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, $\lambda\pi$ [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.

$$\mathsf{Server}(a) \stackrel{\mathrm{def}}{=} !a(y\!:\!\tau).\,run\ y \qquad (1)$$

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.

$$\Gamma \vdash P\ :\ \Delta \qquad (2)$$

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

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

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

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

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 $\mathtt{proc}$ (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 $\mathtt{proc}$, 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.

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

Here $\overline{\mathtt{ack}}$ is a trigger carrying the empty vector, hence typed by $\sigma = ()^{\mathtt{O}}$. But how can we type a when $\mathtt{ack}$ is restricted? In the system in [52], the only possible typing for a is $(\ulcorner\mathtt{proc}\urcorner)^{\mathtt{O}}$, using the most uninformative type $\mathtt{proc}$, so that, for composition with $\mathsf{Send}(a)$, $\mathsf{Server}(a)$ can only be assigned the type $\mathtt{proc}$. 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 $()^{\mathtt{O}}$, 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:

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

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 $(\ulcorner\mathtt{proc}\urcorner)^{\mathtt{O}}$ to a, hence $\mathsf{Server}(a)$ again ends up with $\mathtt{proc}$.

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 $\mathsf{Server}(\mathsf{a})$ 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 $\mathtt{proc}$ 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:

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

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:

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

this process should be typable with affine types, since hidden 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:

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

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:

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

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 $a, b, \ldots$, and an infinite set of *variables* $\mathbf{V}$, ranged over by $x, y, \ldots$ 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.

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

(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
	$\mid$	$P\,P$	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**)

$\ulcorner P \urcorner$	$\stackrel{\text{def}}{=}$	$\lambda(x : \mathtt{unit})P$	$x \notin \mathsf{fv}(P)$	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 $\mathtt{unit}$, $\mathtt{nat}$, .., 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 $\mathtt{proc}$ 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 $\Pi(x_1 : \sigma_1)(\Pi(x_2 : \sigma_2) \ldots (\Pi(x_n : \sigma_n)\tau_i) \ldots)$.

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

Abbreviation:

$(\Pi[x_1 : \sigma_1, \ldots, x_n : \sigma_n]\tau_1, \ldots, \tau_m)^p \stackrel{\text{def}}{=} (\sigma_1, \ldots, \sigma_n, \tau_1, \ldots, \tau_m)^p$
(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 $\mathtt{nat}$ 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:

$(\lambda(y{:}\ulcorner x{:}(\mathtt{nat})^{\mathtt{O}}, b{:}(\mathtt{unit})^{\mathtt{O}\urcorner}).y)\{a/x\}$ is defined but
$(\lambda(y{:}\ulcorner x{:}(\mathtt{nat})^{\mathtt{O}}, b{:}(\mathtt{unit})^{\mathtt{O}\urcorner}).y)\{b/x\}$ 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.

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

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.

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

We discuss each assumption and conclusion.

- $\Gamma \vdash u{:}(\Pi[x{:}\overline{\sigma}]\tau)^{\mathtt{O}}$ 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:

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

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

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

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.

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

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

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

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

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.

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

with $\Delta = \{v{:}(\mathtt{int})^{\mathtt{O}}, c{:}\langle\sigma_c, \overline{\sigma_c}\rangle\}$ and $\Delta' = \{v{:}(\mathtt{int})^{\mathtt{O}}, b{:}(\mathtt{int})^{\mathtt{O}}, c{:}\langle\sigma_c', \overline{\sigma_c'}\rangle\}$. 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 $\sigma_v = \langle(\mathtt{int})^{\mathtt{I}}, (\mathtt{int})^{\mathtt{O}}\rangle$ 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)

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

(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* $P \longrightarrow\!\!\!\!\to P'$. *Then* $\Gamma \vdash P' : \tau$.

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

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

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

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

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.

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

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:

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

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:

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

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

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

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

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

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 $\exists[e{:}(\mathtt{nat})^{\mathtt{O}}]\tau'$, 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, $\overline{a} \mid a.\mathbf{0}$ is typable, while $\overline{a} \mid \overline{a}$ 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, $a.\overline{b} \mid b.\overline{a}$ 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 $\Downarrow_{a^{\mathsf{O}}}$ 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* $P \twoheadrightarrow P'$. *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* $P \twoheadrightarrow P'$, *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* $P' \twoheadrightarrow Q \stackrel{\overline{c}\langle n\rangle}{\longrightarrow}$.

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:

$$\mathtt{Server}(l) \mid \mathtt{let}\ x = \overline{l}\langle\mathtt{twice}\rangle\ \mathtt{in}\ R \mid R' \twoheadrightarrow (\mathtt{Server}(l) \mid R\{2n/x\} \mid R'')$$

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:

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

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.

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

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

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

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

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

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 $\lambda\pi^{\mathsf{LAM}}_{\leq}$.

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 $\lambda\pi^{\mathsf{LAM}}_{\leq}$, 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 $\lambda\pi^{\mathsf{LAM}}_{\leq}$ 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:

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

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 $\tau = [\oplus_i\vec{\tau}_i]^{\uparrow_{\mathrm{L}}}_s$, $\mathsf{md}(\tau) \in \{\uparrow_{\mathrm{A}}, ?_{\mathrm{M}}, ?_{\mathrm{LM}}\}$ 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 $\mathsf{md}(\tau) \in \{?_{\mathrm{L}}, ?_{\mathrm{A}}, \updownarrow\}$ (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 $\pi^{\mathsf{LAM}}_{\mathsf{sec}}$ 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 $(\tau)^{p_{\mathrm{O}}}_s$ 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 $\vdash P_1 \cong_s P_2 : x\!:\!()_s^{\uparrow_{\mathsf{A}}}$ and $P_1 \Downarrow_{x^{\uparrow_{\mathsf{A}}}}$ then $P_2 \Downarrow_{x^{\uparrow_{\mathsf{A}}}}$ [18]. The $[\![\ ]\!]$ and $(\ )^\circ$ are embeddings of terms/types from $\lambda\pi_{<}^{\mathsf{LAM}}$ to the $\pi_{\text{sec}}^{\mathsf{LAM}}$-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:

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

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 $\mathsf{guest} \sqsubseteq \mathsf{bob}$, the code has the static level bob since it may run as bob. If we have $\mathsf{guest} \sqsubseteq \mathsf{alice}$ 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.

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

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

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

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

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 $\lambda\pi_{\leq}^{\mathsf{LAM}}$ 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 $\lambda\pi_{\leq}^{\mathsf{LAM}}$ 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 $\{P\}_{\mathsf{a}} \xrightarrow{Aerr}$ (read: "$\{P\}_{\mathsf{a}}$ does a dangerous thing"), closing the following rule under parallel composition and restriction.

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

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 $\Delta, \mathsf{a} \leq \Delta', \mathsf{a}'$ iff $\Delta \leq \Delta'$ 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* $\mathsf{P} \twoheadrightarrow \mathsf{P}'$ *implies* $\Gamma \vdash \mathsf{P}' : \Delta$.

COROLLARY 4.4. (Access Safety) *Suppose* $\Gamma \vdash \mathsf{P} : \Delta$. *Then for all* P' *such that* $\mathsf{P} \twoheadrightarrow \mathsf{P}'$, *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 $\mathtt{set}$ (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.

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

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) **Location (2)**

```
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;
}
}
```

Location (1):

```
..
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. $\mathtt{m}$ denotes a methods name; $\mathtt{C}, \mathtt{D}$ are class names; $\mathtt{T}$ 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 $\mathtt{m}$ with the result type $\mathtt{T}$, 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 $\mathtt{C}$ 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 $\mathtt{CT}$ 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 $\mathtt{CT}$ at each host contains: (1) a common, fixed *foundation class table*, written $\mathtt{FCT}$, 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]

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

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

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 $\Delta = \{\mathtt{myobj.ack:Ack}\}$ 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:

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

This type indicates: (1) it returns (evaluates to) `double`; (2) its access to other objects are at most $\{\mathtt{myobj.ack:Ack}\}$ 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.

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

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:

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

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.

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

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

[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 $\lambda\pi_{<}^{\mathsf{LAM}}$.

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

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.

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

$s, s', \ldots$ 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 $\texttt{caller}(\mathsf{a})$ 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 $\lambda\pi^{\mathsf{LAM}}_{\leq}$.

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

Access errors are formalised as:

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

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

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.

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

where a in the thunk type is the authority level of effects (which corresponds to a of "Δ, a" in $\lambda\pi^{\mathsf{LAM}}_{\leq}$ in § 4.3). The secrecy typing, whose typability is written $E \vdash e : \rho, \Delta, \mathsf{a}$ and $E \vdash \mathtt{N} : \texttt{net}, \Delta$, 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 $\texttt{deserialize}(\mathtt{x})$ 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* $E \vdash \mathtt{N}:\texttt{net},\Delta$. *Then for all* $\mathtt{N}'$ *such that* $\mathtt{N} \twoheadrightarrow \mathtt{N}'$, *we have* $\mathtt{N}' \not\xrightarrow{Aerr}$.

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.

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

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 $\mathtt{N}[\sigma@l]$ 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* $E \vdash \mathtt{N}[\sigma_i@l]:\texttt{net},\Delta$ *such that* $\sigma_1 \sim_s \sigma_2$. *Then* $\mathtt{N}[\sigma_1@l] \Downarrow_{\langle c,s\rangle}$ *iff* $\mathtt{N}[\sigma_2@l] \Downarrow_{\langle c,s\rangle}$.

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, $D\pi$ [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 $D\pi$. 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 $\lambda\pi_{\leq}^{\mathsf{LAM}}$). 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 $\lambda\pi_{\leq}^{\mathsf{LAM}}$ [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.