Channel Dependent Types for Higher-Order Mobile Processes (Part I)

NOBUKO YOSHIDA

Department of Computing, Imperial College London. E-mail: yoshida@doc.ic.ac.uk

ABSTRACT. We introduce a new expressive theory of types for the higher-order π-calculus which significantly improves our previous work presented in [65] 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 [65], leading to significant enlargement of original typability and offering a general basis for analysing and verifying higher-order mobile computation. This paper establishes a basic theory of dependent and existential types for the higher-order π-calculus, and coherently integrates the resulting formalism with the linear/affine typing disciplines as well as state, concurrency and distribution [6, 27, 63, 66], which allow precise embedding of major language constructs in the calculus. The type discipline can guarantee unique accessibility of resources, termination of migrating code, a response property from a remote server and encapsulation of higher-order code by hidden names. The technical development in this paper gives a firm basis for the applications discussed in its sequels. Part II presents uniform type-based static analyses of two basic security concerns for higher-order mobility, secrecy for data confidentiality and access controls for authorised resources. Part III uses this framework for developing and justifying a type discipline for a simple multi-threaded class-based language with distributed code mobility which enforces secrecy and safe access controls.

1 Introduction

Nowadays the use of mobile code is widespread throughout computing scenes, from intranet 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 [18, 32] and CLR [9, 38]) is to allow *extensibility* [17, 58], 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 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 for analysing and ensuring key security concerns such as integrity and confidentiality [1, 2, 16, 27, 29, 41, 44, 52, 56, 66].

The higher-order π-calculus [47] 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 [47] and reaching a recent work by Jeffrey and Rathke [33]. 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 [6, 7, 24, 27, 40, 60, 63, 66]. 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 [2, 16, 41, 44]. 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 expressiveness and flexibility by articulating higher-order computation by types. The theory is a significant improvement on the author's preceding work with Matthew Hennessy [64, 65], 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 [64].

ASSIGNING TYPES TO PROCESSES We use the call-by-value version of the higher-order π-calculus, $\lambda\pi$ [64, 65]. In the calculus, not only names but also arbitrary higher-order abstracted processes are passed around along communication channels. As a simple example, a thunked process $\lambda(x\!:\!\mathtt{unit}).P$, abbreviated to $\ulcorner P \urcorner$, can be sent and received as a value. The following expresses a server which repeatedly receives thunked processes via port a.

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

Here *run* stands for a function $\lambda y.y()$ which executes a thunked process by applying the unit $()$. Since accepting arbitrary processes for execution obviously breaks access control of local resources [18, 32], one might wish to restrict the behaviour of incoming code so that it can only access some specified channels.

In [65], Matthew Hennessy and the present author introduced a type discipline which can control the effect of migrating code, by assigning a different type to each process depending on its intended use, so that a process can use a typed inputting channel (τ at a in (1) above) to detect, for example, malicious behaviour of received code via static type checking. A type is given as a finite channel environment Δ, prescribing channel usage of each process.

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

This judgement means "P accesses channels at most as specified by Δ under environment Γ". For example, under appropriate $\Gamma \supset \{b:(\mathtt{nat})^{\mathtt{O}}, c:(\mathtt{nat})^{\mathtt{O}}\}$ (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.

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

Now the following indicates a server that 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 [65], 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 $\circ$ or *ok* in the literature), as found in all work [33, 47, 48, 64] before [65], τ can be assigned only $\ulcorner\mathtt{proc}\urcorner$, hence parallel composition with 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 [65] 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. This is a common practice in standard distributed object-oriented programming [69]. Consider the following sender to the above server, which wishes to get the acknowledgement $\overline{\mathtt{ack}}$ at the time P is executed in the remote site. Naturally, this acknowledgement will be sent via a private channel to ensure its safe delivery.

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

Here $\overline{\mathtt{ack}}$ is a trigger carrying the empty vector, hence this channel is typed by $\sigma = ()^{\mathtt{O}}$. But how can we type a when $\overline{\mathtt{ack}}$ is restricted? In the system in [65], 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 $\mathsf{Send}(a)$ above, $\mathsf{Server}(a)$ 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 [65] for more discussions). A related case is non-deterministic name instantiation:

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

Note this process includes another sender to $\mathsf{Server}(a)$ under input prefix. Since we cannot decide which of e_1 and e_2 may be instantiated, the system in [65] 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 (the latter being the case in standard dependent/existential types). For example, if we assume a type of P is Δ, the new typing system assigns to the channel a of $\mathsf{Server}(a)$ 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 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 (which closely 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 type erasure 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 [6, 7, 23, 34, 35, 61–63, 66], 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.

One of the significant features of the resulting typed formalism is that it allows a clean integration with the linear/affine type disciplines [6, 7, 27, 62, 63, 66], which is practically impossible in the foregoing type disciplines for higher-order mobile processes. The linear/affine type disciplines are significant in that, among others, they enable precisely modelling of the statics and dynamics of programming language constructs with code distribution. We briefly discuss how this integration is done in the following.

UNICITY OF NAMES AND FINE-GRAINED PROCESS TYPES Ensuring uniqueness of resource name usage is important in both processes and functions [6, 7, 23, 29, 35, 53, 61, 63, 66]. In distributed system, this condition also guarantees determinacy of the message delivery and preservation of locations. This notion, unicity of names, also works in harmony with fine-gained process types. 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* [6,27], and denote the type of this trigger as $()^{\uparrow}$ where $\uparrow$ means output is used at most once. For example, in the following processes:

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

the server violates affinity since it should repeatedly accept an affine message $\overline{\mathtt{ack}}$. Dually, the sender must not send the acknowledgement to the following server.

$$!\,a(y\!:\!\tau).(run\ y \mid run\ y) \tag{7}$$

Based on the fine-grained process types, we can straightforwardly formulate the affine typing system for $\lambda\pi$ (which makes above terms in (6) and (7) untypable) following the linear type discipline in the π-calculus [6,35,66]. For the example in (6), to accept the messages, $run\,y$ in $\mathsf{Sever}(a)$ should have typed as

$$\Gamma, y\!:\!\tau \vdash run\,y \ :\ \mathtt{ack}\!:\!()^{\uparrow}, \Delta \tag{8}$$

where we assume Δ is a type of P_1 and P_2. In this process type, $\mathtt{ack}\!:\!()^{\uparrow}$ represents that $run\,y$ actually contains the *free* affine output. Hence we know $run\,y$ should not be prefixed by the replication "$!a(y)$" because, if we do so, it means a free channel $\mathtt{ack}$ repeatedly appears violating the unicity of $\mathtt{ack}$. This idea follows the standard linear type discipline of the π-calculus [6,35,66] where $!a.\overline{\mathtt{ack}}$ is untypable with the same reason. Similarly, by process type of $run\,y$, we can check "$run\,y \mid run\,y$" is untypable (as "$\overline{\mathtt{ack}} \mid \overline{\mathtt{ack}}$" is so in π) since a composition of two affine types $\mathtt{ack}\!:\!()^{\uparrow}$ is prohibited. Note that such arguments are unconceivable if only constant $\mathtt{proc}$-type exists.

The above examples do not require the name dependent/existential types. The following example shows subtlety in the interplay between unicity of names and name-hiding in the presence of higher-order mobility. Recall the example in (6). If we *hide* $\overline{\mathtt{ack}}$ at the sender side:

$$\mathsf{Server}(a) \mid (\nu\,\mathtt{ack})(\overline{a}\langle \ulcorner P_1 \mid \overline{\mathtt{ack}} \urcorner \rangle \mid \mathtt{ack}.Q_1) \mid (\nu\,\mathtt{ack})(\overline{a}\langle \ulcorner P_2 \mid \overline{\mathtt{ack}} \urcorner \rangle \mid \mathtt{ack}.Q_2) \tag{9}$$

this process should be typable with affine types, since hidden $\mathtt{ack}$s now denote distinct names. This resembles the following π-processes in which the unicity of $\mathtt{ack}$ is guaranteed by hiding.

$$!a(x).\overline{x} \mid (\nu\,\mathtt{ack})(\overline{a}\langle \mathtt{ack} \rangle \mid \mathtt{ack}.Q_1) \mid (\nu\,\mathtt{ack})(\overline{a}\langle \mathtt{ack} \rangle \mid \mathtt{ack}.Q_2)$$

Typing the example in (9) becomes possible by the use of the new dependent types, so that the server 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, see § 3).

LINEARITY AND LIVENESS Affinity is useful for guaranteeing the unicity of channels and resource availability: however we may wish to guarantee a stronger constraint, for example, in (4) $\overline{\mathtt{ack}}$ *eventually* returns from the server side. This constraint is called *linearity*, which has also a benefit when a precise embedding of language constructs is necessary [27,63,66,68,69].

Here "linearity" means more than just terminating behaviour: it indicates, in essence, a process always returns an answer if it is asked for one. Mixture of nontermination (affinity) with linearity is fine-grained: even inside a process engaged in a linear interaction, nonterminating computation may take place, and linear interactions themselves can involve nontermination and nondeterministic computation (e.g. by access to shared resources). For example, take the following process which is slightly modified from the initial example in (9):

$$!a(x).(\nu b)(xb \mid !b.\overline{b}) \mid (\nu \overline{\mathtt{ack}})(\overline{a}\langle \lambda(x).(\overline{x}|\overline{\mathtt{ack}})\rangle \mid \mathtt{ack}.P) \tag{10}$$

The right process (client) first sends code to the left process (server) at a, then the server does an infinite series of actions at hidden name b, while the client can get a linear answer at $\mathtt{ack}$: here interaction at b is affine, while $\mathtt{ack}$ is linear. A rigorous proof that actions do satisfy liveness needs careful analysis of operational structures [62], for which we use a new type-preserving translation into the corresponding typed π-calculus.

Note that the liveness above needs be guaranteed in the presence of a powerful communication primitive, higher-order process passing, by which arbitrary recursion and replication of processes are representable without explicitly incorporating recursion or replication [48, 55]. We shall later show how we can realise a general framework for guaranteeing liveness in interaction in this sense via a systematic integration of the linear/affine type disciplines in the dependently typed $\lambda\pi$. This integration gives a firm basis for the applications we shall discuss in Part II and III of the present work.

SUMMARY OF CONTRIBUTIONS OF PART I We summarise the main technical contributions of Part I.

1. Introduction of channel dependent/existential types for the $\lambda\pi$-calculus, a higher-order π-calculus first presented in [65], significantly enlarging typability and offering a general basis for applications. Type-safety is proved by extending the technique developed in [65]. The minimum typing theorem (i.e. each process has the best possible fine-grained type) is also established.

2. Consistent integration of linear/affine type disciplines with state, concurrency and non-determinism [6, 27, 63, 66]. We prove the following results in each integrated $\lambda\pi$-calculus.

   - ($\lambda\pi^{\mathsf{A}}$: the affine $\lambda\pi$) Resource availability and unique accessibility of resources.
   - ($\lambda\pi^{\mathsf{L}}$: the linear $\lambda\pi$) Strong normalisation of higher-order processes, closing the open problem posed in [49].
   - ($\lambda\pi^{\mathsf{LA}}$: the linear/affine $\lambda\pi$) A communication liveness property (i.e. linear channels always fire) under the strong fairness condition and encapsulation of higher-order code by hidden name (i.e. the higher-order message bound by $\exists$-name will eventually return from the remote site without being touched by the receiver).
   - ($\lambda\pi^{\mathsf{LAM}}_{\leq}$: the linear/affine $\lambda\pi$ with subtyping and state) A new formulation of a subtyping system based on branching constructors [6, 19, 63, 66] and extensions of the above results to stateful, concurrent and non-deterministic computation.

3. The presentation of new proof techniques for establishing the key properties of the dependently typed $\lambda\pi$ and its integration with linear/affine types, combining the techniques from [6, 27, 62, 63, 65, 66] and a novel type-directed encoding of the higher-order mobile processes into the π-calculus.

Throughout the paper we illustrate the expressiveness of the type structures of the $\lambda\pi_{\leq}^{\mathsf{LAM}}$ by many examples, including an embedding of basic primitives for code distribution in the calculus.

OVERVIEW OF PART II AND PART III To illustrate possible applications of the basic theory developed in Part I, the sequels of this paper, Part II and III, present non-trivial type-based analyses of security in distributed computation based on the typed $\lambda\pi$. We give a brief summary of their content below. First, in Part II, we present type-based analyses of two basic security concerns in mobile code, secrecy [2, 14, 27, 39, 52] and access control of resources [3, 16, 17, 44, 58, 59], both using the typed formalism. We prove noninterference and authority-error freedom for well-typed processes in the presence of higher-order code mobility. For secrecy, we reduce the analysis to the one in the corresponding typed π-calculus, using the type-directed encoding mentioned above, leading to non-interference. For access control, we develop a simple but expressive framework for operational semantics with dynamic change of access rights, accompanied by a basic type discipline which ensures the lack of authorisation error of resources. Part III applies these analyses to design a simple multi-threaded class-based language with distributed communication and code mobility [8, 27, 31, 56]. The type disciplines for secrecy and access control are developed on the basis of the $\lambda\pi$-calculus in Part II. Correctness (type-safety, non-interference and the lack of authority error) of the language is proved via the type-preserving embedding into the $\lambda\pi$ in Part II.

STRUCTURE OF THE PAPER Part I gives a full technical development of the core theory outlined in a short paper found at `http://www.doc.ic.ac.uk/˜yoshida` [67]. In the remainder, Section 2 introduces the $\lambda\pi$ with channel dependent types. Section 3 studies the existential types. Section 4 describes two examples which illustrate the expressiveness of channel dependent/existential types of $\lambda\pi$. Section 5 proposes an affine higher-order typing system. Section 6 formulates the linear higher-order typing system and proves the strong normalisation via linearity preserving translation. Section 7 integrates the affine and linear typing systems, proves two kinds of liveness properties and shows the extension with subtyping, state, concurrency and location. Section 8 illustrates the expressiveness of the extended dependently typed $\lambda\pi$ using non-trivial examples, including embedding of language primitives for dynamic code distribution. Section 9 concludes the paper with discussions on the related work and further issues.

2 Processes and Types

2.1 Processes

The syntax of processes [64, 65] 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 $x, y, \ldots$

(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)

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

(Value)

$V, W, \ldots ::=$	$u, v, w, ..$	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
$\overline{u}(c)P$	$\stackrel{\text{def}}{=} (\nu c)(\overline{a}\langle c \rangle \mid P)$		bound output

FIGURE 1. Syntax

We denote $\vec{V}$ for a potentially empty vector $V_1 ... V_n$. τ and σ denote value and channel types which will be given later. Type annotations are often omitted unless they are relevant.

The reduction semantics is defined in Figure 2; these rules simply combine those of the π-calculus and the call-by-value λ-calculus. It uses the standard structural equivalence $\equiv$ of the π-calculus; the axioms for $\equiv$ are given in Figure 2 where $\equiv_\alpha$ is a standard α-equality and $\mathsf{fn}(P)$ denotes a set of free names of P given later. We also denote multi-step reduction as $\twoheadrightarrow \stackrel{\text{def}}{=} \equiv \cup \longrightarrow^*$. The main rules are β-reduction (β) and communication (com). Note that substitution by channels $P\{u/x\}$ affects not only terms but also types which annotate bound variables. We shall give the formal definition of the substitution after introducing types.

2.2 Types

Syntax

The grammar of types are given in Figure 3. These types are from [64, 65] except for the introduction of channel dependent types. We use a simpler form of IO-modes based on duality [6, 23, 24, 27, 63]. There is unit, nat, .., higher-order function type $\tau \to \tau'$, (functional) name dependent type $\Pi(x : \sigma)\tau$, process type Δ and channel type σ. The last one incorporates dependent quantification on which we shall discuss soon. In $\Pi(x : \sigma)\tau$, we allow the type τ to contain occurrences of the channel variable x; then x in τ is bound. A process type Δ, assigned to a process, is a mapping from a finite subset of $\mathbf{N} \cup \mathbf{V}$ to channel types. Unlike in [64, 65], we no longer have proc types, which are now unnecessary due to the introduction of channel dependent types.

(Reduction)

$$(\beta)\quad (\lambda(x\!:\!\tau)P)\,V \longrightarrow P\{V/x\} \qquad (\text{app}_r)\ \frac{P \longrightarrow P'}{PQ \longrightarrow P'Q} \qquad (\text{app}_l)\ \frac{Q \longrightarrow Q'}{VQ \longrightarrow VQ'}$$

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

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

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

(Structure Equivalence)

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

FIGURE 2. Reduction

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 3, 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 $\mathtt{nat} \to \mathtt{nat}$ 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). The set of free names and variables in Figure 4. 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 $\mathsf{fv}(\lambda(x\!:\!\tau).P) = (\mathsf{fv}(\tau) \cup \mathsf{fv}(P)) \setminus x$.

Then following [6, 23, 24, 27, 63], we define duality of types and a partial algebra on modes/types in order to control composability of processes. By refining these operators, we can give more expressive notions of type constraints. We define:

Value: τ ::= $\texttt{unit} \mid \texttt{nat} \mid \tau \to \tau' \mid \Pi(x{:}\sigma)\tau \mid \Delta \mid \sigma$

Channel: σ ::= $(\Pi[x_1 : \sigma_1, ..., x_n : \sigma_n]\tau_1, ..., \tau_m)^p \mid \langle \sigma, \sigma' \rangle$

Environment: Δ ::= $\emptyset \mid \Delta, x{:}\tau \mid \Delta, u{:}\sigma$

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

(if $x_1, ..., x_n \notin \mathsf{fv}(\sigma_1, ..., \sigma_n, \tau_1, ..., \tau_m)$)

Thunk: $\ulcorner \Delta \urcorner \stackrel{\text{def}}{=} \texttt{unit} \to \Delta$

FIGURE 3. Types

(Free Variables)

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

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

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

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

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

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

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

(Free Names)

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

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

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

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

(Free Names and Variables)

$\mathsf{fnv}(\tau) = \mathsf{fn}(\tau) \cup \mathsf{fv}(\tau)$

FIGURE 4. Free Variables and Names on Types

$$
\begin{array}{rcl}
\texttt{unit}\{V/x\} & = & \texttt{unit} \\
\texttt{nat}\{V/x\} & = & \texttt{nat} \\
(\tau \to \tau')\{V/x\} & = & \tau\{V/x\} \to \tau'\{V/x\} \\
(\Pi(y{:}\sigma)\tau)\{V/x\} & = & \Pi(y{:}\sigma\{V/x\})\tau\{V/x\} \quad (x \neq y) \\
\{u_1 : \sigma_1, ..., u_n : \sigma_n\}\{V/x\} & = & \{u_1\{V/x\} : \sigma_1\{V/x\}\} \odot \cdots \odot \{u_n\{V/x\} : \sigma_n\{V/x\}\} \\
(\Pi[x_1 : \sigma_1, ..., x_n : \sigma_n]\tau_1, ..., \tau_m)^p\{V/x\} & = & (\Pi[x_1 : \sigma_1\{V/x\}, ..., x_n : \sigma_n\{V/x\}]\tau_1\{V/x\}, ..., \tau_m\{V/x\})^p \\
 & & (x \notin \{x_1, ..., x_n\}) \\
\langle \sigma, \sigma' \rangle\{V/x\} & = & \langle \sigma\{V/x\}, \sigma'\{V/x\} \rangle
\end{array}
$$

FIGURE 5. Substitution into Types

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

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

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

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

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

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

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 3. $(\mathtt{nat})^{\mathtt{I}}$ 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 [64, 65], 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. Formally a partial commutative operator $\odot$ on types is defined as:

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

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

As a simple 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. We define: $\Delta_1 \asymp \Delta_2$ if $x : \sigma_1 \in \Delta_1$ and $x : \sigma_2 \in \Delta_2$ imply $\sigma_1 \odot \sigma_2$ is defined. If $\Delta_1 \asymp \Delta_2$, then we set:

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

Note this operation $\odot$ corresponds to the least upper bound operator $\sqcup$ in [65]. Using this operator, the substitution is formally defined in Figure 5.

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

(Well-Formed Environment)

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

(Well-Formed Types)

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

$$\text{(t-emp)}\ \frac{\Gamma \vdash \texttt{Env}}{\Gamma \vdash \emptyset : \texttt{tp}_P} \qquad \text{(t-pro)}\ \frac{\Gamma \vdash \Delta : \texttt{tp}_P \quad \Gamma \vdash u : \sigma \quad u \notin \mathsf{dom}(\Delta)}{\Gamma \vdash \Delta, u{:}\sigma : \texttt{tp}_P}$$

$$\text{(t-input)}\ \frac{\Gamma, x_1{:}\sigma_1, .., x_n{:}\sigma_n \vdash \tau_j : \texttt{tp}_H}{\Gamma \vdash (\Pi[x_1{:}\sigma_1, ..., x_n{:}\sigma_n]\tau_1, ..., \tau_m)^{\mathtt{I}} : \mathtt{I}} \qquad \text{(t-dual)}\ \frac{\Gamma \vdash \sigma : \mathtt{I}}{\Gamma \vdash \overline{\sigma} : \mathtt{O}} \qquad \text{(t-pair)}\ \frac{\Gamma \vdash \sigma : \mathtt{O}}{\Gamma \vdash \langle \sigma, \overline{\sigma} \rangle : \mathtt{IO}}$$

FIGURE 6. Well-Formed Higher-Order IO Types

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

while $\{a{:}(\texttt{nat})^{\mathtt{O}}, x{:}(\texttt{unit})^{\mathtt{I}}\}\{a/x\}$ is undefined. We can apply the same rule to the substitution into terms [65, §.2]. For example, given $\lambda(x{:}(\texttt{nat})^{\mathtt{O}}).\lambda(y{:}\ulcorner x{:}(\texttt{nat})^{\mathtt{O}}, b{:}(\texttt{unit})^{\mathtt{O}}\urcorner).y$ we can observe:

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

Our typing system shall ensure such an undefined substitution never happens during interaction between typable terms. We observe:

PROPOSITION 2.2. (decidability of $\asymp$ and substitution)

1. *Given σ_1 and σ_2, it is decidable whether $\sigma_1 \asymp \sigma_2$.*
2. *Given Δ_1 and Δ_2, it is decidable whether $\Delta_1 \asymp \Delta_2$.*
3. *Given τ, u and x, it is decidable whether $\tau\{u/x\}$ is defined or not.*

PROOF. By induction on σ_i, Δ_i and τ. See Appendix A.1. □

2.3 Well-Formed Types

We define the notions of well-formedness for types and environments by the formal system in Figure 6 where we use the following four forms of judgements:

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

$\texttt{tp}$ stands for either $\texttt{tp}_H$ or $\texttt{tp}_P$ and J stands for any judgement. For simplicity, we omit subtyping rules from [65], but (Weak) in the main typing rule in Figure 7 essentially plays the same rôle as process subsumption. As in [65], the first judgement $\Gamma \vdash \texttt{Env}$ is designed to ensure that an identifier can only be used in the construction of a type if it has already been *declared* in the environment. For example, environments such as $\{y : \ulcorner a{:}(\texttt{nat})^{\mathtt{O}}\urcorner\}$

(a is not declared) and $\{y : \ulcorner x : (\mathtt{nat})^{0}\urcorner,\ x : (\mathtt{nat})^{0}\}$ (x used by y is declared later) are not well-formed. The formation rule for functional types are the same as [65]; (t-base) is for constant and (t-abs$_H$) is for higher-order values. In the formation rule for dependent types, (t-abs$_N$), the bound variable x is allowed in the construction of the result type τ. See [65, § 3.1] for more explanation.

For process types, the second condition $\Gamma \vdash u : \sigma$ in (proc-t) (which means either $\Gamma(u) = \langle\sigma, \sigma'\rangle$ or $\Gamma(u) = \sigma$ together with (Chan,Pair) in Figure 7) ensures that a process always has a channel environment which does not exceed the capability specified by the global environment Γ. In channel types, $x_1, ..., x_n$ bind τ_j and $x_1, ..., x_{i-1}$ bind σ_i from the formation rule. A pair type consists of input and its dual output.

As in [65, Lemma 3.1], for well-formed judgement, we can prove the same basic consistency lemmas such as Weakening and Narrowing for the base environment Γ. We only list the key lemma, which is important to compare with the affine/linear systems introduced later.

LEMMA 2.3.

1. (bounded weakening) *Assume* $\Gamma, u : \sigma, \Gamma' \vdash \mathtt{J}$. *Then* $\Gamma, u : \langle\sigma, \overline{\sigma}\rangle, \Gamma' \vdash \mathtt{J}$.
2. (narrowing) *Assume* $\Gamma, u : \sigma, \Gamma' \vdash \Delta, u : \sigma'$ *with* $u \notin \mathsf{fnv}(\Gamma') \cup \mathsf{fnv}(\Delta)$. *Then* $\Gamma, \Gamma' \vdash \Delta$.
3. (name substitution) *Suppose* $\Gamma \vdash u : \sigma$. *Then* $\Gamma, x : \sigma, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma, \Gamma'\{u/x\} \vdash \mathtt{J}\{u/x\}$.
4. (process types) $\Gamma \vdash \Delta_{1,2} : \mathtt{tp}_P$ *implies* $\Gamma \vdash \Delta_1 \odot \Delta_2 : \mathtt{tp}_P$.

PROOF. The proof follows the routine in [65, Lemma 3.1]. (1) and (2) are straightforward. For (3), we prove the case that the last applied rule is (input). Assume $\Gamma \vdash u : \sigma$. Suppose, by (input), we infer:

$$\Gamma, x : \sigma, \Gamma' \vdash (\Pi[x_1 : \sigma_1, ..., x_n : \sigma_n]\tau_1, ..., \tau_m)^{\mathtt{I}} : \mathtt{I}$$

from the assumption $\Gamma, x : \sigma, \Gamma', x_1 : \sigma_1, .., x_n : \sigma_n \vdash \tau_j : \mathtt{tp}_H$ for each j. By induction (on the size of types and environments, cf. Appendix A.1), we have:

$$\Gamma, \Gamma'\{u/x\}, x_1 : \sigma_1\{u/x\}, .., x_n : \sigma_n\{u/x\} \vdash \tau_j\{u/x\} : \mathtt{tp}_H$$

for each j. Now by applying (input) to the above again, we have:

$$\Gamma, \Gamma'\{u/x\} \vdash (\Pi[x_1 : \sigma_1\{u/x\}, ..., x_n : \sigma_n\{u/x\}]\tau_1\{u/x\}, ..., \tau_m\{u/x\})^{\mathtt{I}} : \mathtt{I}$$

The case that the last applied rule is (dual) or (pair) is obvious by induction. Other cases are the same as the proof in [65, Lemma 3.2]. For the proof of (2), we have to prove $\Delta_1(u) \asymp \Delta_2(u)$ for each $u \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)$. Suppose $\Gamma(u) = \langle\sigma, \overline{\sigma}\rangle$ for some σ. The case $\Gamma(u)$ is not a pair type is simpler. Note that $\Gamma \vdash u : \sigma'$ implies σ' is either $\langle\sigma, \overline{\sigma}\rangle$, σ or $\overline{\sigma}$. Hence, by **(t-1,2)** $\Gamma \vdash u : \sigma'_i$ implies $\sigma'_1 \odot \sigma'_2$ is always defined, thus we have $\sigma'_1 \asymp \sigma'_2$, as required. □

In any derivation rule, the size of hypothesis is strictly smaller than that of the conclusion. Hence we have:

(Common)

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

(Function)

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

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

(Process)

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

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

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

FIGURE 7. Higher-Order IO Typing System

PROPOSITION 2.4. *It is decidable whether, given an environment* Γ, *the judgement* $\Gamma \vdash \mathtt{Env}$ *is derivable. Also it is decidable whether, given an environment* Γ *and* τ, *the judgement* $\Gamma \vdash \tau : \mathtt{tp}$ *is derivable.*

PROOF. See Appendix A.1. □

2.4 Typing with Channel Dependent Types

The typing system for $\lambda\pi$ is given in Figure 7, 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

The inference rules, coming from [65] 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 [65]; 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

[1] In [65], 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.

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.

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

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, cf. Lemma 2.7 (2)). Note that the resulting type $\Delta, u:\sigma$ means $\Delta \odot u:\sigma$ allowing $u \in \mathsf{fnv}(\Delta)$. Finally we notice that ($\mathsf{In}^!$) is derived from (In).

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

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

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 [6, 27, 63, 66].
- 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).

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. Note for the polyadic case, we assume $\sigma_i = \sigma_j$ if $v_i = v_j$ in (Out) in Figure 7.

Below we show an example of inference which uses channel dependent type non-trivially.

EXAMPLE 2.5. 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:

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

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 3. Then we have:

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

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 2.6. 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:

$\Delta \vdash P \,|\, \overline{c}\langle v, \ulcorner\overline{v}\langle 2\rangle\urcorner\rangle : \Delta$, which reduces to $\overline{v}\langle 2\rangle \,|\, \overline{v}\langle 1\rangle$; and
$\Delta' \vdash P \,|\, \overline{c}\langle v, \ulcorner\overline{b}\langle 2\rangle\urcorner\rangle : \Delta'$, 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; but
$(\nu b)(\overline{c}\langle v, \ulcorner\overline{b}\langle 2\rangle\urcorner\rangle \,|\, !b(z : \mathtt{int}).R_z)$ is not.

We can easily check the former can be inferred by (Res) after (Par) as follows.

$$\dfrac{\dfrac{\Gamma \vdash \overline{c}\langle v, \ulcorner\overline{v}\langle 2\rangle\urcorner\rangle : \{c{:}\overline{\sigma_c}, v : (\mathtt{int})^{\mathtt{O}}\} \quad \Gamma \vdash !v(z).R_z : v : (\mathtt{int})^{\mathtt{I}}}{\Gamma \vdash \overline{c}\langle v, \ulcorner\overline{v}\langle 2\rangle\urcorner\rangle \,|\, !v(z).R_z : \{c{:}\overline{\sigma_c}, v{:}\sigma_v\}}\ \text{(Par)}}{\Gamma/v \vdash (\nu v)(\overline{c}\langle v, \ulcorner\overline{v}\langle 2\rangle\urcorner\rangle \,|\, !v(z).R_z) : c{:}\overline{\sigma_c}}\ \text{(Res)}$$

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.

In Example 4.2, we also show the usage of the channel dependent type in which free occurrences of x_1 in the second channel type σ_2 are bound by the dependent quantifier, $(\Pi[x_1\!:\!\sigma_1]\sigma_2)^{\mathrm{I}}$.

2.5 Basic Properties on Types

We list basic syntactic properties of typed processes. We start from the following key lemma.

LEMMA 2.7.

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

PROOF. For (1), the proof is by constructing the syntax direct minimum typing system where we do not use the weakening rule (see Appendix A.2). (2) is from (1). (3) just follows the routine in [65, Lemmas 5.1 and 5.2], using Proposition 2.3, see Appendix A.3. □

Now we have:

THEOREM 2.8.

1. (struct) *Suppose* $\Gamma \vdash P : \tau$ *and* $P \equiv P'$. *Then* $\Gamma \vdash P' : \tau$.
2. (type safety) *Suppose* $\Gamma \vdash P : \tau$ *and* $P \twoheadrightarrow P'$. *Then* $\Gamma \vdash P' : \tau$.

PROOF. For (1), we use Lemma 2.7 (2). We only show the following key scope opening rule

$$(\nu\, a\!:\!\sigma)P\,|\,Q \;\equiv\; (\nu\, a\!:\!\sigma)(P\,|\,Q) \;\; \text{if } a \notin \mathsf{fn}(Q)$$

We prove $\Leftarrow$ direction. $\Rightarrow$ is easier. Suppose $\Gamma \vdash (\nu\, a\!:\!\sigma)(P\,|\,Q) : \Delta$ with $a \notin \mathsf{fn}(Q)$. Then this is inferred from $\Gamma, a\!:\!\sigma \vdash P : \Delta_1$ and $\Gamma, a\!:\!\sigma \vdash Q : \Delta_2$ for some $\Delta_1 \odot \Delta_2 \subseteq \Delta, a\!:\!\sigma$. W.o.l.g. we assume $\Delta_i = \Delta'_i, a\!:\!\sigma_i$ and $\sigma_1 \odot \sigma_2 = \sigma$.

First by applying (Weak) to P, we have $\Gamma, a\!:\!\sigma \vdash P : \Delta'_1, a\!:\!\sigma$. Then by Lemma 2.7 (2), we know $a \notin \mathsf{fnv}(\Delta'_i)$. Hence an application of (Res) gives us: $\Gamma \vdash (a\!:\!\sigma)P : \Delta'_1$. We also can derive $\Gamma \vdash Q : \Delta'_2$ by Lemma 2.7 (1,2). Note that $\Delta_1 \asymp \Delta_2$ implies $\Delta'_1 \asymp \Delta'_2$. Hence by applying (Par) and (Weak), we conclude the proof. The proof (2) essentially follows the same routine in [65]. See Appendix A.4. □

REMARK 2.9. The use of dependent channel types simplifies the proofs of subject reduction (as well as substitution lemmas) in [65].

One of the important properties of the present system is existence of the best possible type (minimum interface type) for each typable process. Not only does this make type checking decidable, but also it indicates a proper qualitative difference which channel dependent/existential types induces in comparison with the original system in [65]. Below $\Delta_1 \subseteq \Delta_2$ 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 $\sigma = \Delta_1(u)$.

PROPOSITION 2.10. (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. (the maximum) $\Gamma \vdash P : \Delta_{max}$ *with* $\Delta_{max} = \{u : \sigma \mid u : \sigma \in \Gamma\}$*.*

PROOF. The proof is by constructing the minimum typing system where we apply weakening rules just before (Res) and (In) as much as we need to obtain the latter. See Appendix A.2. (2) is obvious by (Weak) together with (1). □

The above properties do not hold in the original system [65] (for example, we can take $c(x,y).(run\ y \,|\, \overline{x}\langle 1 \rangle)$ with type σ_c in Example 2.6).

COROLLARY 2.11.

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* $\Gamma \vdash P : \tau$

PROOF. (1) is straightforward from Propositions 2.2, 2.4 and 2.10 (1). (2) is obvious from Proposition 2.10 (2), together with (1). □

From the above (1), we can easily construct the algorithm to compute $\Gamma \vdash P : \tau$ from given Γ and P, cf. [64, § 5.3, Figure 11 in Appendix C]. Next we examine the relationship between process types and process behaviour. Define:

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

Dually we define $P \Downarrow a^{\mathtt{I}}$. Below $\mathsf{md}(\sigma)$ denotes the mode of σ. The proof follows [65, Proposition 5.2].

PROPOSITION 2.12. *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* $\mathtt{O}$*.*

Note that this proposition says an interface can ensure the limit of accessibility, but not availability of resources: $\Gamma \vdash P : \Delta$ with $\mathsf{md}(\Delta(a)) = p$ does not usually imply $P \Downarrow a^p$, cf. Proposition 5.3, Theorem 6.3 and Proposition 7.2.

(Well-Formed Environment and Types)

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

(Typing System)

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

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

Other rules are from Figure 7.

FIGURE 8. Existential Typing System

3 Channel Existential Types

3.1 Basic Idea

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 Example 2.5 in the previous section where c has type $(\Pi[x{:}(\mathtt{int})^{\mathtt{O}}]\ulcorner x{:}(\mathtt{int})^{\mathtt{O}\urcorner})^{\mathtt{I}}$:

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

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

$$c(y).(y() \,|\, \overline{a}\langle y\rangle) \,|\, (\nu\, v)(\overline{c}\langle V\rangle \,|\, !v(z).R) \tag{11}$$

Note the name v is not passed in communication, so that it is not used as such in the receiving process. This way of name usage can faithfully capture scope opening associated with higher-order communication in the $\lambda\pi$-calculus, and is formalised by introducing *a channel existential type*, written $\exists[x:\sigma]\tau$, which functions similar to the existential type $\exists\mathtt{X.T}$ of the polymorphic λ-calculi [42]; the type $\exists\mathtt{X.T}$ represents a pair, written $\{*\mathtt{S},\mathtt{t}\}$, of a type $\mathtt{S}$ and a term $\mathtt{t}$ of type $\mathtt{S}\{\mathtt{T}/\mathtt{X}\}$; in $\lambda\pi$, names are used as types, so that $\exists[x:\sigma]\tau$ is a representation of a pair, written $\{v,V\}$, of a type σ and a term V of type $\tau\{v/x\}$. In the polymorphic λ-calculi, a value $\{*\mathtt{S},\mathtt{t}\}$ of type $\exists\mathtt{X.T}$ forms a package with one hidden representation type. In $\lambda\pi$, the concrete institution is to think of $\{v,V\}$ of type $\exists[x:\sigma]\tau$

as a form of package with one hidden representation name. This package is opened via interaction between input and output processes.

Many examples of higher-order code mobility can be typed using only channel existential types.[2] 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.

3.2 Existential Typing

First we extend the syntax of types in Figure 3 as follows.

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

$\exists[x\!:\!\sigma]\tau$ is an existential type with name x and value type τ (its binding/free names/variables are defined as $\Pi[x\!:\!\sigma]\tau$). We also use $\{v, V\}$, a pair of name v and a higher-order value V, which appears only during type inference.[3]

The extended rules for well-formedness and typing are listed in Figure 8. In the environment, $\{x\!:\!\sigma, y\!:\!\tau\}$ guarantees that x only appears in τ, i.e. its scope is in τ. In the typing rule, (Tupl∃)/(Val∃) is an introduction of a pair of variables/a variable. (Pack) constructs a pack. (In∃) and (Out∃) can be regarded as a special case of (In) and (Out) of the dependent typing system in Figure 7 where scope of x is restricted in τ.

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

EXAMPLE 3.1. (existential typing) We will type the term

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

in (11). First let $\sigma_a = (\exists[x\!:\!\overline{\sigma_x}]\ulcorner x\!:\!\sigma_x\urcorner)^0$, $\sigma_c = \langle \sigma_a, \overline{\sigma_a}\rangle$, $V = \ulcorner \overline{v}\langle 2\rangle \urcorner$ and $\sigma_x = (\mathtt{int})^0$. We also assume $!v(z).R$ has type $v\!:\!(\mathtt{int})^{\mathtt{I}}$. Then by (Var∃) and (App^H), we have:

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

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

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

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

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

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

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

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

---

[2]The generality of channel dependent types becomes important in some significant theoretical and pragmatic settings, as will be discussed elsewhere.

[3]For simplicity, in the existential typing system here, we only consider a single name and a single type, which can be easily generalised to the arbitrary number of names and/or types as $\exists[\vec{x}\!:\!\vec{\sigma}]\vec{\tau}$.

3.3 Properties with Existential Types

We first list basic lemmas for existential types which are different from those for the channel dependency types. Note in (6), existential name v is only substituted to type τ, not to term P (cf. Lemma 2.7 (3)).

LEMMA 3.2.

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

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

THEOREM 3.3. (type safety for existential types) $\Gamma \vdash P : \tau$ *and* $P \twoheadrightarrow P'$ *imply* $\Gamma \vdash P' : \tau$.

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

The same basic propositions in the previous section (Propositions 2.10, 2.11,2.12) hold for existential types. In the following, we prove the type erasure theorem, which shows safe mobile-code can be soundly executed by untyped evaluation strategy. The syntax and operational semantics of untyped $\lambda\pi$-calculus are defined as in Figure 2 erasing type annotations in bound variables and names.

DEFINITION 3.4. *The* erasure *of an IO-typed term is defined as follows.*

$$\mathsf{Erase}(\mathbf{0}) = \mathbf{0} \quad \mathsf{Erase}(P\,|\,Q) = \mathsf{Erase}(P)\,|\,\mathsf{Erase}(Q) \quad \mathsf{Erase}((\nu\, a{:}\sigma)P) = (\nu\, a)\mathsf{Erase}(P)$$
$$\mathsf{Erase}(\overline{u}\langle V_1,...,V_n\rangle) = \overline{u}\langle \mathsf{Erase}(V_1),...,\mathsf{Erase}(V_n)\rangle$$
$$\mathsf{Erase}(u(x_1{:}\tau_1,...,x_n{:}\tau_n).P) = u(x_1,...,x_n).\mathsf{Erase}(P)$$
$$\mathsf{Erase}(!\,u(x_1{:}\tau_1,...,x_n{:}\tau_n).P) = !\,u(x_1,...,x_n).\mathsf{Erase}(P)$$
$$\mathsf{Erase}(P\,Q) = \mathsf{Erase}(P)\mathsf{Erase}(Q) \quad \mathsf{Erase}(\lambda(x{:}\tau).P) = \lambda x.\mathsf{Erase}(P)$$
$$\mathsf{Erase}(u) = u \quad \mathsf{Erase}(()) = () \quad \mathsf{Erase}(1) = 1$$

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

PROPOSITION 3.5.

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

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

REMARK 3.6.

- In $(\mathsf{Tpl}\exists)$ and (Pack), we do not explicitly declare a type annotation for a package as "$\{v,V\}$ `as` $\exists[x:\sigma]\tau$", which is necessary for the (explicitly typed) existential polymorphic λ-calculus in order to guarantee uniqueness of the type of the package [42, §. 24.1]. This is because in our typing system, this information is obtained from a channel type $(\exists[x:\overline{\sigma}]\tau)^{0}$ in an environment Γ when a package is actually used in $(\mathsf{Out}\exists)$. Hence given Γ, we can ensure that there is exactly one unique minimum type τ for term P as proved in Proposition 2.10.
- We also only consider a higher-order value V in $\{v,V\}$; it can be also generalised to V as channel without any difficulty (we only need to replace τ in (t-exist) to σ' and prove $\Gamma,\Gamma'\{V/y\} \vdash P\{V/y\} : \tau'\{v/x\}\{V/y\}$ in the conclusion of Lemma 3.2 (6)). We can also arbitrarily combine the existential and dependent quantifiers in one channel type as $(\exists[\vec{x}:\vec{\sigma}]\Pi[\vec{y}:\vec{\sigma}']\vec{\tau})^{p}$. It can be also extended to "$\{v,P\}$ `as` $\exists[x:\sigma]\tau$" and `let` $\{y,z\} = P$ `in` P by adding the reduction rules for packages and `let` as the standard existential typing system [42].

4 Further Examples of Dependent/Existential Types

This section presents two larger examples which use new dependent/existential types of the $\lambda\pi$.

4.1 Remote Procedure Migration

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:

$$\mathtt{Server}(l) \stackrel{\text{def}}{=} (\nu c)(!l(y:\tau).(yc) \mid !c(x).\overline{x}\langle n\rangle) \tag{12}$$

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

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

This function has the following functional name dependent type:

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

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

$$\mathtt{let}\ x = \overline{l}\langle\mathtt{twice}\rangle\ \mathtt{in}\ R \stackrel{\text{def}}{=} (\nu e)(\overline{l}\langle\mathtt{return\ twice\ to}\ e\rangle \mid e(x).R) \tag{13}$$

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

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

The example shows that the combination of types in the $\lambda\pi$ offers:

(1) protection for the server against the client (by τ, which may be checked at the time of arrival of the code)
(2) the invisibility (hence inaccessibility) of the local service point c in the server to the client;
(3) flexible programming by the client (i.e. it can easily replace `twice` by `thrice`, such as $\overline{c}(f)f(z_1).\overline{c}(f)f(z_2).\overline{c}(f)f(z_3).\overline{e}\langle z_1+z_2+z_3\rangle)$;
(4) ensuring that the returned message to e has not been touched (by existential type $\exists[e\,{:}\,(\mathtt{nat})^{0}]\tau'$, the server will not touch $\overline{e}\langle z_1+z_2\rangle$ if it is well-typed);
(5) safe return to the destination specified by the client (again by existential type).

These features are the key to the applications later.

4.2 Remote Forwarding and Filtering

To demonstrate the power of dependent and existential channel types, we type remote-forwarding and filtering, which are similar to those formalised by combining dynamic and static type-checking in [45]. Our fine-grained types can provide a powerful and flexible host-based protection mechanism, retaining a static checking framework.

Suppose thunked process $\ulcorner P \urcorner$ is passed to remote site l. This can be simply written as $\overline{l}\langle\ulcorner P\urcorner\rangle$ in $\lambda\pi$, which corresponds to $\mathtt{goto}\; l.P$ using the notation in [45]. Using our process types, the receiver site can restrict the usage of migrating code as:

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

Here $!l(x\,{:}\,\ulcorner\Delta_l\urcorner).\,run\,x$ behaves as a filter of location l against which incoming code is compared. We write this filter as $l\langle\!\langle\Delta_l\rangle\!\rangle$ following the notation in [45]. If $\Delta=\{a\,{:}\,(\mathtt{nat})^{0}\}$, accepted code can access only resource $!a(x).R_a$ but has no permission to access $!b(x).R_b$.

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

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

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

$$\{l\,{:}\,(\ulcorner a\,{:}\,(\mathtt{nat})^{0}\urcorner)^{0},\; b\,{:}\,(\mathtt{nat})^{0}\}$$

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

The next example shows our channel dependency type can represent the similar mechanism ensured by the location dependency type in [45]. Let us consider the following simple forwarder

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

which inputs two names x and y and goes to the location y at which x is located. Then b

has a channel dependent type as follows:

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

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

Next let us consider another forwarder

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

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

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

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

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

This type means that location k can accept a process which interacts port b and location l which contains *anonymous* resource which carries `nat` type. Another example which is not statically typable in the system of [45] is:

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

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

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

Note l has a type which contains an output of `nat` type with *anonymous* name x. [20] also shows functional and channel dependent types as well as existential types can provide a static host protection mechanism in a non-trivial higher-order extension of Dπ.

At this moment, types cannot specify causality (timing) of access control; one may wish to describe "move location k, get b then go back to l, and eventually access a" as types. To describe such a specification, we start integrating the dependent typing system with affine/linear typing disciplines from the next section.

5 Affine Type Discipline

This section and the next two integrate the affine/linear type disciplines with the dependently typed $\lambda\pi$. This leads to the following three kinds of typed calculi.

- $\lambda\pi^{\mathsf{A}}$ (the affine $\lambda\pi$-calculus), discussed in § 5 (this section).
- $\lambda\pi^{\mathsf{L}}$ (the linear $\lambda\pi$-calculus), discussed in § 6.
- $\lambda\pi^{\mathsf{LA}}$ (the linear/affine $\lambda\pi$-calculus) and $\lambda\pi^{\mathsf{LAM}}_{\leq}$ (the linear/affine $\lambda\pi$-calculus with subtyping and state), both discussed in § 7.

The integration is done systematically as follows.

- Refining the algebraic laws of compositions of modes and channels, i.e. $\odot$; and
- Replacing the typing rules to add further constraints on process formation.

(Well-Formed Types)

$$(\text{t-abs}_N)\ \frac{\Gamma, x:\sigma_0 \vdash \tau:\mathtt{tp}}{\Gamma \vdash \Pi(x:\sigma_0)\tau:\mathtt{tp}_H} \qquad (\text{t-pros})\ \frac{\Gamma \vdash ?\Delta:\mathtt{tp}_P \quad \Gamma \vdash u:\sigma_0 \quad u \notin \mathsf{dom}(\Delta)}{\Gamma \vdash \Delta, u:\sigma_0:\mathtt{tp}_{PS}}$$

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

Others are from Figure 6 (in (t-dual,pair), we replace $\mathtt{I}$ by $p_{\mathtt{I}}$, $\mathtt{O}$ by $p_{\mathtt{O}}$ and $\mathtt{IO}$ by $p_{\mathtt{I}} \odot p_{\mathtt{O}}$; $\mathtt{tp}$ stands for either $\mathtt{tp}_H$ or $\mathtt{tp}_{PS}$).

(Typing System)

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

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

Others are from Figure 7 (in (Out), we replace $\mathtt{O}$ by $p_{\mathtt{O}}$)

FIGURE 9. Affine Higher-Order Typing System

For brevity hereafter we consider only `unit` as a base type. Since the natural numbers can be encoded into branching and selection introduced in $\lambda\pi^{\text{LAM}}_{<}$ in § 7.4 [6, 27, 63, 66], this leads to no loss of generality.

5.1 Affine Types

In [6], we have presented the affine π-calculus (π^{A}) can precisely characterise sequential higher-order functional behaviours, by means of the full abstract embedding of PCF/PCFv. The term *affinity* denotes possibly diverging behaviour in which a question is given an answer at most once. The affine typing system for $\lambda\pi^{\text{A}}$ is defined by integrating π^{A}; with this refinement, we can guarantee availability of server resources and linear usage of channels, together with locality of channels [48, 64].

The syntax of terms and types remains as the same as Figures 1 and 3 except the modes p in channel types. We first extend the set of modes as follows (we omit script A from modes until we introduce the mixed calculus, $\lambda\pi^{\text{LA}}$ in § 7).

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

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

$\mathsf{md}(\sigma) = p_{\mathrm{I}}$, similarly for $\sigma_{\mathtt{O}}$ and $\sigma_{\mathtt{IO}}$.

The rules **(m-1)** and **(m-2)** in § 2 are refined to:

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

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

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

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

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

Note other compositions of types are undefined. For example, $!x.\mathbf{0} \,|\, !x.\mathbf{0}$ is never typable because $()^! \odot ()^!$ is undefined, while $\overline{x} \,|\, \overline{x}$ is typable by $x : ()^?$ but not by $x : ()^\uparrow$, and $!x.\mathbf{0} \,|\, \overline{x} \,|\, \overline{x}$ by $x : \langle ()^!, ()^? \rangle$. This partial algebra of channel types ensures, among others, determinacy of computation in typable processes by controlling their composability. Hence the channel with ? can appear arbitrary times in processes, while others cannot. The definitions of the operator $\Delta_1 \odot \Delta_2$ and $\asymp$ remain as the same.

The new judgements and typing rules are listed in Figure 9. Here $\vec{p}\Delta$ indicates $\mathsf{md}(\Delta) \subseteq \{\vec{p}\}$ and Δ^{-u} means $u \notin \mathsf{fnv}(\Delta)$. For well-formedness, we add the following three main constraints;

Locality channels/values without input capability are instantiated and sent around via channels [48, 64] (guaranteed by $\sigma_{\mathtt{O}}$ in (t-abs$_N$) and (t-proc))

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

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

The first constraint is now standard in the literature for a natural representation of distributed computation [5, 15, 48, 53, 64]. The last two conditions, which were inherited from [6], had come form game semantics [4, 26, 28]. The resulting calculus is an integration of the local higher-order π-calculus [48, 64] and the affine π-calculus [6]. Since each original calculus can embed the λ-calculus with recursion, it does not lose expressiveness.

5.2 Affine Typing

The typing system for the affine $\lambda\pi$ is defined in Figure 9. We just need to replace **(Process)** in IO-rules in Figure 7 to those of the π^{A}-calculus [6]. In (Par), we need to check the compositionality of P_1 and P_2 by checking $\Delta_1 \asymp \Delta_2$ by using the laws **(a-t-1,2)**. Note that the definition of $\asymp$ allows the parallel composed processes can have many free different affine outputs (for example, $\overline{a} \,|\, \overline{b}$ is typable by $\{a : ()^\uparrow, b : ()^\uparrow\}$). (Weak) can weaken only $\updownarrow$ and ?-nodes since they allow the possibility of no action. In (Res), we do not allow $\uparrow$, ? or $\downarrow$-channel to be restricted since these actions expect their dual actions exist in the environment. ($\mathsf{In}^\downarrow$) ensures that x occurs precisely once (by Δ^{-u}) and that no free input is suppressed under the prefix (by $\uparrow ?\Delta$). $(\mathsf{In}^!)$ is the same as $(\mathsf{In}^{\downarrow})$ except that no free $\uparrow$-channels are suppressed (if a $\uparrow$-channel is under replication, then it can be used more than once). See also [6, 27] for more examples and explanation. Note that the common and functional rules remain as the same.

We list the key name substitution lemma. Note that Lemma 2.3 (3,4) no longer holds for the affine $\lambda\pi$. For example, if $\Delta_i = \{a:()^{\uparrow}\}$, then $\Delta_1 \not\asymp \Delta_2$. This feature makes it more difficult to prove the substitution lemma, in particular, for the annotated types in terms. Thus we need to articulate the lemmas w.r.t. the mode of channels. Below we prove (1,2,3) simultaneously.

LEMMA 5.1.

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

PROOF. (1-3) are proved simultaneously. (1) is proved just as Lemma 2.3 (3). For (2) and (3), we use the substitution lemma for well-formed channel and value types. See Appendix C.1. $\square$

By the above substitution lemma, one can prove the subject reduction theorem.

THEOREM 5.2. (type safety for $\lambda\pi^{\mathsf{A}}$) $\Gamma \vdash P : \tau$ *and* $P \twoheadrightarrow P'$ *imply* $\Gamma \vdash P' : \tau$.

PROOF. See Appendix C.2. $\square$

5.3 *Basic Properties of Affine* $\lambda\pi$

All properties listed in Propositions 2.10, 2.11 and 2.12, as well as basic properties in [6] hold in this calculus. We can show the following more refined properties ensured in $\lambda\pi^{\mathsf{A}}$. First we have:

PROPOSITION 5.3. (resource availability) *Suppose* $\Gamma \vdash P : \Delta$ *in* $\lambda\pi^{\mathsf{A}}$. *If* $a \in \mathsf{dom}(\Delta)$ *with* $\mathsf{md}(\Delta(a)) \in \{\downarrow, !\}$, *then* $P \Downarrow a^{\mathtt{I}}$.

Next we prove the preservation of unicity of input names and affine output names.[4] A definition of *unicity name error*, is given as follows, using a predicate $P \xrightarrow{Uerr}$. This should be read as saying: in the process P there is a runtime error, namely there is some name a which is ready to receive input at two distinct subprocesses. Similarly for the affine output. The definition is by a straightforward by using the predicate $P \Downarrow a^p$.

$$\frac{P \Downarrow a^p \quad Q \Downarrow a^p \quad p \in \{\downarrow, !, \uparrow, \updownarrow\}}{(P \mid Q) \xrightarrow{Uerr}} \qquad \frac{P \xrightarrow{Uerr}}{(P \mid Q) \xrightarrow{Uerr}} \qquad \frac{P \xrightarrow{Uerr}}{(Q \mid P) \xrightarrow{Uerr}} \qquad \frac{P \xrightarrow{Uerr}}{(\nu a)P \xrightarrow{Uerr}}$$

[4] The condition that every replicated input channel name is associated with a unique location is often called *locality* [64].

Then we have:

PROPOSITION 5.4. (unique resource accessibility) *Suppose* $\Gamma \vdash P : \Delta$ *in* $\lambda\pi^{\mathsf{A}}$. *Then for all* Q *such that* $P \twoheadrightarrow Q$, *we have* $Q \not\xrightarrow{U_{err}}$.

PROOF. By induction on derivation of $\Gamma \vdash P : \tau$, together with Theorem 5.2. $\square$

5.4 Affine Existential Type

The existential type of the affine type discipline is defined by regarding $(\exists[x\!:\!\sigma]\tau)^p$ as a special case of $(\Pi[x\!:\!\sigma]\tau)^p$. For example, $(\exists[x\!:\!\sigma]\tau)^{\downarrow}$ is well typed if $\mathsf{md}(\sigma) = ?$. We also need to add one following condition to the assumption of (t-exist) and (Pack).

$$\text{if } \mathsf{md}(\sigma) = \uparrow \text{ then } \mathsf{tail}(\tau) \vdash x\!:\!\sigma \tag{14}$$

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

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

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

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

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

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

By adding the condition in (14), we can prove Type-safety, as well as the basic properties listed in the previous basic properties with existential types.

5.5 Example

We revisit the examples in Section 1. We first show (6) and (7) are untypable in any context with the assumption $\overline{\mathtt{ack}}$ has affine output type $()^{\uparrow}$. Let $R_i \stackrel{\text{def}}{=} \overline{a}\langle \ulcorner P_i \,|\, \overline{\mathtt{ack}} \urcorner\rangle$ and $R_1 \,|\, R_2$ is typable under environment Γ and $\Gamma \vdash P_i : \Delta$ for some Δ such that $\mathsf{md}(\Delta) = ?$. Let $\tau = \ulcorner \Delta, \mathtt{ack}\!:\!()^{\uparrow}\urcorner$. Then we have:

$$\Gamma \vdash a\!:\!(\tau)^{?} \quad \text{ and } \quad \Gamma \vdash R_1 \,|\, R_2 \ : \ \{a\!:\!(\tau)^{?}\}$$

by $(\tau)^{?} \odot (\tau)^{?} = (\tau)^{?}$. Now we try to type $\mathsf{Server}(a) = !a(y\!:\!\tau).\,run\,y$. Then we have:

$$\Gamma, y\!:\!\tau \vdash run\,y \ : \ \mathtt{ack}\!:\!()^{\uparrow}, \Delta \tag{15}$$

However by the side condition $?\Delta$ in $(\mathsf{In}^{!})$, we cannot obtain $\mathsf{Server}(a)$. Similarly $run\,y \,|\, run\,y$ in (7) is untypable since we cannot apply (Par) because of $\{\mathtt{ack}\!:\!()^{\uparrow}\} \not\asymp \{\mathtt{ack}\!:\!()^{\uparrow}\}$.

Now we show the following process in (9) in Section 1 is typable by the affine typing system assuming $\Gamma \vdash Q_i : \Delta'$ for some Δ' such that $\Delta \asymp \Delta'$.

$$\mathsf{Server}(a) \,|\, (\nu\, \mathtt{ack})(\overline{a}\langle \ulcorner P_1 \,|\, \overline{\mathtt{ack}} \urcorner\rangle \,|\, \mathtt{ack}.Q_1) \,|\, (\nu\, \mathtt{ack})(\overline{a}\langle \ulcorner P_2 \,|\, \overline{\mathtt{ack}} \urcorner\rangle \,|\, \mathtt{ack}.Q_2)$$

Assume $\Gamma(a) = \langle \sigma'_a, \overline{\sigma_a}' \rangle$ where $\sigma'_a = (\exists[x:()^{\uparrow}]\ulcorner x:()^{\uparrow},\Delta\urcorner)^!$. Then we have

$$\Gamma, \mathtt{ack}:\langle ()^{\downarrow}, ()^{\uparrow} \rangle \vdash \overline{a}\langle \ulcorner P_1 \,|\, \overline{\mathtt{ack}} \urcorner \rangle : a:\overline{\sigma_a}', \mathtt{ack}:()^{\uparrow}$$

Noting $\mathtt{ack} \notin \mathsf{fn}(\sigma'_a, \Delta')$, by applying (Par) and (Res), we have:

$$\Gamma \vdash (\nu\,\mathtt{ack})(\overline{a}\langle \ulcorner P_1 \,|\, \overline{\mathtt{ack}} \urcorner \rangle \,|\, \mathtt{ack}.Q_1) : a:\overline{\sigma_a}', \Delta'$$

Now by assuming $run\,y \;=\; y()$ and $\tau = \ulcorner x:()^{\uparrow},\Delta\urcorner$, we have:

$$\Gamma \vdash a:\sigma'_a \quad \text{and} \quad \Gamma, \{x:()^{\uparrow}, y:\tau\} \vdash run\,y \;:\; x:()^{\uparrow}, \Delta$$

Hence an application of (In$\exists$) to the above gives us:

$$\Gamma \vdash \mathsf{Serve}(a) \;:\; a:\sigma'_a, \Delta$$

Thus the process in (9) has a non-trivial type "$a:\langle \sigma'_a, \overline{\sigma_a}' \rangle, \Delta \odot \Delta'$", as desired.

5.6 *Example: Remote Procedure Migration in $\lambda\pi^{A}$*

We revisit Example 4.1. The simple IO-system in the previous section cannot guarantee the uniqueness of servers. For example, we have the following server.

$$\mathtt{Server}_{bad1}(l) \stackrel{\text{def}}{=} (\nu c)(!l(y:\tau).(yc) \,|\, !c(x).\overline{x}\langle n \rangle \,|\, !c(x).\overline{x}\langle m \rangle) \tag{16}$$

By the affine typing system, $\mathtt{Server}_{bad}(l)$ is untypable since $!c(x).\overline{x}\langle n \rangle \,|\, !c(x).\overline{x}\langle m \rangle$ is untypable by $((\mathtt{nat})^{\uparrow})^! \not\asymp ((\mathtt{nat})^{\uparrow})^!$. In addition, the uniqueness of the answer at e is guaranteed by changing the type τ annotating y at $\mathtt{Server}(l)$ as follows:

$$\exists[e:(\mathtt{nat})^{\uparrow}]\Pi(x:\sigma_x)\{x:\sigma_x, e:(\mathtt{nat})^{\uparrow}\}$$

where $\sigma_x = ((\mathtt{nat})^{\downarrow})^?$. By the similar argument in the above, a server $!l(y:\tau).(yc\,|\,yc)$ is untypable, because yc has a process type $\Delta \stackrel{\text{def}}{=} \{e:(\mathtt{nat})^{\uparrow}, c:\sigma_x\}$ and $\Delta \not\asymp \Delta$.

6 Linear Type Discipline

6.1 *Termination of Mobile Code*

The second refinement is to integrate the linear type discipline into $\lambda\pi$. While there are varied forms of linearity constraints, here we consider the strongest form in which strong normalisation (SN) can be guaranteed via typing. SN is a basic property known in the typed λ-calculi, and offers a general basis for applications when combined with other type disciplines, including secrecy analysis [27, 63, 66, 67]. In the context of mobile computing, SN is also an interesting and sometimes required property for resource preservation, cf. [21, 46], see also examples at the end of this section, § 6.3. We show fine-grained process types can be used for realising this non-trivial regular behaviour in the presence of higher-order code mobility, leading to strongly normalising processes with the linear liveness property. The developed linear type discipline plays an essential role in the applications given in Part II.

We start from a simple example. Let us consider the following processes:

$$!b.\overline{a}\langle \ulcorner \overline{c} \urcorner \rangle \;|\; !c.\overline{b} \;|\; !a(x:\tau).\,run\,x \tag{17}$$

This process looks safe, but if it is composed with a message $\overline{b}$, a thunk $\ulcorner \overline{c} \urcorner$ is passed to a and executed, then $\overline{c}$ starts creating an infinite loop. Recent work [63] and [49]

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

Others are defined as in Figure 9.

FIGURE 10. Linear Higher-Order Typing System

have used the ordering of names to ensure termination of the π-calculus: however such analysis is not directly applicable in the untyped $\lambda\pi$, since the link between a to c which is dynamically created after higher-order code passing is invisible from the syntax.

To obtain information of causality of names a priori, we can use the fine-grained process types; here type τ can be assigned by $\ulcorner c:()^{?}\urcorner$ (instead of $\ulcorner\mathtt{proc}\urcorner$), hence $run\,x$ has type $\{c:()^{?}\}$, thus we can statically check $!a(x:\tau).run\,x$ has a causality from a to c. Hence we now know this process creates a vicious circle with the first process (which has a causality from b to a) and the second process (which has from c to b). Finally we can conclude (17) should be untypable. Note that if τ is only annotated by the constant type `proc`, the above parallel composed processes ends up `proc`, hence such a fine-gained analysis on causal behaviour is impossible. The reader can check even if c is restricted by ν, our fine-grained process types can represent that a vicious cycle created between the first and second processes by assigning a channel dependent type to b.

LINEAR TYPES AND TYPING The linear typing system for $\lambda\pi$ is obtained simply by integrating with the linear typing system in [63]. The syntax of types is as the same as $\lambda\pi^{\mathsf{A}}$. Further, we write $x:\tau \to x':\tau'$ when x' causally depends on x. $x:\tau \to x':\tau'$ is allowed if either (1) $\mathsf{md}(\tau) = \downarrow$ and $\mathsf{md}(\tau) = \uparrow$ or (2) $\mathsf{md}(\tau) = !$ and $\mathsf{md}(\tau) = ?$. For example, a simple agent $x.\overline{y}$ would be typed by $x:()^{\downarrow} \to y:()^{\uparrow}$. The algebras for modes **(a-m-1/2)** and for types **(a-t-1/2)** remain as the same. Then $\Delta_1 \asymp \Delta_2$ is defined by preventing circularity between channels. For example, $x_1:\tau_1 \to x_2:\overline{\tau_2} \not\asymp x_2:\tau_2 \to x_1:\overline{\tau_1}$. For the formal definition of $\asymp$, see Section 2 in [63]. From the affine typing system, we only have to modify input rules as listed in Figure 10. In these rules, $x:\tau \to \Delta$ denoting the result of adding a new edge from $x:\tau$ to each maximal node in Δ. Note when composing P_1 and P_2 by (Par), we check circularity of names (by $\Delta_1 \asymp \Delta_2$). Other rules remain the same.

6.2 Type Preserving Translation

For proving SN of the typed higher-order processes, a natural idea would be to use a type-preserving translation into the π^{L}-calculus. But a known translation from HOπ into π by [47, 48] cannot be directly used since it is *not* type preserving. In [47], 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,

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

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

$$\mathtt{unit}^\circ \stackrel{\text{def}}{=} (()^\uparrow)^! \qquad (\tau_1 \to \tau_2)^\circ \stackrel{\text{def}}{=} (\overline{\tau_1^\circ}\tau_2^\bullet)^! \qquad (\Pi(y\!:\!\sigma)\tau)^\circ \stackrel{\text{def}}{=} \begin{cases} (\sigma^\circ\tau^\bullet)^! & (\mathsf{md}(\sigma) = ?) \\ (\tau^\bullet)^! & (\mathsf{md}(\sigma) = \uparrow) \end{cases}$$

$$((\Pi[x_1\!:\!\sigma_1,\ldots,x_n\!:\!\sigma_n]\tau_1,\ldots,\tau_m)^{p_\mathrm{I}})^\circ \stackrel{\text{def}}{=} (\sigma_1^\circ,\ldots,\sigma_n^\circ,\overline{\tau_1^\circ},\ldots,\overline{\tau_m^\circ})^{p_\mathrm{I}}$$

$$((\Pi[x_1\!:\!\sigma_1,\ldots,x_n\!:\!\sigma_n]\tau_1,\ldots,\tau_m)^{p_\mathrm{O}})^\circ \stackrel{\text{def}}{=} (\sigma_1^\circ,\ldots,\sigma_n^\circ,\tau_1^\circ,\ldots,\tau_m^\circ)^{p_\mathrm{O}} \qquad \langle\sigma_1,\sigma_2\rangle^\circ \stackrel{\text{def}}{=} \langle\sigma_1^\circ,\sigma_2^\circ\rangle$$

$$((\exists[x\!:\!\sigma]\tau)^{p_\mathrm{I}})^\circ \stackrel{\text{def}}{=} (\sigma^\circ,\overline{\tau^\circ})^{p_\mathrm{I}} \qquad ((\exists[x\!:\!\sigma]\tau)^{p_\mathrm{O}})^\circ \stackrel{\text{def}}{=} (\sigma^\circ,\tau^\circ)^{p_\mathrm{O}}$$

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

FIGURE 11. Mapping of Linear Types

with $\Gamma = \Delta, a\!:\!\langle\sigma_a,\overline{\sigma}_a\rangle$ and $\sigma_a = (\ulcorner\Delta\urcorner)^{p_\mathrm{O}}$. However using a mapping in [47], 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 $\sigma_a^\circ = (())$ and Δ° is a translation of P's process type. With this encoding, basic notions as linearity and affinity are not preserved either: encodings of the typable terms of $\lambda\pi^{\mathsf{A}}$ and $\lambda\pi^{\mathsf{L}}$ such as $\overline{a}\langle\ulcorner\overline{b}\urcorner\rangle \mid a(x).(\lambda z.\overline{b})x$ become untypable in π^{A} and π^{L}.

Fortunately, the fine-grained process types in $\lambda\pi$ 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, linear names in process type in P above originally occur bound, and are eventually instantiated when x finally migrates from the receiver. Note x *knows* linear free names of P because of its assigned type $\ulcorner\Delta\urcorner$. Thus assuming $P = \overline{b}$ with b linear name, the new translation becomes:

$$[\![\overline{a}\langle\ulcorner P\urcorner\rangle]\!] \stackrel{\text{def}}{=} \overline{a}(c)!c(b).\overline{b} :\ a\!:\!\overline{\sigma}_a \ \text{ and } \ [\![!a(x).\mathit{run}\,x]\!] \stackrel{\text{def}}{=} !a(x).\overline{x}\langle b\rangle :\ a\!:\!\sigma_a, b\!:\!()^\uparrow$$

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 [27, Figure 7], we can now translate the linear higher-order process into a the π^{L}-process in [27]. The translation preserves not only typability but action modes and termination. Thus the contextual congruence for $\lambda\pi^{\mathsf{L}}$ (see below) is soundly embedded in the π^{L}-calculus [63].

The formal encoding of types and terms are listed in Figures 11 and 12, respectively. The key idea here is, in order to preserve the linearity, a process type Δ is translated into a unique $\uparrow$-type as found in the mapping $\Delta^\bullet$ in Figure 11.

For the proof of operational correspondence, we use the extended reduction $\mapsto$ from [63] which contracts the redex under prefixes.

DEFINITION 6.1. (extended reductions) We define $\mapsto$ as the compatible relations on

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

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

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

$$[\![u(y{:}\exists[x{:}\sigma]\tau).P]\!]_\Gamma \stackrel{\text{def}}{=} u(x{:}\sigma^\circ,y{:}\overline{\tau^\circ}).[\![P]\!]_{\Gamma,x:\sigma,y:\tau}$$

$$[\![!\,u(y{:}\exists[x{:}\sigma]\tau).P]\!]_\Gamma \stackrel{\text{def}}{=} !\,u(x{:}\sigma^\circ,y{:}\overline{\tau^\circ}).[\![P]\!]_{\Gamma,x:\sigma,y:\tau}$$

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

$$[\![\overline{u}\langle V\rangle]\!]_\Gamma \stackrel{\text{def}}{=} (\nu c)(\overline{u}\langle v,c\rangle\langle c = V\rangle_\Gamma) \quad \text{if } \Gamma \vdash u{:}(\exists[x{:}\sigma]\tau)^{p\mathrm{o}} \quad \text{and } \Gamma \vdash \langle v, V\rangle : \exists[x{:}\sigma]\tau$$

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

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

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

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

$$[\![V]\!]_\Gamma^m \stackrel{\text{def}}{=} \overline{m}(c)\langle c = V\rangle_\Gamma \quad \langle c = u\rangle_\Gamma \stackrel{\text{def}}{=} [c \to u]^{\tau^\circ} \quad (\Gamma \vdash u{:}\tau) \quad \langle c = ()\rangle_\Gamma \stackrel{\text{def}}{=} !c(x).\overline{x}$$

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

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

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

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

Copy Cat

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

FIGURE 12. Encoding of $\lambda\pi^{\mathsf{L}}$-Calculus into π^{L}-Calculus

processes modulo $\equiv$ respectively generated by the following rules.

$$C[\overline{x}(\vec{y})P] | x(\vec{y}).Q \mapsto C[(\nu\vec{y})(P|Q)]$$
$$C[\overline{x}(\vec{y})P] | !x(\vec{y}).Q \mapsto C[(\nu\vec{y})(P|Q)] \mid !x(\vec{y}).Q$$
$$(\nu x)!x(\vec{y}).Q \mapsto \mathbf{0}$$

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

Define $\cong$ on typed $\lambda\pi^{\mathsf{A}}$ and $\lambda\pi^{\mathsf{L}}$ processes as the maximum typed congruence which satisfies:

1. it is reduction-closed, i.e. $P \cong_s Q$ and $P \longrightarrow P'$ implies $Q \twoheadrightarrow Q'$ and $P' \cong_s Q'$; and
2. if $\vdash P_1 \cong P_2 : x{:}()^\uparrow$ and $P_1 \Downarrow_{x^\uparrow}$ then $P_2 \Downarrow_{x^\uparrow}$ [25].

The following is mechanically proved by induction on P. Note also these properties are proved not only between $\lambda\pi^{\mathsf{L}}$ and π^{L} but also between $\lambda\pi^{\mathsf{A}}$ and π^{A}.

PROPOSITION 6.2.

1. (type preserving property for functions) *Suppose* $\Gamma \vdash P : \tau$ *with* $\mathsf{fv}(P) = \emptyset$ *and* $\Gamma \vdash \tau : \mathtt{tp}_H$. *Then we have* $\Gamma^\circ, u:\tau^\bullet \vdash [\![P]\!]_u : u:\tau^\bullet, \vec{u}:\vec{\sigma}^\circ$ *with* $\Gamma \vdash u_i : \sigma_i^?$.
2. (type preserving for processes) *Suppose* $\Gamma \vdash P : \Delta$ *and* $\mathsf{fv}(P) = \emptyset$. *Then* $\Gamma^\circ \vdash [\![P]\!]_u : \Delta^\circ, \vec{u}:\vec{\sigma}^\circ$ *with* $\Gamma \vdash u_i : \sigma_i^?$.
3. (operational correspondence) *Suppose* $\Gamma \vdash P : \tau$ *and* $P \longrightarrow P'$. *Then* $[\![P]\!]_u \mapsto^+ [\![P']\!]_u$ *or* $[\![P]\!] \mapsto^+ [\![P']\!]$.
4. (computational adequacy 1) *Suppose* $\Gamma \vdash P : \Delta$ *with* $x \notin \mathsf{fv}(P)$. *Then* $P \Downarrow_{a^P}$ *iff* $[\![P]\!] \Downarrow_{a^P}$.
5. (computational adequacy 2) *Suppose* $\Gamma \vdash P : \mathtt{unit}$ *with* $x \notin \mathsf{fv}(P)$. *Then* $[\![P]\!]_u \Downarrow_{u^\uparrow}$.
6. (soundness) $[\![P]\!] \cong [\![Q]\!]$ *implies* $P \cong Q$.

Below Δ is closed means $\{!_{\text{L}}, !_{\text{A}}, \updownarrow\} \supseteq \mathsf{md}(\Delta)$. Note that (2) closes the open problem posed in [49].

THEOREM 6.3.

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

PROOF. (1) is proved just as $\lambda\pi^{\text{A}}$. (2) is proved by Proposition 6.2 (3) and Theorem 3.2 in [63]. (3) is proved by Proposition 6.2 (4) and Proposition 7.1 in [63]. □

Intuitively, (3) says that, when composed with all complementary processes, a linear channel *eventually* fires. The property is strengthen and preserved in the presence of non-terminated and non-deterministic computation, as shown in the following sections.

6.3 Examples

We revisit Example 5.6. Suppose the following agent which repeatedly emits infinite messages at c.

$$\omega(c) \stackrel{\text{def}}{=} (\nu\, uu')(!u.(\overline{c}\langle x\rangle \,|\, \overline{u'}) \,|\, !u.\overline{u}' \,|\, \overline{u}) \tag{18}$$

Then it exhibits divergence as:

$$\omega(c) \longrightarrow \overline{c}\langle x\rangle \,|\, \omega(c) \longrightarrow^+ \overline{c}\langle x\rangle \,|\, \overline{c}\langle x\rangle \,|\, \omega(c) \longrightarrow^+ \cdots$$

Now consider the following malicious client request, which consumes the server's local resources unboundedly.

$$\mathtt{let}\ x = \overline{l}\langle \lambda(c).(\omega(c) \,|\, \overline{e}\langle 1\rangle)\rangle\ \mathtt{in}\ R$$

By the previous affine typing systems, this code is typable by the same type as $\mathtt{let}\ x = \overline{l}\langle \mathtt{twice}\rangle\ \mathtt{in}\ R$. If $\mathtt{Server}(l)$ interacts with this program, then the service at c might not be available forever to other clients since channel c is infinitely occupied by $\omega(c)$. By the linear typing system in this section, we can check $\omega(c)$ in (18) is untypable since its body has circular type such that $u : ()^! \to u' : ()^?$ and $u' : ()^! \to u : ()^?$.

(Well-Formed Types)

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

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

We replace by $\uparrow$ by $\mathcal{M}_\uparrow$ and ? by $\mathcal{M}_?$ in (t-$\downarrow$). Others are from Figure 9.

(Typing Rules)

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

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

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

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

Others are from Figure 9 by replacing ! by $\mathcal{M}_!$ in (Res) and ? by $\mathcal{M}_?$ in (Weak-?).

FIGURE 13. Linear-Affine Higher-Order Typing System

As another example, consider the following server which creates an infinite loop after initial application with c:

$$\mathtt{Server}_\Omega(l) \stackrel{\text{def}}{=} (\nu\, c)(!l(y:\tau).(y\,c) \,|\, \Omega(c)) \tag{19}$$

with $\Omega(c) \stackrel{\text{def}}{=} (\nu\, c')(!c(x).\overline{c'}\langle x\rangle \,|\, !c'(x).\overline{c}\langle x\rangle)$. Then the client is unable to receive the answer at e forever. This server is untypable in $\lambda\pi^{\mathrm{L}}$ since $\Omega(c)$ is so. Hence in $\lambda\pi^{\mathrm{L}}$, even if we compose any R', we can guarantee:

$$\mathtt{Serve}(l) \,|\, \mathtt{let}\ x = \bar{l}\langle\mathtt{twice}\rangle\ \mathtt{in}\ R \,|\, R' \twoheadrightarrow (\mathtt{Serve}(l) \,|\, R\{2n/x\} \,|\, R'')$$

7 Further Integration

In this section we integrate linear and affine type disciplines and further extend the calculus with subtyping, references and location.

7.1 Linear/Affine Type Discipline

We start with coherently mixing the two typing systems, linear and affine ones, then prove the two kinds of liveness in the higher-order code passing under the strong fairness condition; one is a response property of the remote procedure call, another is an encapsulation property of higher-order code by hidden names. To clarify this idea, first we define a simple notion of response liveness property.

DEFINITION 7.1. (linear liveness) We say P *eventually answers at* a if and only if whenever $P \twoheadrightarrow Q$ we have $Q \Downarrow_a$. □

The above linear liveness means that we can eventually get a unique reply at a linear channel. Note that even if P eventually answers at x, it may not guarantee termination of P (for example, $P \equiv \overline{x} \mid \Omega$ diverges, while $P \Downarrow_x$). This simple liveness notion is further refined by introducing a fairness condition.

Mixing Types and Typing

From now on we distinguish linear and affine channels by annotating modes with A for affine and L for linear ($\uparrow$ as $\uparrow_{\mathrm{A}}$ for affine output, $\uparrow_{\mathrm{L}}$ for linear output, etc.). The composition rules for modes are changed with respect to modes as:

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

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

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

Then we construct the typing system where a linear output is only carried by a linear replication; and a non-linear input never suppresses linear outputs (see [27, 62] for the details). We define the well-formedness and typing rules in Figures 13. We shall call this calculus $\lambda\pi^{\mathsf{LA}}$.

7.2 Basic Properties of Linear/Affine $\lambda\pi$

The subject reduction theorem is proved just as in λA. Then the same liveness property is proved via translation in Figure 12. Here we only have to modify the mapping of types as:

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

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

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

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

PROPOSITION 7.2. (linear liveness) *Suppose* $\Gamma \vdash P : \Delta, a : ()^{\uparrow_{\mathrm{L}}}$ *with* $\mathsf{fv}(P) = \emptyset$ *and* Δ *closed. Then whenever* $P \twoheadrightarrow P'$, P' *eventually answers at* a.

PROOF. The proof is obtained by translating the $\lambda\pi^{\mathsf{LA}}$ into the π^{LA} via the translation in Figure 12; then the theorem is immediate by the result in [62] where the linear liveness in π^{LA} is proved by reducing the transition relations of π^{LA} into π^{L}, see Figure 16 in § 9. □

Another interesting liveness property comes from existential channel types. Let $l, l', \ldots$ be given by:

$$l \ ::= \ \tau \mid u\langle \vec{V} \rangle \mid (\nu \vec{a})\overline{u}\langle \vec{V} \rangle$$

Visible actions form either $u\langle \vec{V} \rangle$ (input) or $(\nu \vec{a})\overline{u}\langle \vec{V} \rangle$ (output). Using these labels, the typed transition $\Gamma \vdash P : \Delta \stackrel{l}{\longrightarrow} \Gamma' \vdash P' : \Delta'$ is defined as in Figure 17 in Appendix D by mixing the standard higher-order and the linear transition systems [47, 63, 66]. We also write $\Gamma_1 \vdash P_1 : \Delta_1 \xrightarrow{l}$ if there exists $\Gamma_2 \vdash P_2 : \Delta_2$ such that $\Gamma_1 \vdash P_1 : \Delta_1 \xrightarrow{l} \Gamma_2 \vdash P_2 : \Delta_2$. We also write $P_1^{\Delta_1} \xrightarrow{l} P_2^{\Delta_2}$ or $P_1 \xrightarrow{l} P_2$ for $\Gamma_1 \vdash P_1 : \Delta_1 \xrightarrow{l} \Gamma_2 \vdash P_2 : \Delta_2$, if environments and types are obvious from the context.

THEOREM 7.3. (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* $P \xrightarrow{u\langle \ulcorner \overline{c}\langle n \rangle \urcorner \rangle} P'$ *implies there exists* Q *such that* $P' \twoheadrightarrow Q \xrightarrow{\overline{c}\langle n \rangle}$.

PROOF. See Appendix D. □

This theorem means that an acknowledgement bound by existential quantification is eventually returned to the sender without being touched by the receiver (note we can easily generalise n to an arbitrary higher order value V such that $\mathsf{fv}(V) = \emptyset$).

7.3 Response Property

Now we examine the above properties under the assumption of fair schedule [11, 37]. Let S be a possibly infinite set of processes and O be a possibly infinite set of occurrences of processes. Here we abuse the labelled transition relation and occurrences of terms. Then we write

$$\phi \;:\; P_1^{\Delta_1} \xrightarrow{l_1} P_2^{\Delta_2} \xrightarrow{l_2} P_3^{\Delta_3} \xrightarrow{l_3} P_4^{\Delta_4} \cdots$$

for a trace which stars from $P_1^{\Delta_1}$. We say l is *enable* in $P_1^{\Delta_1}$ if there exists $P_2^{\Delta_2}$ such that $P_1^{\Delta_1} \xrightarrow{l} P_2^{\Delta_2}$. Then we define:

DEFINITION 7.4. (fair execution) Let Φ_l be a predicate such that $\Phi_l \stackrel{\text{def}}{=} (\neg l$ enable in $P^{\Delta} \vee P^{\Delta} \xrightarrow{l} Q^{\Delta'})$. We say an infinite trace $\phi \;:\; P_1^{\Delta_1} \xrightarrow{l_1} P_2^{\Delta_2} \xrightarrow{l_2} P_3^{\Delta_3} \xrightarrow{l_3} P_4^{\Delta_4} \cdots$ is Φ-*fair* if for all $l \in O$, $\Phi_l(P_i) = True$ for infinitely many i. □

Then the following two liveness properties are corollaries from Proposition 7.2 and Theorem 7.3, respectively, together with [62, Lemma 2 (4) (the extended liveness)].

COROLLARY 7.5. (response) *Suppose* $\Gamma \vdash P : \Delta$ *and* $\mathsf{fv}(P) = \emptyset$ *with* Δ *closed and* $\Gamma \vdash u : (\Pi[c : (\vec{\tau})^{\uparrow_{\mathrm{L}}}]\vec{\tau}')^{!_{\mathrm{L}}}$. *Assume* $\phi \;:\; P_1^{\Delta} \xrightarrow{l_1} P_2^{\Delta_2} \xrightarrow{l_2} P_3^{\Delta_3} \xrightarrow{l_3} P_4^{\Delta_4} \cdots$ *is* Φ*-fair with* $\mathsf{fv}(l_i) = \emptyset$ *and* $P_j^{\Delta_j} \xrightarrow{u\langle c, \vec{V} \rangle}$ *for some* $\vec{V}$ *in* ϕ. *Then there exists* $k \geq 1$ *such that* $P_{j+k}^{\Delta_{j+k}} \xrightarrow{l} P_{j+k+1}^{\Delta_{j+k+1}}$ *with* $l = \overline{c}\langle \vec{W} \rangle$ *for some* $\vec{W}$ *in* ϕ.

COROLLARY 7.6. (hiding by existential name) *Suppose* $\Gamma \vdash P : \Delta$ *and* $\mathsf{fv}(P) = \emptyset$ *with* Δ *closed and* $\Gamma \vdash u : (\exists[c : (\mathtt{nat})^{\uparrow_{\mathrm{L}}}] \ulcorner c : (\mathtt{nat})^{\uparrow_{\mathrm{L}}} \urcorner)^{!_{\mathrm{L}}}$. *Suppose* $\phi \;:\; P^{\Delta} \xrightarrow{l_1} P_2^{\Delta_2} \xrightarrow{l_2} P_3^{\Delta_3} \xrightarrow{l_3} P_4^{\Delta_4} \cdots$ *is* Φ*-fair with* $\mathsf{fv}(l_i) = \emptyset$ *and* $P_j^{\Delta_j} \xrightarrow{u\langle \ulcorner \overline{c}\langle n \rangle \urcorner \rangle}$ *for some* j. *Then there exists* $k \geq 1$ *such that* $P_{j+k}^{\Delta_{j+k}} \xrightarrow{\overline{c}\langle n \rangle} P_{j+k+1}^{\Delta_{j+k+1}}$ *in* ϕ.

By this typing system, in Example 4.1, 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 procedure call to the server can always succeed as follows.

$$\mathtt{Serve}(l) \,|\, \mathtt{let}\; x = \bar{l}\langle \mathtt{twice}\rangle \;\mathtt{in}\; R \,|\, R' \longrightarrow\!\!\!\!\rightarrow (\mathtt{Serve}(l) \,|\, R\{2n/x\} \,|\, R'')$$

7.4 Subtyping and Nondeterminism

We augment the $\lambda\pi^{\mathrm{LA}}$ 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 [6, 27, 63, 66]. The integration is straightforward, extending the grammar as follows.

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

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:

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

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*, $[\![n]\!]_u \stackrel{\mathrm{def}}{=} !u(c)\overline{c}\mathtt{in}_n$, acts as a server which necessarily returns a fixed answer, n. See [6, 27, 66] for further examples. The incorporation of references results in concurrency and nondeterminism.

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. We add $!_{\mathrm{LM}}, !_{\mathrm{M}}$ to $\mathcal{M}_!$ and $?_{\mathrm{LM}}, ?_{\mathrm{M}}$ to $\mathcal{M}_?$.

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

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

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 say σ is mutable if $\mathsf{md}(\sigma) = ?_{\mathrm{M}}$, $\sigma = \mathsf{rw}\langle\sigma'\rangle$ or $\sigma = \mathsf{w}\langle\sigma'\rangle$.

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. The subtyping of process types means a process can gain their access points until it reaches the limit specified by the global environment. Hence we have:

$$\Gamma \vdash \{u_1{:}\sigma_1, \ldots, u_n{:}\sigma_n\} \leq \{u_1{:}\sigma_1', \ldots, u_n{:}\sigma_n', \ldots, u_n{:}\sigma_{n+k}\}$$

with $\Gamma \vdash \sigma_i \leq \sigma'_i$. We also assume σ'_{n+i} has either ?-mode or $\updownarrow$-mode since the other names cannot be weakened to preserve the unicity of names (cf.(Weak) in Figure 9).

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 σ and output $\overline{\sigma}$). The composition of channel types is similarly defined; we need to add only the following composition rule for with ?-mode;

$$[\tau_1 \ldots \oplus \tau_n]^? \odot [\tau'_1 \ldots \oplus \tau'_n \ldots \oplus \tau_{n+k}]^p \stackrel{\text{def}}{=} [\tau_1 \sqcup \tau'_1 \ldots \oplus \tau_n \sqcup \tau'_n \ldots \oplus \tau_{n+k}]^p$$

where $\sqcup$ is a partial join operator defined in Figure 19 in Appendix D.1. Formally we define:

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

(s-t-2) $\sigma_1 \odot \langle \sigma_2, \sigma \rangle = \sigma_1 \odot \langle \sigma, \sigma_2 \rangle = \langle \sigma_1 \sqcup \sigma_2, \sigma \rangle = \langle \sigma, \sigma_1 \sqcup \sigma_2 \rangle$ with $\mathsf{md}(\sigma_{1,2}) \in \mathcal{M}_?$.

In the above, the unary case is defined setting $I = J = \{1\}$. Note that we only have to check the composability of the outmost node. The definition of $\Delta_1 \odot \Delta_2$ remains as the same. We list the subtyping and the typing rules in Figure 18. This is a straightforward integration with the system in [27, § 4]. See [27, § 4] for the detailed explanation. Due to the branching subtyping for the reference agent, the rules in [27] are simplified. As in [65, Proposition 3.1], we can easily check under an arbitrary well-formed environment, the subtyping relation over types is a partial order and finite bounded complete ([65, Definition 3.1]). We have:

PROPOSITION 7.7.

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

The basic properties such as subject reduction and availability of resources are proved in this system. The main theorem in this system follows.

PROPOSITION 7.8. (the minimum interface) *Given* Γ *and* P, *suppose* $\Gamma \vdash P : \tau$. *Then there exists the unique* τ_{min} *such that* $\Gamma \vdash \tau_{min} < \tau$ *and* $\Gamma \vdash P : \tau_{min}$. *In particular, if* $\Gamma \vdash \tau : \mathtt{tp}_p$, *then* $\tau_{min} = \Delta$.

PROOF. The proof is obtained by constructing the minimum typing system just as done for Proposition 2.10 (we also add the syntactic direct rules for the arrow and functional dependent types as in [64, Figure 11 in Appendix C]). $\square$

Note that this property could *not* be proved in [65] due to the presence of `proc` and the name hiding rule (see footnote 1 in § 2.4). We can also prove the same maximum property as in Proposition 2.10 and the decidability theorem as in Proposition 2.11 in $\lambda\pi^{\mathsf{LAM}}_{<}$.

The liveness properties on linear channels, are again obtained via the correctness of the translation. The reference types are mapped homomorphically, e.g. $(\mathsf{ref}\langle\sigma\rangle)^\circ =$

$\mathsf{ref}\langle\sigma^\circ\rangle$, $(\mathsf{r}\langle\sigma\rangle)^\circ = \mathsf{r}\langle\sigma^\circ\rangle$ and $(\mathsf{w}\langle\sigma\rangle)^\circ = \mathsf{w}\langle\sigma^\circ\rangle$. A reference agent is translated into itself, $[\![\mathsf{ref}\langle xv\rangle]\!]_\Gamma = \mathsf{ref}\langle xv\rangle$. The branching/selection are mapped as the same as the unary case. The correctness of the translation is proved together with the following extended proposition, which is easily proved by induction on τ and $\tau < \tau'$.

PROPOSITION 7.9. *Suppose* $\Gamma \vdash \tau_1 \leq \tau_2$. *Then* $\Gamma^\circ \vdash \tau_1^\bullet \leq \tau_2^\bullet$ *and* $\Gamma^\circ \vdash \tau_1^\circ \leq \tau_2^\circ$.

We conclude with a small example of a branching subtyping.

7.5 Examples: Subtyping and Code Mobility

First we will show that the atomic subtyping such as $\mathtt{bool} < \mathtt{nat}$ can be simulated using the encoding into the $\lambda\pi_{\leq}^{\mathsf{LAM}}$. First boolean agents are encoded as follows:

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

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

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

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

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

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

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

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

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

Then one can check $P\,|\,\overline{a}\langle\ulcorner\mathbf{0}\urcorner\rangle$, $P\,|\,\overline{a}\langle\ulcorner[\![\mathtt{true}]\!]_u\urcorner\rangle$ and $P\,|\,\overline{a}\langle\ulcorner[\![\mathtt{false}]\!]_u\urcorner\rangle$ are typable while $P\,|\,\overline{a}\langle\ulcorner[\![n]\!]_u\urcorner\rangle$, which causes the run-time-error, is not typable, since

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

7.6 Location

We show how we can consistently extend $\lambda\pi$ with a simple notion of location which is similar with those in [64]. The incorporation of location is important in many applications: one example of its usage will be illustrated in the next section. Below $l, k, \ldots$ range over location names.

$$\text{Networks:}\quad \mathsf{N}\ ::=\ \mathbf{0}\ \mid\ l[P]\ \mid\ \mathsf{N}_1\,|\,\mathsf{N}_2\ \mid\ (\nu a\!:\!\sigma)\mathsf{N}$$

We use $\mathsf{fl}(\mathsf{N})$ to denote the set of free location names in N. For simplicity, we do not include other primitives such as location creation and a spawn operator, which are easily integrated with this calculus, see [20]. The reduction rules and the typing rules are defined in Figure 14. Note that Figure 14 can be added to the top of any of $\lambda\pi$-typing systems, including a simple IO-typing system. Immediately we have:

THEOREM 7.10. (subject reduction) $\Gamma \vdash \mathsf{N} : \Delta$ *and* $\mathsf{N} \longrightarrow \mathsf{N}'$ *imply* $\Gamma \vdash \mathsf{N}' : \Delta$.

(Reduction)

$$l[a(\vec{x}\!:\!\vec{\tau}).P\,|\,Q]\,|\,k[\overline{a}\langle\vec{V}\rangle\,|\,R] \longrightarrow l[P\{\vec{V}/\vec{x}\}\,|\,Q]\,|\,k[R]$$
$$l[!\,a(\vec{x}\!:\!\vec{\tau}).P\,|\,Q]\,|\,k[\overline{a}\langle\vec{V}\rangle\,|\,R] \longrightarrow l[P\{\vec{V}/\vec{x}\}\,|\,!\,a(\vec{x}\!:\!\vec{\tau}).P\,|\,Q]\,|\,k[R]$$

$$\frac{P \longrightarrow P'}{l[P] \longrightarrow l[P']} \quad \frac{\mathsf{N} \longrightarrow \mathsf{N}'}{\mathsf{N}\,|\,\mathsf{N}_0 \longrightarrow \mathsf{N}'\,|\,\mathsf{N}_0} \quad \frac{\mathsf{N} \longrightarrow \mathsf{N}'}{(\nu u)\mathsf{N} \longrightarrow (\nu u)\mathsf{N}'} \quad \frac{\mathsf{N} \equiv \mathsf{N}_0 \longrightarrow \mathsf{N}_0' \equiv \mathsf{N}'}{\mathsf{N} \longrightarrow \mathsf{N}'}$$

(Structure Rules) $l[(\nu a\!:\!\sigma)P] \equiv (\nu a\!:\!\sigma)l[P]$.

Others are from 2 replacing P by N.

(Typing Rules)

$$(\mathsf{Net})\ \frac{\Gamma \vdash P : \Delta}{\Gamma \vdash l[P] : \Delta} \qquad (\mathsf{Par})\ \frac{\Gamma \vdash \mathsf{N}_i : \Delta_i \quad \Delta_1 \asymp \Delta_2 \quad \mathsf{fl}(\mathsf{N}_1) \cap \mathsf{fl}(\mathsf{N}_2) = \emptyset}{\Gamma \vdash \mathsf{N}_1\,|\,\mathsf{N}_2 : \Delta_1 \odot \Delta_2}$$

(Res),(Weak) for networks are obtained replacing P with N in an appropriate typing system.

FIGURE 14. Location Typing System

8 Representing Practical Code Mobility in $\lambda\pi$

In the following, we shall illustrate the expressiveness of the typed formalism we have developed so far by two larger examples, in which we represent practical situations of code mobility as found in e.g. Java and CLR. Our aim is to show how the channel dependent/existential types extended with linear/affine type structure play an essential role for assigning significant types to programs with code mobility, and how these types lead to non-trivial behavioural constraints.

8.1 Example: A Shared Resource with Remote Procedure Migration

First we start with a revised version of Example 4.1. Assume the following reference

$$\mathsf{ref}\langle t2\rangle$$

This agent can be regarded as a small low level shared natural number memory. A remote server which accepts client's request to access this memory is modelled as:

$$k_1[\mathsf{Serv}_1(l,2)] \quad \stackrel{\text{def}}{=} \quad k_1[(\nu t)(\mathsf{ref}\langle t2\rangle|\ !\,l[(x\!:\!\tau_1).xt\ \&\ (x\!:\!\tau_2).xt])] \tag{20}$$

A channel t is hidden since the address of the memory should be invisible from the clients. Note that each branching contains the same body "xt" except the type annotation, τ_i. We will demonstrate how the combination of the subtyping and the type annotation τ_i provides the fine-grained resource control.

Assume the following two kinds of code, the first is similar with (13) in Example 4.1, while the second writes the data before reads it twice in order to use the remote memory. Below we write `read` for $\mathtt{in}_1$ and `write` for $\mathtt{in}_2$ to the reference. We also denote $\overline{a}\langle n,(c)\rangle P$ for $(\nu c)(\overline{a}\langle n,c\rangle \mid P)$.

Code 1: $\texttt{return}\,\texttt{twice}\,\texttt{to}\,e \stackrel{\text{def}}{=} \lambda(t).\bar{t}\texttt{read}(c)c(x).\bar{t}\texttt{read}(c)c(x).\overline{e}\langle x+x\rangle$
with type $\Pi(t\!:\!\sigma_r)\{t\!:\!\sigma_r, e\!:\!(\texttt{nat})^{\uparrow_{\mathrm{L}}}\}$ where $\sigma_r = \mathsf{r}\langle\texttt{nat}\rangle$

Code 2: $\texttt{return}\,\texttt{write}\,3\,\texttt{then}\,\texttt{twice}\,\texttt{to}\,e$
$\stackrel{\text{def}}{=} \lambda(t).\bar{t}\texttt{write}\langle 3(c)\rangle c.\bar{t}\texttt{read}(c)c(x).\bar{t}\texttt{read}(c)c(x).\overline{e}\langle x+x\rangle$
with $\Pi(t\!:\!\sigma_{rw})\{t\!:\!\sigma_{rw}, e\!:\!(\texttt{nat})^{\uparrow_{\mathrm{L}}}\}$ where $\sigma_{rw} = \mathsf{rw}\langle\texttt{nat}\rangle$

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

$$\begin{aligned}&\texttt{let}\;x = \bar{l}\texttt{in}_i\langle\texttt{write}\;3\;\texttt{then}\;\texttt{twice}\rangle\;\texttt{in}\;R\\ &\stackrel{\text{def}}{=}\;(\nu e)(\bar{l}\texttt{in}_i\langle\texttt{return}\,\texttt{write}\,3\,\texttt{then}\,\texttt{twice}\,\texttt{to}\,e\rangle \mid e(x).R)\end{aligned}$$

Assume an agent at location k_2 tries to perform this code as follows:

$$k_2[\texttt{let}\;x = \bar{l}\texttt{in}_2\langle\texttt{write}\;3\;\texttt{then}\;\texttt{twice}\rangle\;\texttt{in}\;R] \tag{21}$$

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

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

Then the composition between $\mathsf{Serv}_1(l,2)$ and a client in (21) is untypable since, for any environment, we have $\sigma_{rw} \not\leq \sigma_r$. Similarly we annotate the first branching by:

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

Then the following callers are typable

$$\begin{aligned}&k_2[\texttt{let}\;x = \bar{l}\texttt{in}_1\langle\texttt{write 3 then twice}\rangle\;\texttt{in}\;R]\\ &k_3[\texttt{let}\;x = \bar{l}\texttt{in}_2\langle\texttt{twice}\rangle\;\texttt{in}\;R]\end{aligned}$$

8.2 Example: Basic Primitives for Code Mobility in a Class-Based Distributed Language

A SIMPLE DISTRIBUTED PROGRAM In this subsection, we show $\lambda\pi_{\leq}^{\mathrm{LAM}}$ can embed a simple multi-threaded class-based language with dynamic code and class distribution (the full encoding as well as the language definition can be found in [67, 69]). Figure 15 shows an example of dynamic code passing, using a Java-like syntax [8, 31]. In this code, there are two symmetric primitives for packing and unpacking code.

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

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

In Figure 15, a piece of code is sent from Location (1) to Location (2), with Location (1)'s local class called `Mcal`. 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):

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

	Location (1)	**Location (2)**

```
Class Mcal
{
double z;
double twice(int x){
  z := x^x;
  return z;
}
double thrice(int x){
  z := x^x^x;
  return z;
}
}
```

Location (1)

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

Code A:

```
let o = new Mcal(0)
in myobj.ack();
   return o.twice(777);
```

Location (2)

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

FIGURE 15. Code Passing with Class

The first line serialises 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 $\ulcorner \mathrm{A}\ \mathtt{with}\ \mathtt{CT} \urcorner$ with $\mathtt{CT} = \mathtt{Mcal} \mapsto \mathtt{Class\ Mcal}\{...\}$. If the calculation finishes, the result is sent back to Location (1) and stored in a reference w. `Mcal` 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.

ENCODING We first show the encoding of the class `Mcal`, using a branching and references. The method `twice` is encoded as

$$[\![\mathtt{twice}(\mathtt{int}\ x)...]\!]_e \stackrel{\text{def}}{=} \overline{z}\mathtt{write}\langle x\hat{\ }x,(c)\rangle c.\overline{z}\mathtt{read}(c')c'(y).\overline{e}\langle y\rangle \tag{22}$$

which is essentially similar with Code 1 in the previous subsection. The method `thrice` is defined just by replacing $x\hat{\ }x$ by $x\hat{\ }x\hat{\ }x$.

Then $\mathtt{Class\ Mcal}\{...\}$ is translated into a replication where instance variable z is mapped to a value of a reference and methods are mapped as a replicated branching. Hereafter we explicitely declare each branching with a label l_n such as $a[\&_i l_i : (\vec{x}_i).P_i]$, and write $\overline{a}.l_k\langle\vec{V}_k\rangle$ as $\overline{a}\mathtt{in}_k\langle\vec{V}_k\rangle$ [19, 54].

$$\begin{aligned}[\![\mathtt{Class\ Mcal}\{...\}]\!] \stackrel{\text{def}}{=}\ & !\mathtt{Mcal}(z,y).\overline{y}(f)\\ & (\nu c)(\mathsf{ref}\langle cz\rangle \mid\\ & \quad !f[\mathtt{twice} : (x,e).[\![\mathtt{twice}(\mathtt{int}\ x)...]\!]_e\{c/z\}\\ & \quad\quad \&\ \mathtt{thrice} : (x,e).[\![\mathtt{thrice}(\mathtt{int}\ x)...]\!]_e\{c/z\}])\end{aligned} \tag{23}$$

where we assume c is fresh. Then Code A is translated into:

$$[\![\mathrm{A}]\!]_g \stackrel{\text{def}}{=} \overline{\mathtt{Mcal}}\langle 0,(c)\rangle c(z).\overline{\mathtt{myobj}}.\mathtt{ack}(c')c'.\overline{z}\mathtt{twice}\langle 777,(e)\rangle e(z').\overline{g}\langle z'\rangle$$

Notice that in the above code, $\overline{z}\mathtt{twice}\langle 777,e\rangle$ will activate the branch `twice` in the mapping of `Mcal`. The thunk $\ulcorner \mathrm{A}\ \mathtt{with}\ \mathtt{CT} \urcorner$ with $\mathtt{CT} = \mathtt{Mcal} \mapsto \mathtt{Class\ Mcal}\{...\}$ is now encoded as

$$[\![\ulcorner \mathrm{A}\ \mathtt{with}\ \mathtt{CT}\urcorner]\!] \stackrel{\text{def}}{=} \lambda(g).(\nu\mathtt{Mcal})([\![\mathrm{A}]\!]_g \mid [\![\mathtt{Class}\ \mathtt{Mcal}\{...\}]\!])$$

which is initialised by a local name in the remote location (2), then is executed. Now the serialisation in Location (1) is translated as:

$$P \stackrel{\text{def}}{=} \overline{r}.\mathtt{run}(z)z(c).\overline{c}\langle[\![\ulcorner \mathrm{A}\ \mathtt{with}\ \mathtt{CT}\urcorner]\!],(e)\rangle e(x).\overline{w}\mathtt{write}\langle x,(c')\rangle...$$

Note that the thunk code is directly transferred together with the class table to be executed in Location (2). Further we assume the object `myobj` is located in Location (1). Hence the code in Location (1) is translated into:

$$l_1[(\nu\mathtt{myobj})(P \mid\ !\mathtt{myobj}[\mathtt{ack}:(y).(\overline{y}\mid ...)\&...])]$$

Note the object id `myobj` is restricted by ν. This is scope-opened via a transmission of the thunk. Now the receiver which deseliarises the thunk is encoded as:

$$l_2[!\,r:\mathtt{run}(z\!:\!\tau).\overline{z}(c).!c(x,e).(\nu g)(xg \mid g(z).\overline{e}\langle x\rangle)]$$

This receives the thunk type and executes it by applying the hidden channel g, then the result is returned to Location (1) via channel e.

ASSIGNING FINE-GRAINED TYPES We now assign the fine-grained types to the above encodings. Before that, we informally give structured types for the program, following [8, 31]. First the methods, `twice` and `thrice`, have type $\mathtt{T} = \mathtt{int} \Rightarrow \mathtt{double}$; then class `Mcal` has type $\mathtt{double} \Rightarrow [\mathtt{T\&T}]$ where $[\mathtt{T\&T}]$ represents the standard record type. Since the action $\mathtt{new}(V)$ always terminates (because we can observe in the mapping of `Mcal` in (23), an answer at y is always immediately returned to the caller), we assign the following linear type to channel `Mcal` in (23).

$$(\mathtt{double},[\mathtt{T\&T}]^{\bullet})^{!_{\mathrm{L}}} \text{ with } [\mathtt{T\&T}]^{\bullet} \stackrel{\text{def}}{=} ([\mathtt{T\&T}]^{\circ})^{\uparrow_{\mathrm{L}}}$$

Note this translation follows essentially the same idea in those defined in §. 7.

Depending on immutability/mutability and termination/non-termination of the methods, we assign either $!_{\mathrm{M}}$, $!_{\mathrm{A}}$ or $!_{\mathrm{L}}$ to an encoding of the methods types. Here the encoding in (22) includes mutable action at z, hence we assign the following mutable type to f in (23).

$$[\mathtt{T\&T}]^{\circ} = [\mathtt{int}(\mathtt{double})^{\uparrow_{\mathrm{A}}}\ \&\ \mathtt{int}(\mathtt{double})^{\uparrow_{\mathrm{A}}}]^{!_{\mathrm{M}}}$$

As such, we can inductively map the type of the language into the linear/affine types of $\lambda\pi^{\mathrm{LAM}}_{<}$. Now we assign fine-grained types for the encoding of the thunk $\ulcorner \mathrm{A}\ \mathtt{with}\ \mathtt{CT}\urcorner$ as follows.

$$\tau \stackrel{\text{def}}{=} \exists[\mathtt{myobj}\!:\!\overline{\mathtt{T}^{\circ}_{o}}]\Pi(g\!:\!(\mathtt{double})^{\uparrow_{\mathrm{A}}})\{g\!:\!(\mathtt{double})^{\uparrow_{\mathrm{A}}},\mathtt{myobj}\!:\!\overline{\mathtt{T}^{\circ}_{o}}\}$$

where $\mathtt{T}_o = [\mathtt{void}\&..]$ is a type of `myobj`. This type represents (a) the scope of object id `myobj` is opened via code distribution and (b) the host at Location (2) can detect the code which does harm the local resources via annotating type τ (because all names of process types are bound). Program analysis via encodings of $\lambda\pi^{\mathrm{LAM}}_{<}$ briefly demonstrated in this subsection is useful: Part III [67, 69] shows it can develop a new type discipline for

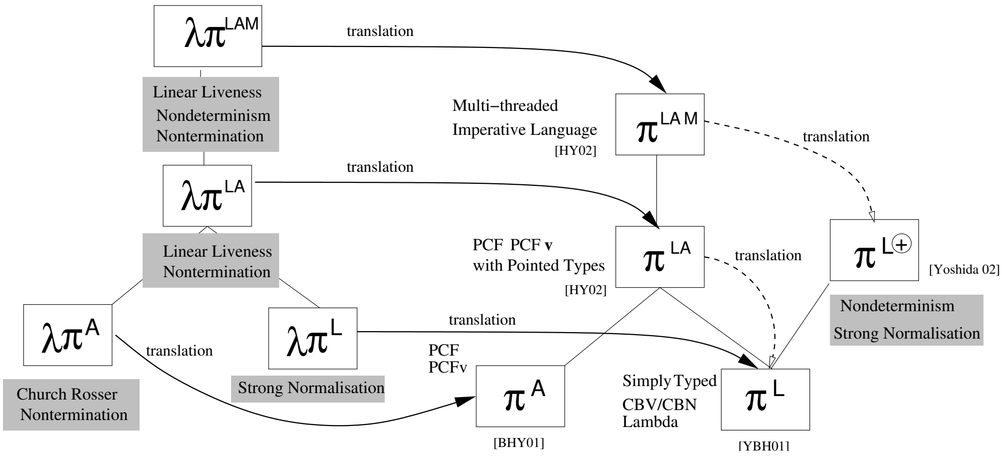

FIGURE 16. A Family of Linear/Affine λπ and π-Calculi

a multi-threaded class-based language with code distribution and dynamic class loading enforcing secrecy and accessibility [8, 27, 31].

9 Discussion

9.1 Summary

This paper introduced a new type theory for controlling dynamic change of process accessibility and safe scope-extrusion by means of channel dependent and existential types. Its expressiveness is demonstrated by articulating a significant class of distributed computational behaviour via a systematic integration with the linear/affine type disciplines of the π-calculus. Such a systematic integration is essential for a precise embedding and analysis of a distributed language (cf. § 8). Figure 16 shows the relationship between the results of this paper, including the corresponding types π-calculi.

In Figure 16, each box lists the typed π/λπ-calculus ("L", "A", "M" and "⊕" mean linear, affine, reference (mutability), and summation respectively). The right-hand side of the box shows systems we can embed in the calculus. The grey box shows a basic property satisfied by the calculus. We also show the relationship of translations from the basic calculi to target calculi for proving the liveness properties (note that the liveness results of the π-calculus have been proved in [62]). The translations denoted by the black arrows from the λπ into the corresponding π are type-preserving and sound (cf. Proposition 6.2 6) with respect to the contextual congruence, while those by the dotted arrows (from π^{LA} to π^{L} and from π^{LAM} to $\pi^{L}\oplus$) are not, but are preserving a causality towards the linear output, see [62]. We also believe incorporation of second-order polymorphism along the line of [7] would be straightforward.

There have been a wide range of studies of mobile and distributed computation on process calculi [5, 10, 12, 13, 15, 22, 45, 47, 50, 51, 53, 60]. In the following we give comparisons with closely related work among them, concentrating on the basic typed formalism for code mobility.

M-calculus [53] 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 [65]; the channel types based on the first order polymorphism in [53] can control higher-order communication and guarantee the unicity of local names. This property is essentially a 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 and a more general form of parameterised code with fine-grained process types; this generality enables to integrate various affine/linear typing structures systematically, and use a typed $\lambda\pi$ as a basic meta-calculus to embed and examine known programming language constructs, extending mobility primitives in §8 (cf. [67, 69]). Relatedly in the technical report version of [65], we discuss expressiveness of our fine-grained types means that channel locality and global/local channel capability controls [50] can be easily and naturally enforced in the presence of higher-order code mobility; specifically it is unnecessary to annotate higher-order values as being *sendable* [12, 53]. However, such a flexible typing system is only meaningful without using `proc`. In turn, many sendable terms in [64] could not be even typable in [65]. The new channel dependent/existential types in this paper solve this problem, since, there is no requirement to use `proc`.

The Seal calculus [12] uses named process types for representing agents accessibility in the presence of nested locations. Their typing system only allows input channel types appears in the process types, hence is unable to control the fine-grained output accessibility like ours. Also due to the similar typability problem with ours (which was closed by channel dependent types in this paper), the resulting typed calculus disallows input on channel names bound by an input action (note that our simple IO-typing system in § 2 and § 3 does not require this constraint). The type systems for mobility is also actively studied in the Ambient calculus [10] and its variants. The work on types for ambients relies on group types which represent a set of ambient names with explicit declaration of ambients movements. Recently [36] proposes a dependency typing system on ambient names which can more flexibly and directly represent the same specification written by group types. We believe our general type theory, together with new channel dependent/existential types, are applicable to the formalisms where agents movements between hierarchical locations are controlled by types with names [12, 36, 53].

The Nomadic Pict [60] is a located π-calculus which provides two levels of communication primitives, high-level location independent primitives and low-level location dependent primitives where subtyping is used for representing location-based access rights. The boxed π-calculus [51] 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. These studies do not include higher-order process passing, which are the main aim of using the $\lambda\pi$ in the present work. The use of safe indirection for composition studied in [51] in the present work, extending general higher-order mobility, would be an interesting further topic.

$D\pi$ [45] proposed a framework where partial type checking prevents migration of code from untrusted locations using location-based types. In § 4, we already observed some of the dynamic program detection mechanism enabled by partial type checking in [45] can be enforced by static type checking with the new dependent/existential types. [20] also shows these type structures can naturally integrated with the location types in [45] in a non-trivial higher-order extension of $D\pi$, offering a powerful static type checking framework for dynamically locating process parameterised by sites. While our aim is to provide the maximum static type checking framework for safe code mobility, practically, type checking would be done at the time of arrival of the code, either by dynamically scanning the code [57] or by checking the signed specification [41]. In this context. sending higher-order code attached with fine-grained process types would offer a firm technical basis which can be smoothly mixed with dynamic checking.

In the foregoing studies of the higher-order mobile processes [33, 47, 48], 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 this paper shows this methodology can be inherited in the setting of strong type discipline, with an added benefit of more faithful operational correspondence including preserving high-level and low-level effects of mobile code [60], which may be useful for various applications.

Appendix

A Section 2

In this section we show the axially definitions and proofs omitted from Section 2.

A.1 Size of Types and Judgements

1. The size of type expression α, $\mathrm{size}_\Gamma(\alpha)$ is defined as follows.
   (a) $\mathrm{size}(\mathtt{unit}) = \mathrm{size}(\mathtt{nat}) = 2$.
   (b) $\mathrm{size}(\tau \to \tau') = \mathrm{size}(\tau) + \mathrm{size}(\tau') + 1$.
   (c) $\mathrm{size}(\Pi(x\!:\!\sigma)\tau) \;=\; \mathrm{size}(\sigma) + \mathrm{size}(\tau) + 1$.
   (d) $\mathrm{size}(\{u_1\!:\!\sigma_1,...,u_1\!:\!\sigma_1\}) \;=\; \sum_{1\le i\le n} \mathrm{size}(\sigma_i) + 2$.
   (e) $\mathrm{size}((\Pi[x_1\!:\!\sigma_1,...,x_n\!:\!\sigma_n]\tau_1,...,\tau_m)^p) \;=\; \sum_{1\le i\le n} \mathrm{size}(\sigma_i) + \sum_{1\le j\le m} \mathrm{size}(\tau_j) + 2$.
   (f) $\mathrm{size}(\langle\sigma_1,\sigma_2\rangle) \;=\; \mathrm{size}(\sigma_1) + \mathrm{size}(\sigma_2) + 2$.
2. The size of general environments, $\mathrm{size}(\Gamma)$, is defined as (d).
3. The size of kinding and environment judgement $\mathtt{J}$, $\mathrm{size}(\mathtt{J})$, is defined as follows.
   (a) $\mathrm{size}(\Gamma \vdash \mathtt{Env}) = \mathrm{size}(\Gamma) + 1$.
   (b) $\mathrm{size}(\Gamma \vdash \tau : \mathtt{tp}) = \mathrm{size}(\Gamma) + \mathrm{size}(\tau)$.

(c) $\mathrm{size}(\Gamma \vdash u\!:\!\sigma) = \mathrm{size}(\Gamma) + \mathrm{size}(\sigma)$.

This size function is the measure we refer to when talking of a "smaller" size of judgement. Then the proof of Proposition 2.2 is straightforward since $\asymp$ and $\{V/x\}$ are defined from smaller size of types. We also have the following lemma whose proof is straightforward.

LEMMA A.1. (well-formedness)

1. *For every* $\mathtt{J}$ *in Figure 6,* $\mathit{size}(\emptyset \vdash \mathtt{Env}) \leq \mathit{size}(\mathtt{J})$.
2. *If* $\frac{\mathtt{J}_1,\ \mathtt{J}_2,\ \ldots,\ \mathtt{J}_n}{\mathtt{J}}$ *in Figure 6 except (t-dual), then* $\mathit{size}(\mathtt{J}_i) \lneq \mathit{size}(\mathtt{J})$.

Note $\overline{\sigma}$ in (t-dual) is inductively defined from strictly smaller types. Thus by the above lemma, we can straightforwardly obtain Proposition 2.4.

A.2 The Minimum Typing System

This subsection starts from constructing the syntax direct typing system, denoted by $\Gamma \vdash_{min} P : \tau$. This is defined by deleting (Weak) and replacing (Res) and (In) to the following rules (below Δ^{-u} means $u \notin \mathsf{fnv}(\Delta)$).

$$(\mathsf{Res})\ \frac{\Gamma, a\!:\!\sigma \vdash_{min} P : \Delta^{-a}, \Delta' \quad \Delta' \subseteq a\!:\!\sigma}{\Gamma \vdash_{min} (\nu a\!:\!\sigma)P : \Delta} \qquad (\mathsf{In})\ \frac{\Gamma \vdash_{min} u\!:\!(\Pi[\vec{x}\!:\!\vec{\sigma}]\vec{\tau})^{\mathtt{I}} \quad \Gamma, \vec{x}\!:\!\vec{\sigma}, \vec{y}\!:\!\vec{\tau} \vdash_{min} P : \Delta^{-\vec{x}}, \Delta' \quad \Delta' \subseteq \vec{x}\!:\!\vec{\sigma}}{\Gamma \vdash_{min} u(\vec{x}\!:\!\vec{\sigma}, \vec{y}\!:\!\vec{\tau}).P :\ u\!:\!(\Pi[\vec{x}\!:\!\vec{\sigma}]\vec{\tau})^{\mathtt{I}} \odot \Delta}$$

Others are from Figure 7 by replacing $\vdash$ by $\vdash_{min}$. Then we can easily check:

LEMMA A.2. (minimum type checking)

1. *Suppose* $\Gamma \vdash_{min} P : \tau$. *Then* $\Gamma \vdash P : \tau$.
2. (unique type) *Suppose* $\Gamma \vdash_{min} P : \tau_i$. *Then* $\tau_1 = \tau_2$.
3. (*min* has a smaller type) *If* $\Gamma \vdash P : \Delta$, *then there exists the unique* $\Delta_{min} \subseteq \Delta$ *such that* $\Gamma \vdash_{min} P : \Delta_{min}$.

By the above lemma, we can straightforwardly prove Proposition 2.7 (1,2) and Proposition 2.10 (1).

A.3 Proofs for Substitution Lemma

In this subsection, we prove the substitution lemma (Lemma 2.7 (4)). Note that the only difference from [65] is related to a channel dependent type. Hence we only have to prove the case of inputs and outputs. The proof is by induction on the judgement of $\Gamma, x\!:\!\sigma, \Gamma' \vdash P : \tau$ and an examination of the last inference rule used.

Suppose the last inference rule is (In) in Figure 7. Here P has the form $u(\vec{y}\!:\!\vec{\sigma}, \vec{z}\!:\!\vec{\tau}).Q$ and we have

$$\Gamma, x\!:\!\tau, \Gamma' \vdash u\!:\!(\Pi[\vec{y}\!:\!\vec{\sigma}]\vec{\tau})^{\mathtt{I}} \quad \text{and} \quad \Gamma, x\!:\!\tau, \Gamma', \vec{y}\!:\!\vec{\sigma}, \vec{z}\!:\!\vec{\tau} \vdash Q :\ \vec{y}\!:\!\vec{\sigma}, \Delta$$

for some Δ and $x \notin \{\vec{y}\}$. Then by induction, we have

$\Gamma, \Gamma'\{V/x\} \vdash u\{V/x\} : (\Pi[\vec{y}\!:\!\vec{\sigma}\{V/x\}]\vec{\tau}\{V/x\})^{\mathtt{I}}$ and
$\Gamma, \Gamma'\{V/x\}, \vec{y}\!:\!\vec{\sigma}\{V/x\}, \vec{z}\!:\!\vec{\tau}\{V/x\} \vdash Q\{V/x\} :\ \vec{y}\!:\!\vec{\sigma}\{V/x\}, \Delta\{V/x\}$

Note that $(u(\vec{y}{:}\vec{\sigma},\vec{z}{:}\vec{\tau}).Q)\{V/x\} = u\{V/x\}(\vec{y}{:}\vec{\sigma}\{V/x\},\vec{z}{:}\vec{\tau}\{V/x\}).(Q\{V/x\})$ and $(\Pi[\vec{y}{:}\vec{\sigma}]\vec{\tau})^{\mathtt{I}}\{V/x\} = (\Pi[\vec{y}{:}\vec{\sigma}\{V/x\}]\vec{\tau}\{V/x\})^{\mathtt{I}}$, and $(\Delta \odot u{:}(\Pi[\vec{y}{:}\vec{\sigma}]\vec{\tau})^{\mathtt{I}})\{V/x\} = \Delta\{V/x\} \odot u{:}(\Pi[\vec{y}{:}\vec{\sigma}]\vec{\tau})^{\mathtt{I}}\{V/x\}$ by definition. Then an application of (In) gives

$$\Gamma,\Gamma'\{V/x\} \vdash (u(\vec{y}{:}\vec{\sigma},\vec{z}{:}\vec{\tau}).Q)\{V/x\} : \Delta\{V/x\} \odot (\Pi[\vec{y}{:}\vec{\sigma}]\vec{\tau})^{\mathtt{I}}\{V/x\}$$

as required.

Next suppose P has a form $\overline{u}\langle\vec{v},\vec{V}\rangle$ and the last inference rule used is (Out). Then we have:

$$\Gamma,x{:}\tau,\Gamma' \vdash u{:}(\Pi[\vec{y}{:}\overline{\vec{\sigma}}]\vec{\tau})^{\mathtt{O}} \quad \Gamma,x{:}\tau,\Gamma' \vdash V_j : \tau_j\{\vec{v}/\vec{y}\} \quad \text{and} \quad \Gamma,x{:}\tau,\Gamma' \vdash v_i : \sigma_i\{\vec{v}/\vec{y}\}$$

We assume $x \notin \{\vec{y}\}$. By induction, we have:

$$\begin{array}{l}
\Gamma,\Gamma'\{V/x\} \vdash u\{V/x\}{:}(\Pi[\vec{y}{:}\overline{\vec{\sigma}}\{V/x\}]\vec{\tau}\{V/x\})^{\mathtt{O}} \\
\Gamma,\Gamma'\{V/x\} \vdash V_j\{V/x\} : \tau_j\{\vec{v}/\vec{y}\}\{V/x\} \quad \text{and} \\
\Gamma,\Gamma'\{V/x\} \vdash v_i\{V/x\} : \sigma_i\{\vec{v}/\vec{y}\}\{V/x\}
\end{array}$$

Note that $(\overline{u}\langle\vec{v},\vec{V}\rangle)\{V/x\} = \overline{u}\{V/x\}\langle\vec{v}\{V/x\},\vec{V}\{V/x\}\rangle$. Hence the application of (Out) gives the following required result.

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

A.4 Proof of Type-Safety

In the following, we prove the subject reduction theorem (Theorem 2.8 (2)). We only prove the case (com). Others are the same as [65]. For simplicity we only treat the two arguments, one is a channel and another is a higher-order value; the extension to the general case is straightforward. Assume

$$a(x{:}\sigma,y{:}\tau).P \,|\, \overline{a}\langle v,V\rangle \longrightarrow P\{v/x\}\{V/y\}$$

and the hypothesis is $\Gamma \vdash a(x{:}\sigma,y{:}\tau).P \,|\, \overline{a}\langle v,V\rangle : \Delta$ for some Δ. We need to show $\Gamma \vdash P\{v/x\}\{V/y\} : \Delta$. Analysing the hypothesis we obtain:

$$\begin{array}{ll}
\Gamma,x{:}\sigma,y{:}\tau \vdash P : \Delta_1,x{:}\sigma & \text{with } \Gamma \vdash \ a{:}(\Pi[x{:}\sigma]\tau)^{\mathtt{I}} \\
\Gamma \vdash v : \sigma, \quad \Gamma \vdash V : \tau\{v/x\} & \text{with } \Gamma \vdash \ a{:}(\Pi[x{:}\overline{\sigma}]\tau)^{\mathtt{O}}
\end{array}$$

with $\Delta_1 \subseteq \Delta$ and $\{a{:}\sigma_a, v{:}\sigma\} \subseteq \Delta$. and $\sigma_a = \langle(\Pi[x{:}\sigma]\tau)^{\mathtt{I}}, (\Pi[x{:}\overline{\sigma}]\tau)^{\mathtt{O}}\rangle$. By applying Substitution Lemma to x and v, we have:

$$\Gamma,y{:}\tau\{v/x\} \vdash P\{v/x\} : \Delta_1 \odot v{:}\sigma$$

Next we apply Substitution Lemma to y and V, noting $y \notin \mathsf{fv}(\Delta_1,\sigma)$. Then we have:

$$\Gamma \vdash P\{v/x\}\{V/y\} : \Delta_1 \odot v{:}\sigma$$

Now the application of (Weak) gives us $\Gamma \vdash P\{v/x\}\{V/y\} : \Delta$, as required.

B Section 3

In this section, we prove the substitution lemma, Lemma 3.2 (5,6). Note that (3-6) are proved simultaneously.

We first analyse (5). Suppose $\Gamma \vdash \{w,W\} : \exists[x{:}\sigma]\tau$ and $\Gamma,\{x{:}\sigma,y{:}\tau\},\Gamma' \vdash \{v,V\} : \exists[z{:}\sigma']\tau'$. There are two cases.

First suppose $x = v$. Here we have $\{v,V\}$ is derived by (Tpl∃). Hence we know $V = y$, $\sigma' = \sigma$ and $\tau' = \tau$. We also know $\Gamma \vdash \{w,W\} : \exists[x:\sigma]\tau$ is inferred by either (Tpl∃) or (Pack). The former case is proved by (3) and (4). So suppose the latter case. Then we know $\Gamma \vdash W : \tau\{w/x\}$, hence done.

Next suppose $x \neq v$. This means $\{v,V\}$ is derived by (Pack) as follows.

$$\Gamma,\{x:\sigma,y:\tau\},\Gamma' \vdash v : \sigma' \qquad \text{and} \qquad \Gamma,\{x:\sigma,y:\tau\},\Gamma' \vdash V : \tau'\{v/z\}$$

By induction, we have:

$$\Gamma,\Gamma' \vdash v : \sigma'\{w/x\} \qquad \text{and} \qquad \Gamma,\Gamma' \vdash V\{W/y\} : \tau'\{v/z\}\{w/x\}$$

Since $\tau'\{v/z\}\{w/x\} = \tau'\{w/x\}\{v/z\}$, an application of (Pack) gives us:

$$\Gamma,\Gamma' \vdash \{v,V\{W/y\}\} : \exists[z:\sigma'\{w/x\}]\tau'\{w/x\}$$

as required.

Next we prove (6). We prove the case that the last applied rule is (Val∃), (In∃) or (Out∃). Others are straightforward by induction.

Suppose the last rule is (Val∃) and $P = y'$.

$$\Gamma,\{x:\sigma,y:\tau\},\Gamma' \vdash y' : \tau'$$

This is derived from the assumption $\Gamma,\{x:\sigma,y:\tau\},\Gamma' \vdash \mathtt{Env}$. Since $x \notin \mathsf{fv}(\Gamma')$, by Narrowing, we have: $\Gamma,\Gamma' \vdash \mathtt{Env}$. Then there are two cases.

Suppose $y' \neq y$. By construction, we know $x \notin \mathsf{fv}(\tau')$ and $\Gamma,\Gamma' \vdash y : \tau'$. Hence done.

Now suppose $y' = y$. Then we know $\tau = \tau'$. By assumption, $\Gamma \vdash V : \tau\{v/x\}$. By Weakening, we have $\Gamma,\Gamma' \vdash V : \tau\{v/x\}$. Hence $\Gamma,\Gamma' \vdash y\{V/y\} : \tau\{v/x\}$, as required.

We prove the case (In∃). Here $P = u(z_2:\exists[z_1:\sigma_1]\tau_2).Q$. This is derived from:

$$\Gamma,\{x:\sigma,y:\tau\},\Gamma' \vdash u:(\exists[z_1:\sigma_1]\tau_2)^{\mathtt{I}} \quad \text{and} \quad \Gamma,\{x:\sigma,y:\tau\},\Gamma',\{z_1:\sigma_1,z_2:\tau_2\} \vdash Q : \Delta,z_1:\sigma_1$$

Note $x \notin \mathsf{fv}(\sigma_1,\tau_2)$ by formation rule. By inductive hypothesis, we have:

$$\Gamma,\Gamma' \vdash u:(\exists[z_1:\sigma_1]\tau_2)^{\mathtt{I}} \quad \text{and} \quad \Gamma,\Gamma',\{z_1:\sigma_1,z_2:\tau_2\} \vdash Q\{V/y\} : \Delta\{v/x\},z_1:\sigma_1$$

By applying (In∃) to the above, we have:

$$\Gamma,\Gamma' \vdash (u(z_2:\exists[z_1:\sigma_1]\tau_2).Q)\{V/y\} : \Delta\{v/x\} \odot u:(\exists[z_1:\sigma_1]\tau_2)^{\mathtt{I}}$$

We also examine the output rule, (Out∃). Suppose $P = \overline{u}\langle W\rangle$. Then this is derived from

$$\Gamma,\{x:\sigma,y:\tau\},\Gamma' \vdash u:(\exists[z_1:\overline{\sigma_1}]\tau_2)^{\mathtt{O}} \quad \text{and} \quad \Gamma,\{x:\sigma,y:\tau\},\Gamma' \vdash \{w,W\} : \exists[z_1:\sigma_1]\tau_2$$

for some w. Note $x \notin \mathsf{fv}(\sigma_1,\tau_2)$. By inductive hypothesis, we have:

$$\Gamma,\Gamma' \vdash u:(\exists[z_1:\overline{\sigma_1}]\tau_2)^{\mathtt{O}} \quad \text{and} \quad \Gamma,\Gamma' \vdash \{w\{v/x\},W\{V/y\}\} : \exists[z_1:\sigma_1]\tau_2$$

Then an application of (Out∃) gives us:

$$\Gamma,\Gamma' \vdash \overline{u}\langle W\{V/y\}\rangle : u : (\exists[z_1:\sigma_1]\tau_2)^{\mathtt{O}}, w\{v/x\}:\sigma_1$$

as desired.

C Section 5

C.1 Proof of Substitution Lemma

We prove 5.1 (2) and (3). We use the following lemma. The proof is straightforward by induction.

LEMMA C.1.

1. $\Gamma \vdash u:\sigma$ *and* $\Gamma, x:\sigma, \Gamma' \vdash \mathtt{Env}$ *imply* $\Gamma, \Gamma'\{u/x\} \vdash \mathtt{Env}$.
2. $\Gamma \vdash u:\sigma$ *and* $\Gamma, x:\sigma, \Gamma' \vdash \tau : \mathtt{tp}$ *imply* $\Gamma, \Gamma'\{u/x\} \vdash \tau\{u/x\} : \mathtt{tp}$.
3. $\Gamma \vdash u:\sigma$ *and* $\Gamma, x:\sigma, \Gamma' \vdash \sigma' : p$ *imply* $\Gamma, \Gamma'\{u/x\} \vdash \sigma'\{u/x\} : p$.
4. $\Gamma \vdash u:\sigma$, $\Gamma, x:\sigma, \Gamma' \vdash \sigma_{1,2} : p_{1,2}$ *and* $\sigma_1 \asymp \sigma_2$ *imply* $\sigma_1\{u/x\} \asymp \sigma_2\{u/x\}$.

Note that if we replace $\mathtt{tp}$ by $\mathtt{tp}_P$ in (1), then the above lemma no longer holds. For (2), the only interesting case is the last applied rule is (Par). Others are special cases of (3). Suppose $P = P_1 \,|\, P_2$. Then we have:

$$\Gamma, x:\sigma, \Gamma' \vdash P_{1,2} : \Delta_{1,2} \qquad \text{with } \Delta_1 \asymp \Delta_2$$

By assumption, we set $\Delta_1 = \Delta_1', x:\sigma$ and $x \notin \mathsf{dom}(\Delta_1', \Delta_2)$ (otherwise $\Delta_1 \not\asymp \Delta_2$). Then analysing the inductive hypothesis, we have:

$$\Gamma, \Gamma'\{u/x\} \vdash P_1\{u/x\} : \Delta_1'\{u/x\}, u:\sigma \quad \text{and} \quad \Gamma, \Gamma'\{u/x\} \vdash P_2\{u/x\} : \Delta_2\{u/x\}$$

Now we need to prove $\Delta_1\{u/x\} (= \Delta_1'\{u/x\}, u:\sigma) \asymp \Delta_2\{u/x\}$. By (t-proc), we know $\Gamma, x: \sigma, \Gamma' \vdash \Delta_i(w) : p_{wi}$ for each $w \in \mathsf{dom}(\Delta_i)$. Hence by Lemma C.1 (3), we have $\Gamma, \Gamma'\{u/x\} \vdash \Delta_i(w)\{u/x\} : p_{wi}$. Then by Lemma C.1 (4), $\Delta_1(w)\{u/x\} \asymp \Delta_2(w)\{u/x\}$ for each $w \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)$. Thus we have $\Delta_1\{u/x\} \asymp \Delta_1\{u/x\}$. Now we derive:

$$\Gamma, \Gamma'\{u/x\} \vdash P_1\{u/x\} \,|\, P_2\{u/x\} : (\Delta_1' \odot \Delta_2)\{u/x\}, u:\sigma$$

as desired. For (3), we prove the case that the last applied rule is (App_N). Assume $P = \lambda(y:\sigma')Q$. Then we have:

$$\Gamma, x:\sigma, \Gamma' \vdash \Pi(y:\sigma')\tau : \mathtt{tp}_H \quad \text{and} \quad \Gamma, x:\sigma, \Gamma', y:\sigma' \vdash Q \;:\; \tau$$

First, by Lemma C.1 (2), we have

$$\Gamma, \Gamma'\{u/x\} \vdash \Pi(y:\sigma'\{u/x\})\tau\{u/x\} : \mathtt{tp}_H \tag{24}$$

Secondly, noting $\Gamma, x:\sigma, \Gamma', y:\sigma' \vdash \tau : \mathtt{tp}_H$, by inductive hypothesis of Lemma C.1 (4), we have:

$$\Gamma, \Gamma'\{v/x\}, y:\sigma'\{v/x\} \vdash Q\{v/x\} \;:\; \tau\{v/x\} \tag{25}$$

Hence an application (App_N) to (24) and (25) above gives us:

$$\Gamma, \Gamma'\{v/x\} \vdash \lambda(y:\sigma'\{v/x\})Q\{v/x\} \;:\; \Pi(y:\sigma'\{v/x\})\tau\{v/x\}$$

The case (App_H) is just similar. Also the cases ($\mathsf{In}^{\downarrow}$) and ($\mathsf{In}^{!}$) are similarly proved by applying Lemma C.1 (3) instead of (4). Other rules are proved as Lemma 2.7.

C.2 Proof of Type-Safety

In the following, we prove the subject reduction theorem (Theorem 5.2). We only prove the case (rep). For simplicity we only treat the two arguments, one is an affine output channel and another is a higher-order value; other rules are straightforward using the substitution lemmas. Assume

$$!a(x\!:\!\sigma,y\!:\!\tau).P\,|\,\overline{a}\langle v,V\rangle \longrightarrow P\{v/x\}\{V/y\}\,|\,!a(x\!:\!\sigma,y\!:\!\tau).P$$

and the hypothesis is $\Gamma\vdash\,!a(x\!:\!\sigma,y\!:\!\tau).P|\overline{a}\langle v,V\rangle : \Delta$ for some Δ. We need to show $\Gamma\vdash P\{v/x\}\{V/y\}\,|\,!a(x\!:\!\sigma,y\!:\!\tau).P : \Delta$. Analysing the hypothesis we obtain:

$$\begin{array}{ll}\Gamma,x\!:\!\sigma,y\!:\!\tau\vdash P : ?\Delta_1,x\!:\!\sigma & \text{with } \Gamma\vdash\ a\!:\!(\Pi[x\!:\!\sigma]\tau)^{!}\\ \Gamma\vdash v:\sigma,\quad \Gamma\vdash V:\tau\{v/x\} & \text{with } \Gamma\vdash\ a\!:\!(\Pi[x\!:\!\overline{\sigma}]\tau)^{?}\end{array}$$

with $\Delta=\Delta',a\!:\!\langle\sigma_a,\overline{\sigma_a}\rangle$, $\Delta_1\subseteq\Delta'$, $\{v\!:\!\sigma\}\subseteq\Delta'$, and $\sigma_a=(\Pi[x\!:\!\sigma]\tau)^{!}$. Note that $x\notin\mathsf{fv}(\Delta_1)$ and $v\notin\mathsf{dom}(\Delta_1)$. Hence we can apply the substitution lemma, Lemma C.1 (3) to x and v to obtain:

$$\Gamma,y\!:\!\tau\{v/x\}\vdash P\{v/x\} : \Delta_1,v\!:\!\sigma$$

Next we apply the standard substitution lemma to y and V, noting $y\notin\mathsf{fv}(\Delta_1,\sigma)$.

$$\Gamma\vdash P\{v/x\}\{V/y\} : \Delta_1,v\!:\!\sigma$$

Note $\Delta_1\asymp\Delta_1$ and $\Delta_1\odot\Delta_1=\Delta_1$, hence $(a\!:\!\sigma_a,\Delta_1)\asymp(\Delta_1,v\!:\!\sigma)$ and $(a\!:\!\sigma_a,\Delta_1)\odot(\Delta_1,v\!:\!\sigma)=(a\!:\!\sigma_a,\Delta_1,v\!:\!\sigma)$. Hence we can apply (Par) to obtain:

$$\Gamma\vdash P\{v/x\}\{V/y\}\,|\,!a(x,y).P\ :\ \Delta_1,v\!:\!\sigma,a\!:\!\sigma_a$$

Now the application of (Weak) gives us $\Gamma\vdash P\{v/x\}\{V/y\}\,|\,!a(x,y).P\ :\Delta$, as required.

D Section 7

First we define the transition relation in Figure 17. Then we have:

LEMMA D.1. *The rules in in Figure 17 are well-typed in the sense that if* $\Gamma_1\vdash P_1:\Delta_1$ *is well-typed and* $\Gamma_1\vdash P_1:\Delta_1\stackrel{l}{\longrightarrow}\Gamma_2\vdash P_2:\Delta_2$ *then* $\Gamma_2\vdash P_2:\Delta_2$ *is well-typed.*

Now we prove Theorem 7.3 using the above lemma. W.o.l.g. we prove the case $P\equiv !u(y\!:\!\tau).R$ with $\tau=(\exists[c\!:\!(\mathtt{nat})^{\uparrow_{\mathrm{L}}}]\ulcorner c\!:\!(\mathtt{nat})^{\uparrow_{\mathrm{L}}}\urcorner)^{!_{\mathrm{L}}}$. Assume $P\stackrel{u\langle\ulcorner\overline{c}\langle n\rangle\urcorner\rangle}{\longrightarrow}P'$. Then we can write $P'\equiv(R\{\ulcorner\overline{c}\langle n\rangle\urcorner/y\}\,|\,P)$. Let $R'\stackrel{\text{def}}{=}R\{\ulcorner\overline{c}\langle n\rangle\urcorner/y\}$. Then we know $\Gamma\vdash R':c\!:\!(\mathtt{nat})^{\uparrow_{\mathrm{L}}}$ by the lemma above. Hence we know $R'\Downarrow c^{\uparrow_{\mathrm{L}}}$ by Theorem 6.3 (3). Here we also need to check R' emits n via c. Then by $(\mathsf{In}^{\exists})$ (adapting to $(\mathsf{In}^{!_{\mathrm{L}}})$), we know $\Gamma,\{x\!:\!(\mathtt{nat})^{\uparrow_{\mathrm{L}}},y\!:\!\tau'\}\vdash R:x\!:\!(\mathtt{nat})^{\uparrow_{\mathrm{L}}}$ with $\tau'=\ulcorner x\!:\!(\mathtt{nat})^{\uparrow_{\mathrm{L}}}\urcorner$. Note that by Proposition 3.2 (2), $x\notin\mathsf{fv}(R)$. Hence $\neg R\Downarrow x^{\uparrow_{\mathrm{L}}}$. We also know $\neg R\Downarrow c^{\uparrow_{\mathrm{L}}}$ by the definition of $\stackrel{l}{\longrightarrow}$ in Figure 17. Thus if $R'\to\to(\nu\vec{a})(\overline{c}\langle V\rangle\,|\,R'')$, then $\overline{c}\langle V\rangle$ should be created from the input transition $l=u\langle\ulcorner\overline{c}\langle n\rangle\urcorner\rangle$, which means $m=n$.

D.1 Section 7.4

We define the subtyping system in Figure 18 and the partial meet and join operators in Figure 19.

$$\Gamma \vdash u(\vec{x},\vec{y}).P : \Delta \xrightarrow{u\langle\vec{v},\vec{V}\rangle} \Gamma \vdash P\{\vec{v}/\vec{x}\}\{\vec{V}/\vec{y}\} : \Delta/u \odot \vec{v}{:}\vec{\sigma} \qquad (\Gamma \vdash u{:}(\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{p}\ p \in \mathcal{M}_{\downarrow})$$

$$\Gamma \vdash !u(\vec{x},\vec{y}).P : \Delta \xrightarrow{u\langle\vec{v},\vec{V}\rangle} \Gamma \vdash P\{\vec{v}/\vec{x}\}\{\vec{V}/\vec{y}\} \mid !u(\vec{x},\vec{y}).P : \Delta \odot \vec{v}{:}\vec{\sigma} \qquad (\Gamma \vdash u{:}(\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{p}\ p \in \mathcal{M}_{!})$$

$$\Gamma \vdash \overline{u}\langle\vec{v},\vec{V}\rangle : \Delta \xrightarrow{\overline{u}\langle\vec{v},\vec{V}\rangle} \Gamma \vdash \mathbf{0} : \Delta/u\vec{v} \qquad (\Gamma \vdash x{:}(\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{p_{\mathrm{O}}})$$

$$\Gamma \vdash u(\vec{y}).P : \Delta \xrightarrow{u\langle\vec{V}\rangle} \Gamma \vdash P\{\vec{V}/\vec{y}\} : \Delta/u \odot \vec{v}{:}\vec{\sigma} \qquad (\Gamma \vdash u{:}(\exists[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{p}\ \Gamma \vdash V_j : \tau_j\{\vec{v}/\vec{x}\}\ p \in \mathcal{M}_{\downarrow})$$

$$\Gamma \vdash !u(\vec{y}).P : \Delta \xrightarrow{u\langle\vec{V}\rangle} \Gamma \vdash P\{\vec{V}/\vec{y}\} \mid !u(\vec{y}).P : \Delta \odot \vec{v}{:}\vec{\sigma} \qquad (\Gamma \vdash u{:}(\exists[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{p}\ \Gamma \vdash V_j : \tau_j\{\vec{v}/\vec{x}\}\ p \in \mathcal{M}_{!})$$

$$\Gamma \vdash \overline{u}\langle\vec{V}\rangle : \Delta \xrightarrow{\overline{u}\langle\vec{V}\rangle} \Gamma \vdash \mathbf{0} : \Delta/u\vec{v} \qquad (\Gamma \vdash u{:}(\exists[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{p_{\mathrm{O}}})$$

$$\frac{P_1' \equiv_\alpha P_1 \quad \Gamma \vdash P_1 : \Delta_1 \xrightarrow{l} \Gamma' \vdash P_2 : \Delta_2 \quad P_2 \equiv_\alpha P_2'}{\Gamma \vdash P_1' : \Delta_1 \xrightarrow{l} \Gamma' \vdash P_2' : \Delta_2}$$

$$\frac{\Gamma, a{:}\sigma \vdash P_1 : \Delta_1, a{:}\sigma \xrightarrow{l} \Gamma', a{:}\sigma \vdash P_2 : \Delta_2, a{:}\sigma}{\Gamma \vdash (\nu a{:}\sigma)P_1 : \Delta_1 \xrightarrow{l} \Gamma' \vdash (\nu a{:}\sigma)P_2 : \Delta_2} \quad a \notin \mathsf{n}(l)$$

$$\frac{\Gamma \vdash P_1 : \Delta_1 \xrightarrow{l} \Gamma' \vdash P_2 : \Delta_2}{\Gamma \vdash P_1|Q : \Delta_1 \odot \Delta \xrightarrow{l} \Gamma \vdash P_2|Q : \Delta_2 \odot \Delta} \quad \Delta_1 \odot \Delta \text{ allows } l$$

$$\frac{\Gamma_1 \vdash P_1 : \Delta_1 \xrightarrow{(\nu\vec{a})\overline{u}\langle\vec{V}\rangle} \Gamma_2 \vdash P_2 : \Delta_2 \quad \Gamma_1 \vdash Q_1 : \Delta_1' \xrightarrow{u\langle\vec{V}\rangle} \Gamma_2 \vdash Q_2 : \Delta_2'}{\Gamma_1 \vdash P_1|Q_1 : \Delta_1 \odot \Delta_1' \xrightarrow{\tau} \Gamma_2 \vdash (\nu\vec{a})(P_2|Q_2) : \Delta_2 \odot \Delta_2'/\vec{a}}$$

$$\frac{\Gamma_1, a{:}\sigma \vdash P_1 : \Delta_1, a{:}\sigma \xrightarrow{(\nu\vec{b})\overline{u}\langle\vec{v},\vec{V}\rangle} \Gamma_2, a{:}\sigma \vdash P_2 : \Delta_2, a{:}\sigma}{\Gamma_1 \vdash (\nu a{:}\sigma)P_1 : \Delta_1 \xrightarrow{(\nu\vec{b}a)\overline{u}\langle\vec{v},\vec{V}\rangle} \Gamma_2, a{:}\sigma \vdash P_2 : \Delta_2, a{:}\sigma} \quad \begin{array}{l}\Gamma \vdash u{:}(\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{p_{\mathrm{O}}} \\ a \in \{\vec{v}\} \setminus \{\vec{b},u\}\end{array}$$

$$\frac{\Gamma_1, a{:}\sigma \vdash P_1 : \Delta_1, a{:}\sigma \xrightarrow{(\nu\vec{b})\overline{u}\langle\vec{V}\rangle} \Gamma_2, a{:}\sigma \vdash P_2 : \Delta_2, a{:}\sigma}{\Gamma_1 \vdash (\nu a{:}\sigma)P_1 : \Delta_1 \xrightarrow{(\nu\vec{b}a)\overline{u}\langle\vec{V}\rangle} \Gamma_2, a{:}\sigma \vdash P_2 : \Delta_2, a{:}\sigma} \quad \begin{array}{l}\Gamma \vdash u{:}(\exists[\vec{x}{:}\vec{\sigma}]\vec{\tau})^{p_{\mathrm{O}}} \\ \Gamma \vdash \{\vec{v},\vec{V}\},\ a \in \{\vec{v}\} \setminus \{\vec{b},u\}\end{array}$$

We assume all l.h.s. processes are well-typed. We define the relation Δ *allows* l as the negation of: (1) $\Delta(\mathsf{sbj}(l)) = \updownarrow$ or (2) l is output and $\mathsf{md}(\Delta(\mathsf{sbj}(l))) \in \mathcal{M}_{!}$ [6]. $\mathsf{n}(l)/\mathsf{sbj}(l)$ is the set of names/free subject names in l. We omit rules for symmetric case of $|$.

FIGURE 17. Labelled Transition System

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] Amadio, R., Boudol, G. and Lhoussain, C., The receptive distributed π-calculus, *FSTTCS'99*, LNCS, Springer-Verlag, 1999.

(Well-Formedness)

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

(Subtyping Rules)

$$\text{(s-base)}\ \frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \leq \alpha} \qquad \text{(s-abs}_H\text{)}\ \frac{\Gamma \vdash \tau_{1i} \to \tau_{2i} : \mathtt{tp}_H \quad \Gamma \vdash \tau_{12} \leq \tau_{11} \quad \Gamma \vdash \tau_{21} \leq \tau_{22}}{\Gamma \vdash \tau_{11} \to \tau_{21} \leq \tau_{12} \to \tau_{22}} \qquad \text{(s-abs}_N\text{)}\ \frac{\Gamma \vdash \Pi(x{:}\sigma_i)\tau_i : \mathtt{tp}_H \quad \Gamma \vdash \sigma_1 \leq \sigma_2 \quad \Gamma, x{:}\sigma_2 \vdash \tau_1 \leq \tau_2}{\Gamma \vdash \Pi(x{:}\sigma_1)\tau_1 \leq \Pi(x{:}\sigma_2)\tau_2}$$

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

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

$$\text{(s-pro)}\ \frac{\mathsf{dom}(\Delta_1) = \mathsf{dom}(\Delta_2) \quad \Gamma \vdash ?\updownarrow\Delta \odot \Delta_2 : \mathtt{tp}_P \quad \Gamma \vdash \Delta_1(a) \leq \Delta_2(a)}{\Gamma \vdash \Delta_1 \leq \Delta_2 \odot \Delta} \qquad (\mathsf{Sub}_H)\ \frac{\Gamma \vdash P : \tau \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash P : \tau'} \qquad (\mathsf{Sub}_N)\ \frac{\Gamma \vdash u : \sigma \quad \Gamma \vdash \sigma' \leq \sigma}{\Gamma \vdash u : \sigma'}$$

(Typing Rules)

$$(\mathsf{Ref})\ \frac{\Gamma \vdash u{:}\,\mathsf{ref}\langle\sigma\rangle \quad \Gamma \vdash v{:}\,\overline{\sigma}}{\Gamma \vdash \mathsf{ref}\langle uv\rangle : \mathsf{ref}\langle\sigma\rangle, v{:}\,\overline{\sigma}} \qquad (\mathsf{Bra}^{\downarrow_{\mathrm{A}}})\ \frac{\Gamma \vdash u{:}\,[\&_{i\in I}\Pi[\vec{x}_i : \vec{\sigma}_i]\vec{\tau}_i]^{\downarrow_{\mathrm{A}}} \quad \Gamma, \vec{x}_i : \vec{\sigma}_i, \vec{y}_i : \vec{\tau}_i \vdash P_i : \vec{x}_i : \vec{\sigma}_i, \uparrow_{\mathrm{A}}?\Delta^{-u\vec{x}_i}}{\Gamma \vdash u[\&_{i\in I}(\vec{x}{:}\vec{\sigma}, \vec{y}{:}\vec{\tau}).P_i] : u{:}\,[\&_{i\in I}\Pi[\vec{x}_i : \vec{\sigma}_i]\vec{\tau}_i]^{\downarrow_{\mathrm{A}}}, \Delta}$$

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

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

For well-formedness, we assume the same constraint for branching/selection channel types for each summand as in Figure 13. Subtyping rule for the unary output is defined as (s-select) by setting $I_1 = I_2 = \{1\}$. Δ in (Rep)/($\mathsf{In}^{\downarrow_{\mathrm{L}}, !_{\mathrm{L}}, !_{\mathrm{A}}}$) is replaced by Δ^{-a}/Δ^{-x}. We delete (Pair) and (Weak). ($\mathsf{Bra}^{\downarrow_{\mathrm{L}}, !_{\mathrm{L}}, !_{\mathrm{A}}}$) are defined as ($\mathsf{In}^{\downarrow_{\mathrm{L}}, !_{\mathrm{L}}, !_{\mathrm{A}}}$) in Figure 13. Δ^r in ($\mathsf{In}^{!_{\mathrm{A}}}$) means it does not contain mutable types. The subtyping of $\Pi[\vec{x}{:}\vec{\sigma}]\vec{\tau}$ is defined as (s-abs$_N$).

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

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

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

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

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

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

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

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

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

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

$\langle \sigma_{\mathtt{I}}, \sigma_{\mathtt{O}} \rangle \sqcap \langle \sigma'_{\mathtt{I}}, \sigma'_{\mathtt{O}} \rangle \stackrel{\text{def}}{=} \langle \sigma_{\mathtt{I}} \sqcap \sigma_{\mathtt{I}}, \sigma_{\mathtt{O}} \sqcap \sigma'_{\mathtt{O}} \rangle$

$\langle \sigma_{\mathtt{I}}, \sigma_{\mathtt{O}} \rangle \sqcup \langle \sigma'_{\mathtt{I}}, \sigma'_{\mathtt{O}} \rangle \stackrel{\text{def}}{=} \langle \sigma_{\mathtt{I}} \sqcup \sigma'_{\mathtt{I}}, \sigma_{\mathtt{O}} \sqcup \sigma'_{\mathtt{O}} \rangle \qquad \sigma_{\mathtt{I}} \geq \sigma'_{\mathtt{O}}$ and $\sigma'_{\mathtt{I}} \geq \sigma_{\mathtt{O}}$ else undefined.

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

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

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

FIGURE 19. Partial Meet and Join Operators

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

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

[8] 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.

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

[10] Cardelli, L. and Gordon, A., Typed Mobile Ambients, *POPL'99*, pp.79–92, ACM Press, 1999.

[11] Clarke, M., Grumberg, O. and Peled, D., *Model Checking*, MIT, 1999.

[12] Castagna, G., Vitek, J. and Zappa, F., The seal calculus, 2003. Available from ftp://ftp.di.ens.fr/pub/castagna/.

[13] De Nicola, R., Ferrari, G. and Pugliese, R., Klaim: a Kernel Language for Agents Interaction and Mobility, IEEE Trans. on Software Engineering, Vol.24(5), 1998.

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

[15] Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., and Rémy, D., A Calculus for Mobile Agents, *CONCUR'96*, LNCS 1119, pp.406–421, Springer-Verlag, 1996.

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

[17] 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.

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

[19] Gay, S. and Hole, M., Types and Subtypes for Client-Server Interactions. *ETAPS'99*, LNCS, Springer, 1999.

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

[21] Hicks, M., Kakkar, P., Moore, J.T., Gunter, C.A. and Nettles, S. PLAN: A Packet Language for Active Networks, *ICFP'98*, ACM, 1998.

[22] Hennessy, M. and Riely, J., Resource Access Control in Systems of Mobile Agents, *Info.&.Comp.*, Vol 173(1), pp.82–120, February 25, 2002.

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

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

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

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

[27] 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/~yoshida`.

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

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

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

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

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

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

[34] Kobayashi, N., Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness, *Proc. of TCS2000*, LNCS 1872, pp.365-389, Springer, 2000.

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

[36] Lhoussain, C. and Sasson, V., A dependently typed ambient calculus, submitted for a publication, pp.12, 2003.

[37] Manna, Z. and Pnueli, A., The Temporal Logic of Reactive and Concurrent Systems – Specification: Springer-Verlag, 1992.

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

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

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

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

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

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

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

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

[46] Rudys, A. et al., Termination in Language-based Systems, Network and Distributed System Security Symposium, 2001.

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

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

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

[50] Sewell, P., Global/Local Subtyping and Capability Inference for a Distributed π-calculus, *ICALP'98*, LNCS 1443, pp.695–706, Springer-Verlag, 1998.

[51] 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.

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

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

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

[55] Vasconcelos, V., A Note on a Typing System for the Higher-Order π-Calculus, Keio Technical Report, 1993.

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

[57] Vivas, J.-L. and Yoshida, N., Dynamic Channel Screening in the Higher Order Pi-Calculus, Proceeding of F-WAN, Foundations of Wide Area Network Computing , Electronic Notes in Theoretical Computer Science, Elsevier Science, August 2002.

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

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

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

[61] 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.

[62] 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.

[63] 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. To apper in *Info. &. Comp.*. Available at www.dos.ic.ac.uk/˜yoshida.

[64] 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.

[65] 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*.

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

[67] Yoshida, N., Channel Dependent Types for Higher-Order Mobile Processes (extended abstract with appendix), Available at `www.doc.ic.ac.uk/˜yoshida/`.

[68] Part II of this paper: Secrecy and Access Control.

[69] Part III of this paper: Type-Based Security for a Language with Code Distribution.