tau_2\ =\ \exists[e{:}(\texttt{nat})^{\uparrow_{\mathbb{L}}}]\Pi(t{:}\sigma_r)\{t{:}\sigma_r, e{:}(\texttt{nat})^{\uparrow_{\mathbb{L}}}\}$$

Then the composition between $\{\mathsf{Serv}_1(l,2)\}_{\mathsf{root}}$ and alice in (15) is untypable since, for any environment, we have $\sigma_{rw} \not\leq \sigma_r$. Similarly we annotate the first branching by:

$$\tau_1\ =\ \exists[e{:}(\texttt{nat})^{\uparrow_{\mathbb{L}}}]\Pi(t{:}\sigma_{rw})\{t{:}\sigma_{rw}, e{:}(\texttt{nat})^{\uparrow_{\mathbb{L}}}\}$$

and set $\tau_3 = \tau_2$. Then the following principals are all typable

$$\{\texttt{let}\ x = \overline{l}\texttt{in}_1\langle\texttt{write 3 then twice}\rangle\ \texttt{in}\ R\}_{\mathsf{root}} \quad (16)$$
$$\{\texttt{let}\ x = \overline{l}\texttt{in}_2\langle\texttt{twice}\rangle\ \texttt{in}\ R\}_{\mathsf{alice}} \quad (17)$$
$$\{\texttt{let}\ x = \overline{l}\texttt{in}_3\langle\texttt{twice}\rangle\ \texttt{in}\ R\}_{\mathsf{applet}} \quad (18)$$

while if we replace root in (16) by either alice or applet, then it is untypable (by the guarded levels of the branching channel type of l). For this moment, carol and applet have the same capability to resource t. For more fine-grained control, root may wish to allow alice to write a file by raising its privilege to root, then return the final result as the original principal, alice. This is specified by raising the second branch in $\mathsf{Serv}(l,2)$ as

$$\mathsf{Serv}_2(l,2) \stackrel{\text{def}}{=} !l[...\&^{\mathsf{carol}}(x{:}\tau_1).(\texttt{set}(\mathsf{root})\{xt\})\&...]$$

Note that τ_2 is also changed to τ_1. alice can now perform (15) with the permission given by root. Then we have:

$$\begin{array}{l}\{\mathsf{Serv}_2(l,2)\}_{\mathsf{root}} \,|\, \{\texttt{let}\ x = \overline{l}\texttt{inr}\langle\texttt{write 3 then twice}\rangle\ \texttt{in}\ R\}_{\mathsf{alice}}\\ \twoheadrightarrow \{!l[..]\}_{\mathsf{root}} \,|\, \{\mathsf{ref}^{\mathsf{applet},\mathsf{root}}\langle t2\rangle\}_{\mathsf{root}}\\ \qquad\qquad |\, \{\texttt{set}(\mathsf{root})\{(\texttt{return}...\ e)\ t\}\}_{\mathsf{alice}} \,|\, \{e(x).R\}_{\mathsf{alice}})\\ \longrightarrow \{!l[..]\}_{\mathsf{root}} \,|\, \{\mathsf{ref}^{\mathsf{applet},\mathsf{root}}\langle t2\rangle\}_{\mathsf{root}}\\ \qquad\qquad |\, \{(\texttt{return}...\ e)\ t\}_{\mathsf{root}} \,|\, \{e(x).R\}_{\mathsf{alice}})\\ \twoheadrightarrow \{\mathsf{Serv}_2(l,3)\}_{\mathsf{root}} \,|\, (\nu e)(\{\overline{e}\langle 6\rangle\}_{\mathsf{root}} \,|\, \{e(x).R\}_{\mathsf{alice}})\\ \twoheadrightarrow \{\mathsf{Serv}_2(l,3)\}_{\mathsf{root}} \,|\, |\, \{R\{6/x\}\}_{\mathsf{alice}}\end{array}$$

Now let us consider the following example, which shows information flow analysis is usually more difficult to predict [3, 11]. We change from the above server to:

$$\begin{array}{rl}\mathsf{Serv}_3(l,2) & \stackrel{\text{def}}{=}\ (\nu t)(\mathsf{ref}^{\mathsf{root},\mathsf{root}}\langle t2^{\mathbb{H}}\rangle\\ & \quad |\, !l[^{\mathsf{root}}(x{:}\tau_1).xt\ \&\ ^{\mathsf{alice}}(x{:}\tau_1).\texttt{set}(\mathsf{root})\{xt\}] \quad (19)\end{array}$$

Actually it seems this server is more secure than $\mathsf{Serv}_1(l,2)$ and $\mathsf{Serv}_2(l,2)$, because the memory (reference) is only read/writable by root, and applet cannot read access it. Both (16) and (15) are composable with $\mathsf{Serv}_3(l,2)$ as they are with $\mathsf{Serv}_2(l,2)$ by branching subtyping. Now assume the data 3 in (16) is secret ($\mathbb{H}$), hence root wishes not to leak it to applet. He believes to perform (16) is safe since the memory is protected by root. However we need to track more computational flows to guarantee the secrecy; suppose the following alice and it performs this remote call to Serv_3 after root in (16) interacts with $\mathsf{Serv}_3(l,2)$.

$$\{\mathsf{Serv}_3(l,3^{\mathbb{H}})\}_{\mathsf{root}} \,|\, \{\texttt{let}\ x = \overline{l}\texttt{in}_2\langle\texttt{twice}\rangle\ \texttt{in}\ \mathsf{ref}^{\mathsf{applet},\mathsf{carol}}\langle ax^{\mathbb{H}}\rangle\}_{\mathsf{alice}}$$

$$\texttt{set}(\mathsf{a})\{e\} \longrightarrow \{e\}_{\mathsf{a}} \qquad \{V\}_{\mathsf{a}} \longrightarrow V \qquad \{\texttt{skip}\}_{\mathsf{a}} \longrightarrow \texttt{skip}$$

$$\frac{(e,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(e',\sigma',H',\mathtt{CT}')}{(\{e\}_{\mathsf{a}},\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(\{e'\}_{\mathsf{a}},\sigma',H',\mathtt{CT}')}$$

$$\frac{\begin{array}{l}l_1[\mathtt{P}\,|\,\mathtt{P}_1,\sigma_1,H_1,\mathtt{CT}_1]\,|\,l_2[\mathtt{P}_2,\sigma_2,H_2,\mathtt{CT}_2]\\ \qquad\longrightarrow l_1[\mathtt{P}_1,\sigma_1,H_1,\mathtt{CT}_1]\,|\,l_2[\mathtt{P}\,|\,\mathtt{P}_2,\sigma_2,H_2,\mathtt{CT}_2]\end{array}}{\begin{array}{l}l_1[\{\mathtt{P}\}_{\mathsf{a}}\,|\,\mathtt{P}_1,\sigma_1,H_1,\mathtt{CT}_1]\,|\,l_2[\mathtt{P}_2,\sigma_2,H_2,\mathtt{CT}_2]\\ \qquad\longrightarrow l_1[P_1,\sigma_1,H_1,\mathtt{CT}_1]\,|\,l_2[\{\mathtt{P}\}_{\mathsf{a}}\,|\,\mathtt{P}_2,\sigma_2,H_2,\mathtt{CT}_2]\end{array}}$$

[Structural Rules]

$\{\mathbf{0}\}_{\mathsf{a}} = \mathbf{0}$, $\{\mathtt{P}\,|\,\mathtt{Q}\}_{\mathsf{a}} \equiv \{\mathtt{P}\}_{\mathsf{a}}\,|\,\{\mathtt{Q}\}_{\mathsf{a}}$, $\{(\nu u)\mathtt{P}\}_{\mathsf{a}} \equiv (\nu u)\{\mathtt{P}\}_{\mathsf{a}}$.

Figure 24: Reduction for a Language with Authority

Then the high level data 3 originally from root is leaked to applet via port *a without violating accessibility*.

$$\{\overline{a}\texttt{read}(c)c(x).\texttt{if}\ x^{\mathbb{H}}\ \texttt{then}\ \overline{y}^{\mathbb{L}}\ \texttt{else}\ \overline{z}^{\mathbb{L}}\}_{\mathsf{applet}}$$

By using the typing system for secrecy in §. C.1, we can statically track this possible leakage. Obviously $\texttt{if}\ x^{\mathbb{H}}\ \texttt{then}\ \overline{y}^{\mathbb{L}}\ \texttt{then}\ \overline{z}^{\mathbb{L}} \stackrel{\text{def}}{=} x(w).(\nu c)(wc\,|\,c[.\overline{y}\&.\overline{z}])$ is untypable by $(\mathsf{Bra}^{\uparrow_{\mathbb{L}}})$ in the secrecy typing system in § 4.2 and § C.1 (more precisely we also need to map 3 to $\lambda x.\overline{x}\texttt{in}_3$ as done in Example B.7).

D. TYPE-BASED SECURITY FOR A LANGUAGE WITH CODE MOBILITY

This section lists the definitions omitted from § 5 in the main section.

D.1 Operational Semantics

The full definition of the operational semantics of the language is defined in Figure 23. As explained in the main section, the following runtime syntax is used to define the operational semantics.

$$\begin{array}{rcl}e & ::= & \ldots\ \overline{o}.\mathtt{m}(\vec{V})\ \texttt{to}\ c\\ & \mid & \texttt{wait}\ z\ \texttt{at}\ c\ \texttt{in}\ e\ \mid\ e;\texttt{return}\ e'\ \texttt{to}\ c\end{array}$$

Then [Local Rules] come from [20], which use a standard call-by-value evaluation strategy. In these rules, we omit σ, H and/or CT if unchanged between the reduction. Note that newvar creates a new memory whose imperative variable is bound by the ν-operator. The rules from [NewC] to [Deserialization] are explained in §.5.2 in the main section. Others are standard contextual rules come from the π-calculus.

D.2 Operational Semantics for Access Control

The operational semantics for the access control is defined in Figure 24. They are already explained in § 5.4 in the main section, except the contextual rules and the structure rules. Then the authority error is defined in Figure 25, which is also explained in § 5.4. The omitted rules are the third and forth read/write error rules for references, which correspond to those for the reference agents in $\lambda\pi^{\mathsf{LAM}}_{<}$ defined in § C.2.

D.3 Typing

Hereafter we shall consider the language with secrecy and authority.

[Local Rules]

$\mathtt{succ}(n) \longrightarrow n+1 \qquad \mathtt{pred}(n) \longrightarrow n-1$

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

$(\mathtt{if}\ V\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2,\sigma) \longrightarrow (e_1,\sigma) \qquad (V = 0)$

$(\mathtt{if}\ V\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2,\sigma) \longrightarrow (e_2,\sigma) \qquad (V \neq 0)$

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

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

$$\frac{(e_1,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(e_1',\sigma',H',\mathtt{CT}')}{(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(\mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2,\sigma',H',\mathtt{CT}')} \quad (\vec{u} \notin \mathsf{fnv}(e_2))$$

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

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

$(\mathtt{newvar}\ x \mapsto V\ \mathtt{in}\ e,\sigma) \longrightarrow (\nu x)(e,\sigma \cup [x \mapsto V]) \qquad (x \notin \mathsf{dom}(\sigma))$

$$\frac{(e_1,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(e_1',\sigma',H',\mathtt{CT}')}{(e_1;e_2,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(e_1';e_2,\sigma',H',\mathtt{CT}')} \quad (\vec{u} \notin \mathsf{fnv}(e_2))$$

$$\frac{(e_1,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(\sigma',H',\mathtt{CT}')}{(e_1;e_2,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(e_2,\sigma',H',\mathtt{CT}')} \quad (\vec{u} \notin \mathsf{fnv}(e_2))$$

$\mathtt{return}\ V \longrightarrow V$

$$\frac{(e,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(e',\sigma',H',\mathtt{CT}')}{(\mathtt{return}\,e,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(\mathtt{return}\,e',\sigma',H',\mathtt{CT}')} \quad (\vec{u} \notin \mathsf{fnv}(e_2))$$

[Threads]

$$\frac{(\nu\vec{u})(\mathtt{P},\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u}')(\mathtt{P}',\sigma',H',\mathtt{CT}')}{(\nu u\vec{u})(\mathtt{P},\sigma,H,\mathtt{CT}) \longrightarrow (\nu u\vec{u}')(\mathtt{P}',\sigma',H',\mathtt{CT}')}$$

$$\frac{(\mathtt{P}_1,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(\mathtt{P}_1',\sigma',H',\mathtt{CT}')}{(\mathtt{P}_1 \,|\, \mathtt{P}_2,\sigma,H,\mathtt{CT}) \longrightarrow (\nu\vec{u})(\mathtt{P}_1' \,|\, \mathtt{P}_2,\sigma',H',\mathtt{CT}')} \quad (\vec{u} \notin \mathsf{fnv}(\mathtt{P}_2))$$

[NewC]

$$\frac{\mathtt{C} \mapsto\ \mathtt{class}\ \mathtt{C}\ \{\vec{\mathtt{T}}\,\vec{y}\ ;\ \vec{\mathtt{M}}\} \in \mathtt{CT}}{(\mathtt{let}\ x = \mathtt{new}\ \mathtt{C}(\vec{V})\ \mathtt{in}\ e_2,\sigma,H,\mathtt{CT}) \longrightarrow (\nu o\vec{y})(e_2\{o/x\},\sigma \cup [\vec{y} \mapsto \vec{V}],H \cup [o \mapsto \vec{\mathtt{M}}],\mathtt{CT})} \quad (o,\vec{x}\ \text{fresh})$$

[Serialization]

$$\frac{\mathtt{CT} \subset (\mathtt{CT}' \setminus \mathtt{FCT}) \quad \mathsf{dom}(\mathtt{CT}) = \mathsf{fc}(e_1) \setminus \mathsf{dom}(\mathtt{FCT})}{(\mathtt{let}\ x = \mathtt{serialize}(e_1)\ \mathtt{in}\ e_2,\sigma,H,\mathtt{CT}') \longrightarrow (e_2\{\ulcorner e_1\ \mathtt{with}\ \mathtt{CT}\urcorner/x\},\sigma,H,\mathtt{CT}')}$$

[Deserialization]

$$(\mathtt{let}\ x = \mathtt{deserialize}(\ulcorner e_1\ \mathtt{with}\ \mathtt{CT}\urcorner)\ \mathtt{in}\ e_2,\sigma,H,\mathtt{CT}') \longrightarrow (\mathtt{let}\ x = e_1\,\mathtt{in}\ e_2,\sigma,H,\mathtt{CT}' \cup \mathtt{CT})$$

[Method Go]

$\mathtt{let}\ x = o.\mathtt{m}(\vec{V})\ \mathtt{in}\ e \longrightarrow (\nu c)(o.\mathtt{m}(\vec{V})\ \mathtt{to}\ c \mid \mathtt{wait}\ x\ \mathtt{at}\ c\ \mathtt{in}\ e) \quad (c\ \text{fresh})$

[Method Invocation]

$$\frac{o \mapsto \tilde{\mathtt{M}} \in H \qquad \mathtt{M}_i = \mathtt{T}\ \mathtt{m}(\vec{\mathtt{T}}\,\vec{x})\{e;\}}{(o.\mathtt{m}(\vec{V})\ \mathtt{to}\ c,\sigma,H) \longrightarrow (e\{\vec{V}/\vec{x}\}\{\mathtt{return}\ e\ \mathtt{to}\ c/\ \mathtt{return}\,e\},\sigma,H)}$$

[Return]

$(\mathtt{return}\ V\ \mathtt{to}\ c \mid \mathtt{wait}\ x\ \mathtt{at}\ c\ \mathtt{in}\ e) \longrightarrow e\{V/x\}$

[Method Migration]

$l_1[o.\mathtt{m}(\vec{V})\ \mathtt{to}\ c \,|\, \mathtt{P}_1,\sigma_1,H_1,\mathtt{CT}_1] \,|\, l_2[\mathtt{P}_2,\sigma_2,H_2,\mathtt{CT}_2]$
$\longrightarrow l_1[\mathtt{P}_1,\sigma_1,H_1,\mathtt{CT}_1] \,|\, l_2[o.\mathtt{m}(\vec{V})\ \mathtt{to}\ c \,|\, \mathtt{P}_2,\sigma_2,H_2,\mathtt{CT}_2] \quad (o \in \mathsf{dom}(H_2))$

[Remote Return]

$l_1[\mathtt{return}\ V\ \mathtt{to}\ c \,|\, \mathtt{P}_1,\sigma_1,H_1,\mathtt{CT}_1] \,|\, l_2[\mathtt{P}_2,\sigma_2,H_2,\mathtt{CT}_2]$
$\longrightarrow l_1[\mathtt{P}_1,\sigma_1,H_1,\mathtt{CT}_1] \,|\, l_2[\mathtt{return}\ V\ \mathtt{to}\ c \,|\, \mathtt{P}_2,\sigma_2,H_2,\mathtt{CT}_2] \quad (c \in \mathsf{fn}(\mathtt{P}_2))$

[Networks]

$$\frac{\mathtt{H} \longrightarrow \mathtt{H}'}{l[\mathtt{H}] \longrightarrow l[\mathtt{H}']} \qquad \frac{\mathtt{N} \longrightarrow \mathtt{N}'}{\mathtt{N} \,|\, \mathtt{N}_0 \longrightarrow \mathtt{N}' \,|\, \mathtt{N}_0}$$

$$\frac{\mathtt{N} \longrightarrow \mathtt{N}'}{(\nu u)\mathtt{N} \longrightarrow (\nu u)\mathtt{N}'} \qquad \frac{\mathtt{N} \equiv \mathtt{N}_0 \longrightarrow \mathtt{N}_0' \equiv \mathtt{N}'}{\mathtt{N} \longrightarrow \mathtt{N}'}$$

$l[(\nu u\vec{u})(\mathtt{P},\sigma,H,\mathtt{CL})] \equiv (\nu u)l[(\nu\vec{u})(\mathtt{P},\sigma,H,\mathtt{CL})]$

The reduction rule for `return` e `to` c is defined as `return` e. Other structure rules for expressions, principals and hosts are defined as in Figure 7.

Figure 23: Reduction

$(\{\texttt{let}\ x = \texttt{new}\ \texttt{C}(\vec{V})\ \texttt{in}\ e_2\}_{\mathsf{a}}, \sigma, H, \texttt{CT}) \xrightarrow{Aerr}$
with $\texttt{C} \mapsto \texttt{class C}\ \{\mathsf{a}'\}\ \{\vec{\texttt{T}}\ \vec{x}\ ;\ \vec{\texttt{M}}\} \in \texttt{CT}$ and $\mathsf{a}' \not\sqsubseteq \mathsf{a}$

$(\{o.\texttt{m}(\vec{V})\ \texttt{to}\ c\}_{\mathsf{a}}, \sigma, H) \xrightarrow{Aerr}$
with $o \mapsto \tilde{\texttt{M}} \in H$, $\texttt{M}_i = \texttt{T m}\{\mathsf{a}'\}(\vec{\texttt{T}}\ \vec{x})$ and $\mathsf{a}' \not\sqsubseteq \mathsf{a}$

$(\{\texttt{let}\ x = !y\ \texttt{in}\ e_2\}_{\mathsf{a}}, \sigma \cup [x^{\mathsf{a}_r,\mathsf{a}_w} \mapsto V]) \xrightarrow{Aerr}$ with $\mathsf{a}_r \not\sqsubseteq \mathsf{a}$

$(\{x := V\}_{\mathsf{a}}, \sigma \cup [x^{\mathsf{a}_r,\mathsf{a}_w} \mapsto V])$ with $\mathsf{a}_w \not\sqsubseteq \mathsf{a}$

Other rules are the contextual rules.

Figure 25: Authority Error

Types. In the following, we first define the types which are used in the type inference.

$$
\begin{array}{rrcl}
\text{(term)} & \rho & ::= & \texttt{T}_v \mid \texttt{T}_c \quad \text{(value)}\ \texttt{T}_v ::= \texttt{T} \mid \llcorner \texttt{T} \lrcorner_s \\
\text{(total)} & \texttt{T} & ::= & \texttt{C} \mid \texttt{nat}_s \mid \texttt{void} \mid \exists[\vec{x}{:}\vec{\texttt{C}}]\ulcorner \texttt{T}_v; \Delta; \vec{\texttt{D}} \urcorner \\
\text{(deserialize)} & \texttt{T}_d & ::= & \texttt{T}, \Delta \\
\text{(reference)} & \texttt{T}_r & ::= & \texttt{ref}_s^{\mathsf{a}_r,\mathsf{a}_w}(\texttt{T}) \mid \texttt{r}_s^{\mathsf{a}_r}(\texttt{T}) \mid \texttt{w}_s^{\mathsf{a}_w}(\texttt{T}) \\
\text{(command)} & \texttt{T}_c & ::= & \texttt{cmd} \mid \llcorner \texttt{cmd} \lrcorner_s \\
\text{(effect)} & \Delta_e & ::= & \emptyset \mid \Delta_e, o{:}\texttt{C} \mid \Delta_e, o.\texttt{m}{:}\texttt{C} \mid \Delta_e, x{:}\texttt{T}_r \mid \Delta_e, c{:}\sigma \\
\text{(interface)} & \Delta & ::= & \Delta_e, \mathsf{a} \\
\text{(chan)} & \sigma_r & ::= & (\texttt{T})_s^p \mid \langle \sigma_r, \overline{\sigma_r} \rangle \quad \mathsf{md}(\sigma_r) \in \{\uparrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}, \downarrow_{\mathrm{L}}, \downarrow_{\mathrm{A}}\} \\
\text{(env)} & E & ::= & \emptyset \mid E, c{:}\sigma \mid E, o{:}\texttt{C} \mid E, x{:}\tau \in \{\texttt{T}, \texttt{T}_r, \texttt{T}_d\}
\end{array}
$$

Term types contain command types and value types. Value types are either total T or partial $\llcorner \texttt{T} \lrcorner_s$. $\llcorner \texttt{T} \lrcorner_s$ denotes possibly non-terminating value with secrecy level s. This lifting operator comes from [2, 20]. Total types are explained in the main section. Note by the well-formed rules given later, Δ in the thunk type and the deserialisation type only contains the mapping from object id (plus its method) to the class name. The deserialized type is a pair of value type and effect; this is only used for the deserialization. Reference types are annotated by security level (s), and the read (a_r) and write (a_w) authority levels. `r` means read only and `w` means write only; they correspond to $\mathsf{r}\langle\sigma\rangle$ and $\mathsf{w}\langle\sigma\rangle$ types in $\lambda\pi^{\mathsf{LAM}}_{<:}$. As in the value type, the command type `cmd` denotes terminating command, and $\llcorner \texttt{cmd} \lrcorner_s$ denotes possibly non-terminating command with secrecy level s. The channel types are used to carry the return value. If the carried value is non-terminating, then it has a $\uparrow_{\mathrm{A}}$-mode, else it has a $\uparrow_{\mathrm{L}}$-mode. Then as explained in the main section, the interface type is a pair of the authority level a and a mapping which represents the effect to the environment. The effect is used to calculate the secrecy level of the expression.

Class Types. We define a couple of definitions which are needed for the type inference. We use a fixed signature for class tables, denote by *CSig*, which is a mapping from a class name to a class type. A class type is inductively defined as follows.

$$
\begin{array}{rrcl}
\text{(method)} & \texttt{T}_{\texttt{m}} & ::= & \{\mathsf{a}\}\vec{\texttt{T}} \Rightarrow \texttt{T}_v \mid \{\mathsf{a}\}\vec{\texttt{T}} \overset{s}{\Rightarrow} \llcorner \texttt{T} \lrcorner_s \\
\text{(object)} & \texttt{T}_o & ::= & [\texttt{T}_{\texttt{m}_1} \& .. \& \texttt{T}_{\texttt{m}_n}] \\
\text{(class)} & \texttt{T}_{\texttt{C}} & ::= & \{\mathsf{a}\}\vec{\texttt{T}} \Rightarrow \texttt{T}_o
\end{array}
$$

A method has either total arrow or partial arrow types. $\overset{s}{\Rightarrow}$ means the function which gives an effect to the environment at the level s. The object types takes a form of a record type. The class declaration has a total type since the `new` action always succeed (hence it terminates), and every class consists no free reference variables. The given class signature is assumed to satisfy the condition for every class name C appears anywhere in *CSig*, we have $\texttt{C} \in \mathsf{dom}(\mathit{CSig})$; there is no cycles in the occurrences of names in thunk types. Then we define two look up functions [24] as follows:

If $\mathit{Csig}(\texttt{C}) = \{\mathsf{a}\}\vec{\texttt{T}} \Rightarrow [\texttt{T}_{\texttt{m}_1} \& .. \& \texttt{T}_{\texttt{m}_n}]$, then $\mathit{inst}(\texttt{C}) = \{\mathsf{a}\}\vec{\texttt{T}}$ and $\mathit{mtype}(\texttt{C}, \texttt{m}_i) = \texttt{T}_{\texttt{m}_i}$.

Here $\mathit{inst}(\texttt{C})$ returns types of instance variables of the class C while $\mathit{mtype}(\texttt{C}, \texttt{m}_i)$ returns the type of the method $\texttt{m}_i$ in C in *Csig*.

We also define a function $\mathsf{alev}(\Delta)$ which returns the authority level of Δ (i.e. $\mathsf{alev}(\Delta_e, \mathsf{a}) = \mathsf{a}$) and a function $\mathsf{sec}(\tau)$ which returns the outmost secrecy level.

Tampering Level. We first define the tampering level of a type τ following [20]. We omit the authority levels from types since they are unrelated.

DEFINITION D.1. (tampering level) The *tamperting level* of τ, denoted by $\mathsf{tamper}(\tau)$, is inductively given by:

- $\mathsf{tamper}(\texttt{nat}_s) = \mathsf{tamper}(\llcorner \tau \lrcorner_s) = s$
- $\mathsf{tamper}(\texttt{void}) = \mathsf{tamper}(\texttt{cmd}) = \top$
- $\mathsf{tamper}(\texttt{C}) = \mathsf{tamper}(\texttt{T}_o) \qquad (\mathit{Csig}(\texttt{C}) = \vec{\texttt{T}} \Rightarrow \texttt{T}_o)$
- $\mathsf{tamper}(\exists[\vec{x}{:}\vec{\texttt{C}}]\ulcorner \texttt{T}_v; \Delta; \vec{\texttt{D}} \urcorner) = \mathsf{tamper}((\texttt{T}_v, \Delta)) = \mathsf{tamper}(\texttt{T}_v) \sqcap \mathsf{tamper}(\Delta)$
- $\begin{cases} \mathsf{tamper}(\texttt{ref}_s(\texttt{T})) = \mathsf{tamper}^{\mathcal{E}}(\texttt{T}) \sqcap \mathsf{tamper}(\texttt{T}) \\ \mathsf{tamper}(\texttt{r}_s(\texttt{T})) = \mathsf{tamper}(\texttt{T}) \\ \mathsf{tamper}(\texttt{w}_s(\texttt{T})) = \mathsf{tamper}^{\mathcal{E}}(\texttt{T}) \end{cases}$
- $\begin{cases} \mathsf{tamper}((\texttt{T})_s^p) = s & p = \uparrow_{\mathrm{A}} \\ \mathsf{tamper}((\texttt{T})_s^p) = \mathsf{tamper}(\texttt{T}) & p = \uparrow_{\mathrm{L}} \\ \mathsf{tamper}((\texttt{T})_s^p) = \mathsf{tamper}^{\mathcal{E}}(\texttt{T}) & p = \{\downarrow_{\mathrm{A}}, \downarrow_{\mathrm{L}}\} \\ \mathsf{tamper}(\langle \sigma_r, \overline{\sigma_r} \rangle) = \top \end{cases}$
- $\mathsf{tamper}([\texttt{T}_{\texttt{m}_1} \& .. \& \texttt{T}_{\texttt{m}_n}]) = \sqcap_i \mathsf{tamper}(\texttt{T}_{\texttt{m}_i})$
- $\mathsf{tamper}(\vec{\texttt{T}} \Rightarrow \tau) = \mathsf{tamper}(\vec{\texttt{T}} \overset{s}{\Rightarrow} \tau) = \sqcap_i \mathsf{tamper}^{\mathcal{E}}(\texttt{T}_i) \sqcap \mathsf{tamper}(\tau)$
- $\begin{cases} \mathsf{tamper}(\emptyset) = \top \\ \mathsf{tamper}(\Delta, o{:}\texttt{C}) = \mathsf{tamper}(\Delta) \sqcap \mathsf{tamper}^{\mathcal{E}}(\texttt{C}) \\ \mathsf{tamper}(\Delta, o.\texttt{m}{:}\texttt{C}) = \mathsf{tamper}(\Delta) \sqcap \mathsf{tamper}^{\mathcal{E}}(\texttt{T}_{\texttt{m}}) \quad (\mathit{mtype}(\texttt{C}, \texttt{m}) = \texttt{T}_{\texttt{m}}) \\ \mathsf{tamper}(\Delta, x{:}\texttt{T}_r) = \mathsf{tamper}(\Delta) \sqcap \mathsf{tamper}^{\mathcal{E}}(\texttt{T}_r) \\ \mathsf{tamper}(\Delta, c{:}(\texttt{T})_s^p) = \mathsf{tamper}(\Delta) \sqcap \mathsf{tamper}((\texttt{T})_s^p) \end{cases}$

where $\mathsf{tamper}^{\mathcal{E}}(\ )$ is defined as:

- $\mathsf{tamper}^{\mathcal{E}}(\texttt{nat}_s) = \mathsf{tamper}^{\mathcal{E}}(\texttt{void}) = \top$
- $\mathsf{tamper}^{\mathcal{E}}(\exists[\vec{x}{:}\vec{\texttt{C}}]\ulcorner \texttt{T}_v; \Delta; \vec{\texttt{D}} \urcorner) = \top$
- $\begin{cases} \mathsf{tamper}^{\mathcal{E}}(\texttt{ref}_s(\texttt{T})) = \mathsf{tamper}^{\mathcal{E}}(\texttt{w}_s(\texttt{T})) = s \\ \mathsf{tamper}^{\mathcal{E}}(\texttt{r}_s(\texttt{T})) = \mathsf{tamper}^{\mathcal{E}}(\texttt{T}) \end{cases}$
- $\mathsf{tamper}^{\mathcal{E}}([\texttt{T}_{\texttt{m}_1} \& .. \& \texttt{T}_{\texttt{m}_n}]) = \sqcap_i \mathsf{tamper}^{\mathcal{E}}(\texttt{T}_{\texttt{m}_i})$
- $\mathsf{tamper}^{\mathcal{E}}(\emptyset) = \mathsf{tamper}^{\mathcal{E}}(\vec{\texttt{T}} \Rightarrow \tau) = \top$
- $\mathsf{tamper}^{\mathcal{E}}(\vec{\texttt{T}} \overset{s}{\Rightarrow} \tau) = s$

Well-Formed Types and Subtyping. From the given signature, we generate a well-formed types and the subtyping relation. They are defined in Figure 26. The condition of secrecy levels on immutable method type and the reference follows [20, § 7]. The subtyping relation is extension from [20, § 7]. The subsumption of the authority level follows § 4.3 in the main section.

Typing. The main typing judgements are:

$$E \vdash e : \rho, \Delta \qquad \text{and} \qquad E \vdash \texttt{N} : \texttt{net}, \Delta$$

A partial commutative operator $\odot$ on types is defined as:

$$\text{(e-nil)}\quad \emptyset \vdash \texttt{Env} \qquad \text{(e-c)}\ \frac{E \vdash \texttt{C} : \texttt{tp} \quad o \notin \mathsf{dom}(E)}{E, o{:}\texttt{C} \vdash \texttt{Env}} \qquad \text{(e-val)}\ \frac{x \notin \mathsf{dom}(E) \quad E \vdash \tau : \texttt{tp} \quad \tau \in \{\texttt{T}, \texttt{T}_r, \texttt{T}_d, \texttt{C}\}}{E, x{:}\tau \vdash \texttt{Env}} \qquad \text{(e-pack)}\ \frac{E \vdash \exists[\vec{x}{:}\vec{\texttt{C}}]\ulcorner \texttt{T}; \Delta; \vec{\texttt{D}}\urcorner : \texttt{tp} \quad \vec{x}, y \notin \mathsf{dom}(E)}{E, \{\vec{x}{:}\vec{\texttt{C}}, y{:}(\texttt{T}, \Delta)\} \vdash \texttt{Env}}$$

$$\text{(t-base)}\ \frac{E \vdash \texttt{Env}}{E \vdash \texttt{void}, \texttt{nat}_s, \texttt{cmd} : \texttt{tp}} \qquad \text{(t-thunk)}\ \frac{E, \vec{x}{:}\vec{\texttt{C}} \vdash \texttt{T}, \Delta : \texttt{tp} \quad \texttt{D}_i \in \mathsf{dom}(\mathit{CSig}) \quad \vec{x} = \mathsf{dom}(\Delta)}{E \vdash \exists[\vec{x}{:}\vec{\texttt{C}}]\ulcorner \texttt{T}; \Delta; \vec{\texttt{D}}\urcorner : \texttt{tp}} \qquad \text{(t-ref)}\ \frac{E \vdash \texttt{T} : \texttt{tp} \quad s \sqsubseteq \mathsf{tamper}(\texttt{ref}_s^{\mathtt{a}_r, \mathtt{a}_w}(\texttt{T}))}{E \vdash \texttt{ref}_s^{\mathtt{a}_r, \mathtt{a}_w}(\texttt{T}) : \texttt{tp}} \qquad \text{(t-lift)}\ \frac{E \vdash \tau : \texttt{tp} \quad \tau = \texttt{T} \text{ or } \tau = \texttt{cmd}}{E \vdash \llcorner \tau \lrcorner_s : \texttt{tp}}$$

$$\text{(t-m)}\ \frac{E \vdash \texttt{T} : \texttt{tp} \quad E \vdash \vec{\texttt{T}} : \texttt{tp}}{E \vdash \{\mathtt{a}\}\vec{\texttt{T}} \Rightarrow \texttt{T}_v : \texttt{tp}} \qquad \text{(t-mp)}\ \frac{E \vdash \llcorner \texttt{T} \lrcorner_{s'} : \texttt{tp} \quad E \vdash \vec{\texttt{T}} : \texttt{tp} \quad s \sqsubseteq \mathsf{tamper}(\vec{\texttt{T}} \overset{s}{\Rightarrow} \llcorner \texttt{T} \lrcorner_{s'})}{E \vdash \{\mathtt{a}\}\vec{\texttt{T}} \overset{s}{\Rightarrow} \llcorner \texttt{T} \lrcorner_{s'} : \texttt{tp}}$$

$$\text{(t-obj-total)}\ \frac{E \vdash \texttt{T}_{\mathtt{m}i} : \texttt{tp} \quad \texttt{T}_{\mathtt{m}i} = \{\mathtt{a}_i\}\vec{\texttt{T}}_i \Rightarrow \vec{\texttt{T}}_{vi} \quad \mathsf{sec}(\vec{\texttt{T}}_{vi}) = \mathsf{sec}(\vec{\texttt{T}}_{vj})}{E \vdash [\texttt{T}_{\mathtt{m}_1} \& \ldots \& \texttt{T}_{\mathtt{m}_n}] : \texttt{tp}} \qquad \text{(t-obj-total)}\ \frac{E \vdash \texttt{T}_{\mathtt{m}i} : \texttt{tp} \quad \texttt{T}_{\mathtt{m}i} = \{\mathtt{a}_i\}\vec{\texttt{T}}_i \overset{s}{\Rightarrow} \vec{\texttt{T}}_{vi} \quad \mathsf{sec}(\vec{\texttt{T}}_{vi}) = \mathsf{sec}(\vec{\texttt{T}}_{vj})}{E \vdash [\texttt{T}_{\mathtt{m}_1} \& \ldots \& \texttt{T}_{\mathtt{m}_n}] : \texttt{tp}}$$

$$\text{(t-class)}\ \frac{E \vdash \texttt{T}_o : \texttt{tp} \quad E \vdash \vec{\texttt{T}} : \texttt{tp}}{E \vdash \{\mathtt{a}\}\vec{\texttt{T}} \Rightarrow \texttt{T}_o : \texttt{tp}}$$

$$\text{(t-pro-b)}\ \frac{E \vdash \texttt{Env}}{E \vdash \emptyset, \mathtt{a} : \texttt{tp}} \qquad \text{(t-pro-o)}\ \frac{E \vdash \Delta \quad E \vdash o : \texttt{C}}{E \vdash \Delta \odot o : \texttt{C} : \texttt{tp}} \qquad \text{(t-pro-m)}\ \frac{E \vdash \Delta \quad E \vdash o : \texttt{C}}{E \vdash \Delta \odot o.\mathtt{m} : \texttt{C} : \texttt{tp}} \qquad \text{(t-class)}\ \frac{E \vdash \tau : \texttt{tp} \quad \mathit{Csig}(\texttt{C}) = \tau}{E \vdash \texttt{C} : \texttt{tp}} \qquad \text{(t-chan)}\ \frac{E \vdash \texttt{T} : \texttt{tp}}{E \vdash (\texttt{T})_s^{\uparrow} : \texttt{tp}}$$

The rules for $\mathtt{r}_s(\texttt{T}), \mathtt{w}_s(\texttt{T})$ are defined as (t-ref). $(\texttt{T})_s^{\downarrow}, \langle(\texttt{T})_s^{\downarrow}, (\texttt{T})_s^{\uparrow}\rangle$ are defined as (t-chan).

$$\text{(s-base)}\ \frac{E \vdash \alpha : \texttt{tp}}{E \vdash \alpha < \alpha} \qquad \text{(s-nat)}\ \frac{s \sqsubseteq s' \quad E \vdash \texttt{Env}}{E \vdash \texttt{nat}_s \leq \texttt{nat}_{s'}} \qquad \text{(s-lift)}\ \frac{s_1 \sqsubseteq s_2 \quad E \vdash \llcorner \tau_1 \lrcorner_s : \texttt{tp} \quad E \vdash \tau_1 \leq \tau_2}{E \vdash \llcorner \tau_1 \lrcorner_{s_1} \leq \llcorner \tau_2 \lrcorner_{s_2}} \qquad \text{(s-t)}\ \frac{E \vdash \{\mathtt{a}\}\vec{\texttt{T}}_i \Rightarrow \tau_i : \texttt{tp} \quad E \vdash \vec{\texttt{T}}_2 \leq \vec{\texttt{T}}_1, \tau_1 \leq \tau_2}{E \vdash \{\mathtt{a}\}\vec{\texttt{T}}_1 \Rightarrow \tau_1 \leq \{\mathtt{a}\}\vec{\texttt{T}}_2 \Rightarrow \tau_2} \qquad \text{(s-p)}\ \frac{s_2 \sqsubseteq s_1 \quad E \vdash \{\mathtt{a}\}\vec{\texttt{T}}_i \overset{s_i}{\Rightarrow} \tau_i : \texttt{tp} \quad E \vdash \vec{\texttt{T}}_2 \leq \vec{\texttt{T}}_1, \tau_1 \leq \tau_2}{\{\mathtt{a}\}\vec{\texttt{T}}_1 \overset{s_1}{\Rightarrow} \tau_1 \leq \{\mathtt{a}\}\vec{\texttt{T}}_2 \overset{s_2}{\Rightarrow} \tau_2}$$

$$\text{(s-ref)}\ \frac{E \vdash \texttt{ref}_{s_i}^{\mathtt{a}_r, \mathtt{a}_w}\langle\texttt{T}\rangle : \texttt{tp} \quad s_3 \sqsubseteq s_2 \sqsubseteq s_1}{\begin{array}{l} E \vdash \texttt{ref}_{s_1}^{\mathtt{a}_r, \mathtt{a}_w}\langle\texttt{T}\rangle \leq \texttt{ref}_{s_2}^{\mathtt{a}_r, \mathtt{a}_w}\langle\texttt{T}\rangle \\ E \vdash \texttt{ref}_{s_1}^{\mathtt{a}_r, \mathtt{a}_w}\langle\texttt{T}\rangle \leq \mathtt{r}_{s_2}^{\mathtt{a}_r}(\texttt{T}) < \mathtt{r}_{s_3}^{\mathtt{a}_r}(\texttt{T}) \\ E \vdash \texttt{ref}_{s_1}^{\mathtt{a}_r, \mathtt{a}_w}\langle\texttt{T}\rangle \leq \mathtt{w}_{s_2}^{\mathtt{a}_w}(\texttt{T}) < \mathtt{w}_{s_2}^{\mathtt{a}_w}(\texttt{T}) \end{array}} \qquad \text{(s-thunk)}\ \frac{E, \vec{x}{:}\vec{\texttt{C}} \vdash \Delta_1 < \Delta_2, \texttt{T}_1 < \texttt{T}_2 \quad E \vdash \exists[\vec{x}{:}\vec{\texttt{C}}]\ulcorner \texttt{T}_i; \Delta_i; \vec{\texttt{D}}\urcorner : \texttt{tp}}{E \vdash \exists[\vec{x}{:}\vec{\texttt{C}}]\ulcorner \texttt{T}_1; \Delta_1, \vec{\texttt{D}}\urcorner < \exists[\vec{x}{:}\vec{\texttt{C}}]\ulcorner \texttt{T}_2; \Delta_2, \vec{\texttt{D}}\urcorner}$$

$$\text{(s-proc)}\ \frac{E \vdash \Delta_i : \texttt{tp} \quad \mathsf{alev}(\Delta_1) < \mathsf{alev}(\Delta_2) \quad \forall u \in \mathsf{dom}(\Delta_1).\Delta_1(u) < \Delta_2(u)}{E \vdash \Delta_1 < \Delta_2}$$

$$\text{(s-obj)}\ \frac{E \vdash \texttt{T}_{\mathtt{m}i} < \texttt{T}'_{\mathtt{m}i} : \texttt{tp}}{E \vdash [\texttt{T}_{\mathtt{m}1} \& .. \& \texttt{T}_{\mathtt{m}n}] < [\texttt{T}'_{\mathtt{m}1} \& .. \& \texttt{T}'_{\mathtt{m}n}]} \qquad \text{(s-out)}\ \frac{E \vdash \texttt{T}_1 < \texttt{T}_2 \quad s_1 \sqsubseteq s_2}{E \vdash (\texttt{T}_1)_{s_1}^{po} < (\texttt{T}_2)_{s_2}^{po}}$$

Figure 26: Well-Formedness and Subtyping

(lang-t-id) $\mathtt{T} \odot \mathtt{T} = \mathtt{T}$ and $\mathtt{T}_r \odot \mathtt{T}_r = \mathtt{T}_r$

(lang-t-ref) $\mathtt{ref}_s^{\mathtt{a}_r,\mathtt{a}_w}(\mathtt{T}) \odot \mathtt{r}_s^{\mathtt{a}_r}(\mathtt{T}) = \mathtt{ref}_s^{\mathtt{a}_r,\mathtt{a}_w}(\mathtt{T}) \odot \mathtt{w}_s^{\mathtt{a}_r}(\mathtt{T}) = \mathtt{ref}_s^{\mathtt{a}_r,\mathtt{a}_w}(\mathtt{T})$;

(lang-t-chan) The composition of channel type σ_r is defined as in $\lambda\pi^{\mathsf{LA}}$, § B.3.

We then define $\Delta_1, \mathtt{a}_1 \odot \Delta_2, \mathtt{a}_2$ as $\Delta_1 \odot \Delta_2, \mathtt{a}_1 \sqcup \mathtt{a}_2$. We also denote $\asymp$ if $\odot$ is defined.

The typing rule is a combination of those in [20] for secrecy and those in § 4.3 in the main section for authority. There are three key points.

- We record the effect when reading/writing references as in [20]. Similarly we record the effect of *object ids* as we do so for the channel types in $\lambda\pi$ (note object ids are passed, instantiated and restricted as the channels, thus we treat them as channels).
- `let` plays a rôle of `sec` in [2, 20].
- The authority level of classes, methods and references are recorded and cancelled when an expression is surrounded by `set` and the frame.

The typing systems for the host and the network follow $\lambda\pi$. Finally a given class signature, a well formed class table, which is a mapping from class names to class declaration, is defined combining all rules together. The typing systems for methods, classes, memories and heaps are listed in Figure 28.

Proving the Authority Error Freedom. We first prove the following lemma by rule induction of $\longrightarrow$.

LEMMA D.2. (subject reduction theorem) *Suppose* $\Gamma \vdash \mathtt{N} : \mathtt{net}, \Delta$ *and* $\mathtt{N} \longrightarrow\!\!\!\!\!\rightarrow \mathtt{N}'$. *Then* $\Gamma \vdash \mathtt{N}' : \mathtt{net}, \Delta$.

Then as in § 4.3, Access Safety is proved via the above lemma. We only have to check reading/writing references, [NewC] and [Method Invocation] do not violate the authority conditions. It is easy to analyse that when expression e interacts with σ, H or $\mathtt{CT}$, the innermost frame surrounding e comes from either `set` or [*Prin*], hence the authority level of the frame is always higher than the resources which e accesses (i.e. the innermost frame protects an expression). Note that in the definition of class declarations ([*Class*] in Figure 28), the authority level of the class is set to the higher level than those of its methods, so that generated expression $e_i\{V/x\}$ via a method invocation $\mathtt{m}_i$ by [Method Invocation] does not violate the accessibility.

Proving the Non-Interference Theorem. This is proved via translations of types and terms into $\lambda\pi^{\mathsf{LAM}}_{<}$. The translations are listed in Figures 29 and 30.

We have a couple of remarks on the encoding.

- Each class `C` is a mapped to a replicated process whose name is $a_{\mathtt{C}}$. It gets a vector of values then instantiates them to newly created references.
- The thunk $\ulcorner e \texttt{ with } \mathtt{CT} \urcorner$ is encoded as a parallel composition of a mapping of e and that of the class table `CT`. Further all class names of `CT` are hidden not to be confused at the receiver side. On the other hand, the names of foundation classes are instantiated at the receiver side (because the foundation classes exist at every location); hence it is mapped to a $\lambda\pi$-process parameterised by foundation class names.
- A serialization is a combination of a `let` and a thunk. A deserialization instantiates parameters (i.e. the foundation class names) to value V and runs it.

$$\begin{cases} \mathtt{T}^\bullet & = (\mathtt{T}^\circ)^{\uparrow_{\mathrm{L}}} \\ \llcorner \mathtt{T} \lrcorner_s^\bullet & \stackrel{\text{def}}{=} (\mathtt{T}^\circ)_s^{\uparrow_{\mathrm{A}}} \end{cases} \qquad \begin{cases} \mathtt{cmd}^\bullet & = ()^{\uparrow_{\mathrm{L}}} \\ \llcorner \mathtt{cmd} \lrcorner_s^\bullet & = ()_s^{\uparrow_{\mathrm{A}}} \end{cases}$$

$$\mathtt{void}^\circ = \Pi(x{:}()^{\uparrow_{\mathrm{L}}})\{x{:}()^{\uparrow_{\mathrm{L}}}\}$$

$$\mathtt{nat}_s^\circ = \Pi(x{:}[\oplus_{i\in\omega}]_s^{\uparrow_{\mathrm{L}}})\{x{:}[\oplus_{i\in\omega}]_s^{\uparrow_{\mathrm{L}}}\}$$

$$\mathtt{C}^\circ = \tau^\circ \text{ with } \mathit{Csig}(\mathtt{C}) = \tau$$

$$\exists[\vec{x}{:}\vec{\mathtt{C}}]\ulcorner \mathtt{T}_v; \Delta; \vec{\mathtt{D}}\urcorner^\circ = \Pi(\vec{y}{:}\vec{\tau})\exists[\vec{x}{:}\overline{\vec{\mathtt{C}^\circ}}]\Pi(g{:}\mathtt{T}_v^\bullet)\{\Delta^\circ, g{:}\mathtt{T}_v^\bullet, \vec{y}{:}\vec{\tau}\}$$
$$\text{where } \vec{\tau} = \overline{\vec{\mathtt{F}^\circ}} \text{ with } \vec{\mathtt{F}} = \mathtt{dom}(\mathtt{FCF}).$$

$$(\mathtt{T}, \Delta)^\circ = \Pi(g{:}\mathtt{T}_v^\bullet)\{\Delta^\circ, g{:}\mathtt{T}_v^\bullet\}$$

$$\mathtt{ref}_s(\mathtt{T})^\circ = \mathtt{ref}_s(\mathtt{T}^\circ), \quad \mathtt{r}_s(\mathtt{T})^\circ = \mathtt{r}_s(\mathtt{T}^\circ) \quad \mathtt{w}_s(\mathtt{T})^\circ = \mathtt{w}_s(\mathtt{T}^\circ)$$

$$\begin{cases} \emptyset^\circ = \emptyset \\ (\Delta, o : \mathtt{C})^\circ = \Delta^\circ, o : \overline{\mathtt{C}^\circ} \\ (\Delta, o.\mathtt{m} : \mathtt{C})^\circ = \Delta^\circ, o.\mathtt{m} : \overline{\mathtt{T}_{\mathtt{m}}^\circ} \quad \text{with} \quad \mathit{mtype}(\mathtt{C}, \mathtt{m}) = \mathtt{T}_{\mathtt{m}} \\ (\Delta, x : \mathbf{T}_r)^\circ = \Delta^\circ, x : \overline{\mathbf{T}_r}^\circ \\ (\Delta, c : \sigma_r)^\circ = \Delta^\circ, c : \sigma_r^\circ \end{cases}$$

$$((\mathtt{T})_s^p)^\circ = (\mathtt{T}^\circ)_s^p, \ \langle \sigma_r, \overline{\sigma_r} \rangle^\circ = \langle \sigma_r^\circ, \overline{\sigma_r}^\circ \rangle$$

$$\begin{cases} (E, u : \tau)^\circ = E^\circ, u : \langle \tau^\circ, \overline{\tau^\circ} \rangle & \text{if} \tau \neq \sigma_r \\ (E, u : \sigma_r)^\circ = E^\circ, u : \sigma_r^\circ \end{cases}$$

$$\begin{cases} (\vec{\mathtt{T}} \Rightarrow \mathtt{T})^\circ & = (\overline{\vec{\mathtt{T}}^\circ} \mathtt{T}^\bullet)^{!_{\mathrm{L}}} \\ (\vec{\mathtt{T}} \Rightarrow \llcorner \mathtt{T} \lrcorner_s)^\circ & = (\overline{\vec{\mathtt{T}}^\circ} \llcorner \mathtt{T} \lrcorner_s^\bullet)^{!_{\mathrm{A}}} \\ (\vec{\mathtt{T}} \stackrel{s'}{\Rightarrow} \llcorner \mathtt{T} \lrcorner_s)^\circ & = (\overline{\vec{\mathtt{T}}^\circ} \llcorner \mathtt{T} \lrcorner_s^\bullet)_{s'}^{!_{\mathrm{M}}} \end{cases}$$

$$(\vec{\mathtt{T}} \Rightarrow \mathtt{T}_o)^\circ = (\overline{\vec{\mathtt{T}}^\circ} \mathtt{T}_o^\bullet)^{!_{\mathrm{L}}}$$

$$[\mathtt{T}_{\mathtt{m}1} \& \ldots \& \mathtt{T}_{\mathtt{m}n}]^\circ = \begin{cases} [\tau_1 \& \ldots \& \tau_n]^{!_{\mathrm{L}}} & \mathtt{T}_{\mathtt{m}i}^\circ = (\tau_i)^{?_{\mathrm{L}}} \\ [\tau_1 \& \ldots \& \tau_n]^{!_{\mathrm{A}}} & \mathtt{T}_{\mathtt{m}i}^\circ = (\tau_i)^{?_{\mathrm{A}}} \\ [\tau_1 \& \ldots \& \tau_n]_s^{!_{\mathrm{M}}} & \mathtt{T}_{\mathtt{m}i}^\circ = (\tau_i)_s^{?_{\mathrm{M}}} \end{cases}$$

Figure 29: Mapping of Types and Environment

- A method invocation is encoded as a selection. Dually a heap is encoded as a branching replication.
- Due to the structural security condition, a reference which stores higher order V, denoted by $\mathtt{ref}\langle xV \rangle$, has the same tampering level as $(\nu y)(\mathtt{ref}\langle xy \rangle \,|\, !yV^\circ)$. Hence we use $\mathtt{ref}\langle xV \rangle$ in the encoding.

The encodings of other expressions are from [20, Figure 18].

For the proof, first we show the subtyping on types in Figure 26 corresponds to the secrecy subtyping in $\lambda\pi^{\mathsf{LAM}}_{<}$.

PROPOSITION D.3. *(subtyping)* $E \vdash \tau_1 \leq \tau_1$ *iff* $E^\circ \vdash \tau_1^\circ \leq \tau_2^\circ$ *iff* $E^\circ \vdash \tau_1^\bullet \leq \tau_2^\bullet$.

The protection levels in the language and the tampering levels in the $\lambda\pi^{\mathsf{LAM}}_{<}$-calculus coincide via encoding.

PROPOSITION D.4. (protection levels and tampering levels) *Assume* τ *is either* $\mathtt{T}$ *or* $\mathtt{T}_r$. *Then*

1. $\mathsf{tamper}(\tau^\bullet) = \mathsf{tamper}(\tau^\circ) = \mathsf{tamper}(\llcorner \tau \lrcorner_s^\circ)$.
2. $\mathsf{tamper}(\mathtt{T}) = \mathsf{tamper}(\mathtt{T}^\bullet)$ *and* $\mathsf{tamper}^{\mathcal{E}}(\mathtt{T}) = \mathsf{tamper}(\overline{\mathtt{T}^\bullet})$
3. $\mathsf{tamper}(\mathtt{T}_r) = \mathsf{tamper}(\mathtt{T}_r^\circ)$ *and* $\mathsf{tamper}^{\mathcal{E}}(\mathtt{T}_r) = \mathsf{tamper}(\overline{\mathtt{T}_r^\circ})$
4. $\mathsf{tamper}(\sigma_r) = \mathsf{tamper}(\sigma_r^\circ)$
5. $\mathsf{tamper}(\Delta) = \mathsf{tamper}(\Delta^\circ)$

Using this proposition, we can prove the coincidence between structurally security of $\lambda\pi^{\mathsf{LAM}}_{<}$ and the language on mutable types.

(Expression)

$$[Sub]\ \frac{E \vdash e : \alpha \quad E \vdash \alpha < \alpha'}{E \vdash e : \alpha'} \qquad [Id]\ \frac{E, u{:}\mathtt{T}, E' \vdash \mathtt{Env}}{E, u{:}\mathtt{T}, E' \vdash u{:}\mathtt{T}} \qquad [Id_d]\ \frac{E, x{:}(\mathtt{T},\Delta), E' \vdash \mathtt{Env}}{E, x{:}(\mathtt{T},\Delta), E' \vdash x{:}\mathtt{T},\Delta} \qquad [Val\exists]\ \frac{E, \{\vec{x}{:}\vec{\mathtt{C}}, y{:}\mathtt{T}\}, E' \vdash \mathtt{Env}}{E, \{\vec{x}{:}\vec{\mathtt{C}}, y{:}\mathtt{T}\}, E' \vdash y : \mathtt{T}}$$

$$[Const]\ \frac{E \vdash \mathtt{Env}}{E \vdash n : \mathtt{nat}_s} \qquad [Succ]\ \frac{E \vdash e : \mathtt{nat}_s}{E \vdash \mathtt{succ}(e) : \mathtt{nat}_s} \qquad [Pred]\ \frac{E \vdash e : \mathtt{nat}_s}{E \vdash \mathtt{pred}(e) : \mathtt{nat}_s} \qquad [Void]\ \frac{E \vdash \mathtt{Env}}{E \vdash \ : \mathtt{void}}$$

$$[Lift]\ \frac{E \vdash e : \mathtt{T},\Delta}{E \vdash e : \llcorner \mathtt{T} \lrcorner_s,\Delta} \qquad [Let]\ \frac{E \vdash e_1 : \mathtt{T},\Delta_1 \quad E, x{:}\mathtt{T} \vdash e_2 : \rho,\Delta_2}{E \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \rho, \Delta_1 \odot \Delta_2/x} \qquad [LetP]\ \frac{E \vdash e_1 : \llcorner \mathtt{T} \lrcorner_{s_1},\Delta_1 \quad E, x{:}\mathtt{T} \vdash e_2 : \rho,\Delta_2 \quad \rho = \llcorner \mathtt{cmd} \lrcorner_{s_2} \text{ or } \llcorner \mathtt{T}' \lrcorner_{s_2}}{E \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \rho, \Delta_1 \odot \Delta_2/x}\ (\dagger 1)$$

$$[Skip]\ \frac{E \vdash \mathtt{Env}}{E \vdash \mathtt{skip} : \mathtt{cmd}} \qquad [Ret]\ \frac{E \vdash e_1 : \mathtt{cmd},\Delta_1 \quad E \vdash e_2 : \mathtt{T},\Delta_2}{E \vdash e_1;\ \mathtt{return}\ e_2 : \mathtt{T}, \Delta_1 \odot \Delta_2/x} \qquad [RetP]\ \frac{E \vdash e_1 : \llcorner \mathtt{cmd} \lrcorner_{s_1},\Delta_1 \quad E \vdash e_2 : \llcorner \mathtt{T} \lrcorner_{s_2},\Delta_2}{E \vdash e_1;\ \mathtt{return}\ e_2 : \llcorner \mathtt{T} \lrcorner_{s_2}, \Delta_1 \odot \Delta_2/x}\ (\dagger 1)$$

$$[Ass]\ \frac{E \vdash x{:}\mathtt{ref}_s^{\mathtt{a}_r,\mathtt{a}_w}(\mathtt{T}) \quad E \vdash V : \mathtt{T}}{E \vdash x := V\ :\ \mathtt{cmd}, x{:}\mathtt{w}_s(\mathtt{T}) \odot \Delta \sqcup \mathtt{a}_w}\ (\dagger 2) \qquad [Com]\ \frac{\begin{array}{l} E \vdash e_i : \mathtt{T}_{ci},\Delta_i\ (i=1,2) \\ \text{if } \mathtt{T}_{c1} = \llcorner \mathtt{cmd} \lrcorner_{s1} \text{ then } \mathtt{T}_{c2} = \llcorner \mathtt{cmd} \lrcorner_{s2} \end{array}}{E \vdash e_1; e_2 : \mathtt{T}_{c2}, \Delta_1 \odot \Delta_2}\ (\dagger 1)$$

$$[If]\ \frac{E \vdash V : \mathtt{nat}_s \quad E \vdash e_i : \mathtt{T}_c,\Delta\ (i=1,2)}{E \vdash \mathtt{if}\ V\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \mathtt{T}_c,\Delta}\ (\dagger 3) \qquad [While]\ \frac{E \vdash x{:}\mathtt{r}_{s'}^{\mathtt{a}}(\mathtt{nat}_s) \quad E \vdash e : \llcorner \mathtt{cmd} \lrcorner_{s_0},\Delta}{E \vdash \mathtt{while}\ !x\ \mathtt{then}\ e : \llcorner \mathtt{cmd} \lrcorner_{s_0}, x{:}\mathtt{r}_{s'}(\mathtt{nat}_s) \odot \Delta \sqcup \mathtt{a}}\ (\dagger 4)$$

$$[Deref]\ \frac{E \vdash z{:}\mathtt{r}_s^{\mathtt{a}}(\mathtt{T}) \quad E, x{:}\mathtt{T} \vdash e : \rho,\Delta_2}{E \vdash \mathtt{let}\ x = !z\ \mathtt{in}\ e : \rho, z{:}\mathtt{r}_s(\mathtt{T}) \odot \Delta_1 \odot \Delta_2/x \sqcup \mathtt{a}} \qquad [Newvar]\ \frac{E \vdash V : \mathtt{T} \quad E \cdot x{:}\mathtt{ref}_s^{\mathtt{a}_r,\mathtt{a}_r}(\mathtt{T}) \vdash e : \rho,\Delta}{E \vdash \mathtt{newvar}\ x\{\mathtt{a}_r,\mathtt{a}_r\}[s] \mapsto V\ \mathtt{in}\ e : \rho, \Delta/x \sqcup \mathtt{a}_r \sqcup \mathtt{a}_r}$$

$$[New]\ \frac{\begin{array}{l} E \vdash V_j : \mathtt{T}_j \\ E, x{:}\mathtt{C} \vdash e : \mathtt{T}_c,\Delta \quad Inst(\mathtt{C}) = \{\mathtt{a}\}\vec{\mathtt{T}} \end{array}}{E \vdash \mathtt{let}\ x = \mathtt{new}\ \mathtt{C}(\vec{V})\ \mathtt{in}\ e : \mathtt{T}_c,\ \vec{o}{:}\vec{\mathtt{C}} \odot \Delta/x \sqcup \mathtt{a}}\ (\dagger 7) \qquad [SetA]\ \frac{E \vdash e : \mathtt{T}_c,\Delta}{E \vdash \mathtt{set}(\mathtt{a})\{e\} : \mathtt{T}_c,\ (\Delta)_\perp}\ (*1) \quad [Prin]\ \frac{E \vdash e : \mathtt{T}_c,\Delta}{E \vdash \{e\}_\mathtt{a} : \mathtt{T}_c, (\Delta)_\perp}\ (*1)$$

$$[Imm]\ \frac{\begin{array}{l} E \vdash V_j : \mathtt{T}_j \quad E, x{:}\mathtt{T} \vdash e : \mathtt{T}_c,\Delta \\ mtype(\mathtt{C},\mathtt{m}) = \{\mathtt{a}\}\vec{\mathtt{T}} \Rightarrow \mathtt{T}_v \end{array}}{E \vdash \mathtt{let}\ x = o.\mathtt{m}(\vec{V})\ \mathtt{in}\ e : \mathtt{T}_c,\ \vec{o}{:}\vec{\mathtt{C}} \odot o.\mathtt{m}{:}\mathtt{C} \odot \Delta/x \sqcup \mathtt{a}}\ (\dagger 7) \qquad [Mut]\ \frac{\begin{array}{l} E \vdash V_j : \mathtt{T}_j \quad E, x{:}\mathtt{T} \vdash e : \llcorner \mathtt{cmd} \lrcorner_{s_2},\Delta_2 \\ mtype(\mathtt{C},\mathtt{m}) = \{\mathtt{a}\}\vec{\mathtt{T}} \overset{S}{\Rightarrow} \llcorner \mathtt{T} \lrcorner_{s_1} \end{array}}{E \vdash \mathtt{let}\ x = o.\mathtt{m}(\vec{V})\ \mathtt{in}\ e : \llcorner \mathtt{cmd} \lrcorner_{s_2},\ \vec{o}{:}\vec{\mathtt{C}} \odot o.\mathtt{m}{:}\mathtt{C} \odot \Delta_2/x \sqcup \mathtt{a}}\ (\dagger 1)$$

$$[Thunk]\ \frac{\begin{array}{l} E \vdash e : \mathtt{T},\Delta \quad e \neq o \quad \mathsf{fv}(\Delta) = \emptyset \quad E \vdash \exists[\vec{x}{:}\vec{\mathtt{C}}]\ulcorner \mathtt{T};\Delta;\vec{\mathtt{D}} \urcorner : \mathtt{tp} \\ \mathtt{dom}(\mathtt{CT}) = \vec{\mathtt{D}} = \mathsf{fc}(e) \setminus \mathtt{dom}(\mathtt{FCT}) \quad E \vdash \mathtt{CT}\ \mathtt{OK} \end{array}}{E \vdash \ulcorner e\ \mathtt{with}\ \mathtt{CT} \urcorner : \exists[\vec{x}{:}\vec{\mathtt{C}}]\ulcorner \mathtt{T};\Delta;\vec{\mathtt{D}} \urcorner} \qquad [Seri]\ \frac{\begin{array}{l} E \vdash \ulcorner e\ \mathtt{with}\ \mathtt{CT} \urcorner : \mathtt{T} \\ E, x{:}\mathtt{T} \vdash e' : \rho,\Delta' \end{array}}{E \vdash \mathtt{let}\ x = \mathtt{serialize}\ (e)\ \mathtt{in}\ e' : \rho,\ \Delta'/x}$$

$$[Des]\ \frac{\begin{array}{l} E \vdash V : \exists[\vec{x}{:}\vec{\mathtt{C}}]\ulcorner \mathtt{T};\Delta,\vec{\mathtt{D}} \urcorner \\ E, \{\vec{x}{:}\vec{\mathtt{C}}, y : (\mathtt{T},\Delta)\} \vdash e_2 : \rho,\Delta_2,\vec{x}{:}\vec{\mathtt{C}} \end{array}}{E \vdash \mathtt{let}\ y = \mathtt{derialize}\ (V)\ \mathtt{in}\ e_2 : \rho,\ \Delta_2/y} \qquad [DesP]\ \frac{\begin{array}{l} E \vdash V : \exists[\vec{x}{:}\vec{\mathtt{C}}]\llcorner \mathtt{T} \lrcorner_{s_1};\Delta;\vec{\mathtt{D}} \urcorner \\ E, \{\vec{x}{:}\vec{\mathtt{C}}, y{:}(\mathtt{T},\Delta)\} \vdash e_2 : \rho,\Delta_2,\vec{x}{:}\vec{\mathtt{C}} \end{array}}{E \vdash \mathtt{let}\ y = \mathtt{derialize}\ (V)\ \mathtt{in}\ e_2 : \rho,\ \Delta_2/y}\ (\dagger 1)$$

$$[RMI]\ \frac{\begin{array}{l} E \vdash V_j : \mathtt{T}_j \quad E \vdash c : \mathtt{T}^\bullet \\ mtype(\mathtt{C},\mathtt{m}) = \{\mathtt{a}\}\vec{\mathtt{T}} \Rightarrow \mathtt{T}_v \end{array}}{E \vdash o.\mathtt{m}(\vec{V})\ \mathtt{at}\ c : \mathtt{cmd},\ \vec{o}{:}\vec{\mathtt{C}} \odot o.\mathtt{m}{:}\mathtt{C}, c{:}\mathtt{T}^\bullet, \mathtt{a}}\ (\dagger 7) \qquad [RMIP]\ \frac{\begin{array}{l} E \vdash V_j : \mathtt{T}_j \quad E \vdash c : \llcorner \mathtt{T} \lrcorner^\bullet_{s_1} \\ mtype(\mathtt{C},\mathtt{m}) = \{\mathtt{a}\}\vec{\mathtt{T}} \overset{S}{\Rightarrow} \llcorner \mathtt{T} \lrcorner_{\mathtt{s}_1} \end{array}}{E \vdash o.\mathtt{m}(\vec{V})\ \mathtt{at}\ c : \llcorner \mathtt{cmd} \lrcorner_{s_1},\ \vec{o}{:}\vec{\mathtt{C}} \odot o.\mathtt{m}{:}\mathtt{C} \odot c{:}\llcorner T \lrcorner^\bullet_{s_1}, \mathtt{a}}\ (\dagger 7)$$

$$[Wait]\ \frac{E \vdash c : \overline{\mathtt{T}^\bullet_v} \quad E, x{:}\mathtt{T}_v \vdash e : \mathtt{T}_c,\Delta_2}{E \vdash \mathtt{wait}\ x\ \mathtt{at}\ c\ \mathtt{in}\ e : \mathtt{T}_c,\ \Delta_2/x}(\dagger 1,5) \qquad [RemR]\ \frac{E \vdash e_1; \mathtt{return}\ e_2 : \mathtt{T}_v,\Delta}{E \vdash e_1; \mathtt{return}\ e_2\ \mathtt{to}\ c : \mathtt{T}_c, c : \mathtt{T}^\bullet_v,\Delta}(\dagger 6)$$

(†1) $s_1 \sqsubseteq s_2 \sqcap \mathsf{tamper}(\Delta_2/x)$. (†2) if $V = o$, then $\Delta' = o{:}\mathtt{T}$ else $\Delta = \emptyset$. (†3) $s \sqsubseteq \mathsf{tamper}(\Delta) \sqcap \mathsf{tamper}(\mathtt{T}_c)$. (†4) $s \sqsubseteq s_0 \sqsubseteq \mathsf{tamper}(\Delta)$. (†5) if $\mathtt{T}^\bullet_v = (\mathtt{T}_v)^{\uparrow_{\mathrm{A}}}_{s_1}$ then $\mathtt{T}_c = \llcorner \mathtt{cmd} \lrcorner_{s_2}$. (†6) if $\mathtt{T}^\bullet_v = (\mathtt{T}_v)^{\uparrow_{\mathrm{A}}}_{s}$, then $\mathtt{T}_c = \llcorner \mathtt{cmd} \lrcorner_{s}$. (†7) $\vec{o}{:}\vec{\mathtt{C}} = \{o_i : \mathtt{C}_i \mid V_i = o_i \wedge \mathtt{T}_i = \mathtt{C}_i\}$
(∗1) $\mathsf{alev}(\Delta) \sqsubseteq \mathtt{a}$.

(Thread)

$$[Thread]\ \frac{E \vdash \{e; \mathtt{return}\ e'\ \mathtt{to}\ c\}_\mathtt{a} : \mathtt{T}_c,\Delta}{E \vdash \{e; \mathtt{return}\ e'\ \mathtt{to}\ c\}_\mathtt{a} : \mathtt{th},\Delta} \qquad [Nil]\ \frac{E \vdash \mathtt{Env}}{E \vdash \mathbf{0} : \mathtt{th},\emptyset} \qquad [Par]\ \frac{E \vdash \mathsf{P}_i : \mathtt{th},\Delta_i \quad \Delta_1 \asymp \Delta_2}{E \vdash \mathsf{P}_1 \mid \mathsf{P}_2 : \mathtt{th},\Delta_1 \odot \Delta_2} \qquad [CRes]\ \frac{E, c{:}\sigma \vdash \mathsf{P} : \mathtt{th},\Delta, c{:}\langle\sigma, \overline{\sigma}\rangle}{E \vdash (\nu c)\mathsf{P} : \mathtt{th},\Delta}$$

(Network)

$$[Host]\ \frac{\begin{array}{l} E \vdash \mathsf{P}_i : \mathtt{th},\Delta_i \quad \Delta_i \asymp \Delta_j \quad \Delta = \odot\Delta_i \\ E \vdash \sigma, H : \mathtt{OK}, \quad E \vdash \mathtt{CT}\ \mathtt{OK} \quad \mathtt{FCT} \subseteq \mathtt{CT} \\ \cup \mathsf{fv}(\mathsf{P}_i) \subseteq \mathtt{dom}(\sigma) = \{\vec{x}\} \quad \{\vec{o}\} \subseteq \mathtt{dom}(H) \end{array}}{E \vdash l[(\nu\vec{x}\vec{o})(\mathsf{P}_1 \mid .. \mid \mathsf{P}_n, \sigma, H, \mathtt{CT})] : \mathtt{net},\Delta/\vec{x}\vec{o}} \qquad [HPar]\ \frac{\begin{array}{l} E \vdash \mathsf{N}_i : \mathtt{net},\Delta_i \quad \Delta_1 \asymp \Delta_2 \\ \mathsf{fl}(\mathsf{N}_1) \cap \mathsf{fl}(\mathsf{N}_2) = \emptyset \\ \mathtt{dom}_H(\mathsf{N}_1) \cap \mathtt{dom}_H(\mathsf{N}_2) = \emptyset \end{array}}{E \vdash \mathsf{N}_1 \mid \mathsf{N}_2 : \mathtt{net},\Delta_1 \odot \Delta_2}$$

$\mathsf{fl}(\mathsf{N})$ is a set of free location names, and $\mathtt{dom}_H(\mathsf{N})$ is a set of heap names, inductively defined as: $\mathtt{dom}_H(\mathbf{0}) = \emptyset$, $\mathtt{dom}_H(l[\mathsf{P},\sigma,H,\mathtt{CL}]) = \mathtt{dom}(H)$, $\mathtt{dom}_H((\nu u)\mathsf{P}) = \mathtt{dom}_H(\mathsf{P})/u$ and $\mathtt{dom}_H(\mathsf{P}_1 \mid \mathsf{P}_2) = \mathtt{dom}_H(\mathsf{P}_1) \cup \mathtt{dom}_H(\mathsf{P}_2)$. We omit *[Nil,CRes]* for the network.

Figure 27: Typing System for Simple Distributed Class Based Language with Authority and Secrecy

(Method/Class)

$$[MeT]\quad \frac{E,\vec{x}{:}\vec{\mathtt{T}} \vdash e_1;\mathtt{return}\ e_2 : \mathtt{T}_v,\Delta \quad mtype(\mathtt{C},\mathtt{m}) = \{\mathtt{a}\}\vec{\mathtt{T}} \Rightarrow \mathtt{T}_v \quad E \vdash \mathtt{C} : \mathtt{tp}}{E \vdash \mathtt{T}\ \mathtt{m}\{\mathtt{a}\}\ \mathtt{return}[s](\vec{\mathtt{T}}\ \vec{x})\{e_1;\mathtt{return}\ e_2;\} : \Delta/\vec{x},\ \mathtt{m\ OK\ in\ C}} \quad (\ddagger 1,2)$$

$$[MeP]\quad \frac{E,\vec{x}{:}\vec{\mathtt{T}} \vdash e_1;\mathtt{return}\ e_2 : \mathtt{T},\Delta \quad mtype(\mathtt{C},\mathtt{m}) = \{\mathtt{a}\}\vec{\mathtt{T}} \overset{s}{\Rightarrow} \llcorner \mathtt{T} \lrcorner_{s_2} \quad E \vdash \mathtt{C} : \mathtt{tp}}{E \vdash \mathtt{T}\ \mathtt{m}\{\mathtt{a}\}[s]\mathtt{return}[s_2](\vec{\mathtt{T}}\ \vec{x})\{e_1;\mathtt{return}\ e_2;\} : \Delta/\vec{x},\ \mathtt{m\ OK\ in\ C}} \quad (\ddagger 1,3)$$

$$[Class]\quad \frac{E,\vec{x}{:}\mathtt{ref}_{\vec{s}}^{\vec{\mathtt{a}}_r,\vec{\mathtt{a}}_w}(\vec{\mathtt{T}}) \vdash \vec{\mathtt{M}} : \Delta,\ \mathtt{m\ OK\ in\ C} \quad Csig(\mathtt{C}) = \{\mathtt{a}\}\mathtt{T} \Rightarrow \mathtt{T}_o \quad E \vdash \mathtt{C} : \mathtt{tp}}{E \vdash \mathtt{class\ C}\ \{\mathtt{a}\}[s]\ \{\vec{\mathtt{T}} : \{\vec{\mathtt{a}}_r,\vec{\mathtt{a}}_w\}[\vec{s}]\ \vec{x}\ ;\ \vec{\mathtt{M}}\} : \mathtt{OK}} \quad (\ddagger 4)$$

($\ddagger$1) $\mathsf{alev}(\Delta) \sqsubseteq \mathtt{a}$ ($\ddagger$2) if $\mathtt{T}_v = \llcorner \mathtt{T} \lrcorner_s$ then $\mathsf{tamper}(\Delta/\vec{x}\vec{c}) = \top$; if $\mathtt{T}_v = \mathtt{T}$ then $\Delta/\vec{x}\vec{c}$ is immutable and $s = \bot$; ($\ddagger$3) $s \sqsubseteq \mathsf{tamper}(\Delta/\vec{x}\vec{c})$ ($\ddagger$4) $\Delta/\vec{x}\vec{c}$ immutable, and $\mathsf{alev}(\mathtt{T}_o) \sqsubseteq \mathtt{a}$. $\mathsf{tamper}(\mathtt{T}_o) = s$.

$$[empty]\ \frac{E \vdash \mathtt{Env}}{E \vdash \emptyset : \mathtt{OK}} \qquad [Env\text{-}\sigma]\ \frac{E \vdash \sigma : \mathtt{OK} \quad E \vdash x{:}\mathtt{ref}_s(\mathtt{T}) \quad E \vdash V : \mathtt{T} \quad x \notin \mathsf{dom}(\sigma)}{E \vdash \sigma \cup [x \mapsto V] : \mathtt{OK}} \qquad [Heap]\ \frac{E \vdash o{:}\mathtt{C} \quad E \vdash H : \mathtt{OK} \quad o \notin \mathsf{dom}(H)}{E \vdash H \cup [o \mapsto \mathtt{C}] : \mathtt{OK}}$$

Figure 28: Typing Rules for Method, Class, Memory and Heaps

(Value)

$u^\circ \overset{\text{def}}{=} u$

$\varepsilon^\circ \overset{\text{def}}{=} \lambda(x).\overline{x}$

$n^\circ \overset{\text{def}}{=} \lambda(x).\overline{x}\mathtt{in}_n$

$\mathtt{succ}(y)^\circ \overset{\text{def}}{=} \lambda(x).\overline{y}(e)e[\&_n\ .\overline{x}\mathtt{in}_{n+1}]$

$\ulcorner e\ \mathtt{with\ CT} \urcorner^\circ \overset{\text{def}}{=} \lambda(\vec{a}_{\vec{\mathtt{F}}},g).(\nu\vec{c})([\![e]\!]_g \,|\, [\![\mathtt{CT}]\!]_{\vec{a}_{\vec{c}}})$ ($\ddagger$)

(Term)

$[\![V]\!]_u \overset{\text{def}}{=} \overline{u}\langle V^\circ\rangle$

$[\![\mathtt{skip}]\!]_u \overset{\text{def}}{=} \overline{u}$

$[\![x := V]\!]_u \overset{\text{def}}{=} \overline{x}\mathtt{inr}\langle V^\circ, u\rangle$

$[\![e_1;e_2]\!]_u \overset{\text{def}}{=} (\nu e)([\![e_1]\!]_c \,|\, c.[\![e_2]\!]_u)$

$[\![\mathtt{if}\ V\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\!]_u \overset{\text{def}}{=} (\nu y)([\![V]\!]_y \,|\, \mathtt{ifzero}\langle y, [\![e_1]\!]_u, [\![e_2]\!]_u\rangle)$

$[\![\mathtt{while}\ !x\ \mathtt{do}\ e]\!]_u \overset{\text{def}}{=} (\nu c)(\overline{c}\langle u\rangle \,|\, !f(k).\overline{c}\langle k\rangle \,|\, !c(k).\overline{x}\mathtt{inl}(c')\mathtt{ifzero}\langle c', (\nu l)([\![e]\!]_l \,|\, l.\overline{f}\langle k\rangle), \overline{k}\rangle)$

$[\![\mathtt{let}\ x = e\ \mathtt{in}\ e']\!]_u \overset{\text{def}}{=} (\nu m)([\![e]\!]_m \,|\, m(x).[\![e']\!]_u)$

$[\![\mathtt{newvar}\ x \mapsto V\ \mathtt{in}\ e]\!]_u \overset{\text{def}}{=} (\nu x)(\mathtt{ref}\langle xV^\circ\rangle \,|\, [\![e]\!]_u)$

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

$[\![\mathtt{let}\ x = \mathtt{new\ C}(\vec{V})\ \mathtt{in}\ e]\!]_u \overset{\text{def}}{=} (\nu c)(\overline{a_{\mathtt{C}}}\langle \vec{V}^\circ, c\rangle \,|\, c(x).[\![e]\!]_u)$

$[\![\mathtt{let}\ x = o.\mathtt{m}(\vec{V})\ \mathtt{in}\ e]\!]_u \overset{\text{def}}{=} (\nu c)(\overline{o}.\mathtt{m}\langle \vec{V}^\circ, c\rangle \,|\, c(x).[\![e]\!]_u)$

$[\![\mathtt{let}\ x = \mathtt{serialize}(e)\ \mathtt{in}\ e']\!]_u^{\mathtt{CT}} \overset{\text{def}}{=} (\nu c)([\![\ulcorner e\ \mathtt{with\ CT} \urcorner]\!]_c \,|\, c(x).[\![e']\!]_u)$ ($\dagger$)

$[\![\mathtt{let}\ x = \mathtt{deserialize}(V)\ \mathtt{in}\ e]\!]_u \overset{\text{def}}{=} (\nu c)(V^\circ \vec{a}_{\vec{\mathtt{F}}}\, c \,|\, c(x).[\![e]\!]_u)$ ($\ddagger$)

$[\![\mathtt{return}\ e]\!]_u \overset{\text{def}}{=} [\![\mathtt{return}\ e\ \mathtt{to}\ u]\!] \overset{\text{def}}{=} (\nu m)([\![e]\!]_m \,|\, m(x).[\![x]\!]_u)$

(Memory, Heap, Method, Class)

$[x \mapsto V]^\circ \overset{\text{def}}{=} \mathtt{ref}\langle xV^\circ\rangle$

$[o \mapsto \vec{\mathtt{M}}]^\circ \overset{\text{def}}{=} [\![\vec{\mathtt{M}}]\!]_o$

$[\![\vec{\mathtt{M}}]\!]_o \overset{\text{def}}{=} !o[\&_i\ \mathtt{m}_i(\vec{y}_i, c_i).[\![e_i]\!]_{c_i}] \quad (\mathtt{M}_i = \mathtt{m}_i(\vec{y}_i)\{e_i;\})$

$\mathtt{class\ C}\ \{\vec{\mathtt{T}}\ \vec{x}\ ;\ \vec{\mathtt{M}}\}^\circ = !a_{\mathtt{C}}(\vec{y},z).\overline{z}(f)(\nu\vec{x})(\mathtt{ref}\langle x_i y_i\rangle \,|\, [\![\vec{\mathtt{M}}]\!]_f)$

(Principal, Network)

$[\![(\nu\vec{v})(e_1 \,|\, \ldots \,|\, e_n)]\!] \overset{\text{def}}{=} (\nu\vec{v})([\![e_1]\!] \,|\, \ldots \,|\, [\![e_1]\!])$

$[\![\{e\}_{\mathtt{a}}]\!]_u \overset{\text{def}}{=} \{[\![e]\!]_u\}_{\mathtt{a}} \quad [\![\mathtt{set(a)}\{e\}]\!]_u \overset{\text{def}}{=} \mathtt{set(a)}\{[\![e]\!]_u\}$

$[\![l[\mathtt{P}, \sigma, H, \mathtt{CT}]]\!] \overset{\text{def}}{=} l[[\![\mathtt{P}]\!]^{\mathtt{CT}}, \sigma^\circ, H^\circ, \mathtt{CT}^\circ]$

$[\![(\nu\vec{c})(\mathtt{H}_1 \,|\, \ldots \,|\, \mathtt{H}_n)]\!] \overset{\text{def}}{=} (\nu\vec{c})([\![\mathtt{H}_1]\!] \,|\, \ldots \,|\, [\![\mathtt{H}_n]\!])$

$[\![\mathtt{wait}\ z\ \mathtt{at}\ c\ \mathtt{in}\ e]\!]_u \overset{\text{def}}{=} c(z).[\![e]\!]_u$

$[\![\overline{o}.\mathtt{m}(\vec{V})\ \mathtt{to}\ c]\!] \overset{\text{def}}{=} \overline{o}.\mathtt{m}\langle \vec{V}^\circ, c\rangle$

$\mathtt{ifzero}\langle x, P, Q\rangle \overset{\text{def}}{=} x(y).(\nu c)(xc \,|\, c[\&_i\ .R_i])$ ($R_0 = P, R_i = Q$ if $i \geq 1$)

($\ddagger$) $\vec{\mathtt{F}} = \mathsf{dom}(\mathtt{FCF})$

($\dagger$) $\mathtt{CT} \subset (\mathtt{CT}' \setminus \mathtt{FCT}) \quad \mathsf{dom}(\mathtt{CT}) = \mathsf{fc}(e_1) \setminus \mathsf{dom}(\mathtt{FCT})$

We omit $\mathtt{CT}$ from $[\![\ ]\!]$ and $(\ )^\circ$.

Figure 30: Encoding of the Language into the $\lambda\pi_<^{\mathsf{LAM}}$-Calculus

PROPOSITION D.5. (computational adequacy) *Suppose* $E \vdash \mathtt{N} : \mathtt{net}, \Delta$ *and* $\mathtt{N} \Downarrow_{\langle c,s \rangle}$. *Then* $[\![\mathtt{N}]\!] \Downarrow_{c^{\dagger_{\mathrm{A}}}}$ *and* $E^{\circ} \vdash c : (\mathtt{void}^{\circ})^{\uparrow_{\mathrm{A}}}_{s}$.

With this proposition, we can prove the main theorem via the same propositions stated in C.1 following the same routine in [20, § 7.8].