Proc. of FST/TCS'16,
LNCS 1180, pp. 371–386,
Springer-Verlag, 1996.

GRAPH TYPES FOR MONADIC MOBILE PROCESSES

NOBUKO YOSHIDA

Abstract. While types for name passing calculi have been studied extensively in the context of sorting of polyadic π-calculus [5, 34, 9, 28, 32, 19, 33, 10, 17], the same type abstraction is not possible in the monadic setting, which was left as an open issue by Milner [21]. We solve this problem with an extension of sorting which captures dynamic aspects of process behaviour in a simple way. Equationally this results in the full abstraction of the standard encoding of polyadic π-calculus into the monadic one: the sorted polyadic π-terms are equated by a basic behavioural equality in the polyadic calculus if and only if their encodings are equated in a basic behavioural equality in the typed monadic calculus. This is the first result of this kind we know of in the context of the encoding of polyadic name passing, which is a typical example of translation of high-level communication structures into π-calculus. The construction is general enough to be extendable to encodings of calculi with more complex operational structures.

1. Introduction

The monadic π-calculus [22, 20] is a powerful formalism in which we can construct complex structures of concurrent computing by combining simple monadic name passing. The construction of significant computational structures in this calculus is done by passing and using private names between interacting parties to control the sharing of interaction points. For example, the following process expresses communication of a sequence of names (below $ax.P$ is input and $\overline{a}v.P$ is output, $(a)P$ denotes scope restriction, and c, z are fresh).

$$az.zx_1.zx_2.zx_3.P \mid (c)\ \overline{a}c.\overline{c}v_1.\overline{c}v_2.\overline{c}v_3.Q \longrightarrow\!\!\!\!\!\rightarrow\ P\{v_1v_2v_3/x_1x_2x_3\} \mid Q \qquad (1.1)$$

In this example, coming from [22], the private channel c is used during interaction, so that, after the initial step, the communication of v_1, v_2 and v_3 is done deterministically without interference from the third party. This example also shows that we can represent polyadic (multiple) name passing from monadic name passing. Another example with a more complex communication structure follows.

$$az.zx_1.\overline{z}v_1.zx_2.\overline{z}v_2.P \mid (c)\ \overline{a}c.\overline{c}w_1.cy_1.\overline{c}w_2.cy_2.Q \longrightarrow\!\!\!\!\!\rightarrow\ P\{w_1w_2/x_1x_2\} \mid Q\{v_1v_2/y_1y_2\} \qquad (1.2)$$

Note input and output are mixed together. As in (1.1), once two parties start interaction, a sequence of communications is done following a prescribed protocol using a private channel. The same scheme can be used to realise more complex communication structures, including those with parallelism and intricate information flow.

As a means to study rich computational structures representable in name passing processes, a notion of types called *sorting* was introduced by Milner [21] and has been studied extensively since then [5, 34, 9, 28, 32, 19, 33, 10, 17]. Sorting shows how a name carries another name. For example, if v has a type, say, nat, and we have a term $\overline{a}v.\mathbf{0}$,

Department of Computer Science, University of Edinburgh, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, UK. e-mail: ny@dcs.ed.ac.uk. Partially supported by JSPS Fellowship for Japanese Junior Scientists and Ogata-jyosei grant of Keio University.

then a should have a type (nat), a type which carries nat. This idea and its ramifications have been used to analyse significant semantic properties of calculi with polyadic name passing. However, as was already noticed in [21], sorting in monadic π-calculus is not powerful enough to type-abstract known encodings of polyadic name passing. Indeed, (1.1) becomes ill-sorted if v_1, v_2, v_3 have different sorts. As far as we know, the situation remains the same with the refinements of sorting proposed in [28, 17, 10]. This means, among other things, sorting restricted to monadic terms does not give a rich semantic analysis as in the polyadic case, while behaviourally the above encoding does mimic polyadic name passing. In general, this is because sorting does not capture the dynamic structure of interaction, especially those using the transmission of private names as in (1.1) and (1.2), which is omnipresent even in the polyadic setting.

In this paper we develop a syntactic theory of types for monadic π-calculus which extends sorting in a simple way and by which we can extract the abstract operational structures of π-terms including those of (1.1) and (1.2) as well as more complex ones. The key technical idea is to represent a class of dynamic communication behaviours using simple combinatorial expressions, i.e. graphs. In a graph type, nodes denote atomic actions and edges denote the activation ordering between them, representing as a whole a deterministic protocol a process will be engaged in. This departure from the type abstraction of static usage of names to that of dynamic process behaviour makes it possible to type-abstract many non-trivial computational structures representable in π-calculus. Indeed it allows us to solve Milner's open issue mentioned above in a sharper form: We show that not only the encoding of sorted polyadic π-terms into the monadic terms is typable preserving the original type structure, but also it results in *equational full abstraction*: let $=_{\mathrm{P}}$ and $=_\pi$ be suitably defined behavioural equalities over sorted polyadic π-terms and typed monadic π-terms, respectively. Then we get:

$$P =_{\mathrm{P}} Q \quad \Leftrightarrow \quad [\![P]\!] =_\pi [\![Q]\!]$$

where $[\![\cdot]\!]$ is the standard encoding from polyadic π-terms to monadic π-terms. In the untyped setting, we can only get "$\Leftarrow$" (adequacy) direction in terms of usual weak behavioural equivalences. This is due to possible violation of "protocols" by environment processes, and suggests the lack of precise type abstraction of the communication structure of various encodings such as (1.1) and (1.2) in preceding type disciplines, even though the effect of process types on behavioural equalities has been known since the work of Pierce and Sangiorgi [28, 17]. In this context, our result, which seems the first of this kind, shows how precisely our type discipline captures essential behavioural information these protocols carry, while using a simple idea of graph-based types and a small typing system. Moreover the results can be easily extended to calculi with more complex communication structures, cf. 6.2 later. Because polyadic name passing is a typical example of encoding of high-level communication structures into π-calculus, cf. [22, 20, 16], the presented construction and its extensions will hopefully become a basis for using typed equality over π-terms to verify the semantic equality of various target languages and calculi in a uniform framework.

Some preliminaries: throughout the paper, we fix a countable set $\mathcal{N}$ of *names*, ranged over by $a, b, c, ..$ or $x, y, z, ...$ The syntax of monadic π-calculus follows that of [20] (writing $ax.P$ for $a(x).P$ in [20]):

$$P \quad ::= \quad ax.P \;\mid\; \overline{a}x.P \;\mid\; P \,|\, Q \;\mid\; (a)P \;\mid\; !P \;\mid \mathbf{0}$$

The definitions for free and bound names in P are standard and denoted by $\mathcal{FN}(P)$ and $\mathcal{BN}(P)$. The structural congruence $\equiv$, the reduction relation $\longrightarrow$ and the early transition relation $\stackrel{l}{\longrightarrow}$ again follow the standard definitions [22, 20]. For the space sake,

detailed definitions, technical lemmas and all proofs of the results in the paper are left to the full version [36].

The structure of the rest of the paper: Section 2 introduces the basic notion of graph types. Section 3 gives the typing system and presents the subject reduction. Section 4 clarifies relationship between types and the behaviour of typed terms. Section 5 proves the full abstraction of the standard encoding of polyadic π-calculus into the monadic one, together with preservation of type structure. Section 6 concludes with comparison with related work and discussions on further issues.

2. Graph Types

2.1. Given a finite poset $\langle X, \leq \rangle$, its *covering relation* $\rightarrow$ is defined: $x \rightarrow y$ iff (1) $x \lneq y$ and (2) for no z, $x \lneq z \lneq y$ where $x, y, z \in X$ (cf. [3], Chap.1). Then $\langle X, \rightarrow \rangle$ gives an acyclic directed simple graph (equivalent to the familiar Hasse diagram), from which $\leq$ is recovered as $\rightarrow^*$ (the reflexive transitive closure) and $\lneq$ as $\rightarrow^+$ (the transitive closure). In this paper the term "graph" always means this kind of a directed graph, presented by the covering relation denoted $\rightarrow$. We write $G, G', \ldots$ for such graphs, $N(G)$ for the sets of nodes of G, and $G_1 \uplus G_2$ for the (disjoint) graph union of G_1 and G_2.

2.2. **Definition.** (graph types) A *graph type* is a graph in the above sense whose nodes are occurrences of the following *atomic types*[1]:

$$\mathbf{n} \quad ::= \quad a{\downarrow}(b) \mid a{\uparrow}(b) \mid a{\downarrow}\mathbf{x} \mid a{\uparrow}\mathbf{x}$$

where $a, b, c, \ldots$ range over $\mathcal{N}$ (cf. Sec.1) and $\mathbf{x}, \mathbf{x}', \ldots$ range over the set $\mathcal{V}$ of *base types*. In $a \updownarrow (b)$ or $a \updownarrow \mathbf{x}$, a is the *subject* and $b, \mathbf{x}$ are *objects*. Over atomic types we define a self-inverse function $\overline{(\cdot)}$ such that: $\overline{a{\downarrow}(b)} = a{\uparrow}(b)$ and $\overline{a{\downarrow}\mathbf{x}} = a{\uparrow}\mathbf{x}$, which is extended to graph types so that $\overline{G}$ (the *dual* of G) is the result of replacing all occurrences of atomic types in G by their duals. *Heads* of a graph type are those occurrences of atomic types which are minimal w.r.t. the order $\rightarrow^*$. We often use atomic types to denote their occurrences in a graph type, writing, for example, $\mathbf{n}_1 \rightarrow \mathbf{n}_2$, if no confusion arises.

Intuitively, nodes in a graph type denote atomic actions of a process (best understood as abstraction of labels in a transition relation), while edges denote their synchronisation ordering. If a graph type represents the structure of interaction from one party's viewpoint, then its dual represents the same interaction from another party's viewpoint.

2.3. **Binding.** In atomic types, the only bound occurrence of a name is b in $a \updownarrow (b)$, others occurring free. $\mathsf{bn}(\mathbf{n})$ and $\mathsf{fn}(\mathbf{n})$ are the sets of a bound/free name in $\mathbf{n}$, respectively. In a graph type, $\mathbf{n}$ binds the set of nodes $\{\mathbf{n}' \mid \mathbf{n} \rightarrow^+ \mathbf{n}' \wedge \mathsf{bn}(\mathbf{n}) = \mathsf{fn}(\mathbf{n}')\}$. We write $\mathbf{n} \twoheadrightarrow \mathbf{n}'$, if $\mathbf{n}$ binds $\mathbf{n}'$. A graph type is *normal* if, in any of its bindings, no two binders bind the same name occurrence (see Example 2.4). *Hereafter a "graph type" always means a normal graph type.* For (normal) graph types, the standard idea of the sets of free/bound names and the α-equality applies, which we denote $\mathsf{bn}(G)$, $\mathsf{fn}(G)$ and $\equiv_\alpha$, respectively. We always assume that a graph type obeys the usual *binding condition*, i.e. no two binding occurrences are of the same name and $\mathsf{bn}(G)$ and $\mathsf{fn}(G)$ are disjoint.

[1] Equivalently nodes of a graph type are labeled by atomic types. Graph types are considered modulo the graph isomorphism preserving these labels and the direction of edges.

2.4. **Example.** Some simple graph types can be written syntactically, for example, $b\downarrow\mathbf{x} \to a\uparrow\mathbf{x}'$ and $b\uparrow(c) \to a\uparrow\mathbf{x}$. Further examples are given in Figure 1. Notice (2) and (3) of Figure 1 are normal while (1) is not (so (1) is not considered as a graph type). We can also see (2) and (3) are α-convertible to each other and both obey the binding condition.

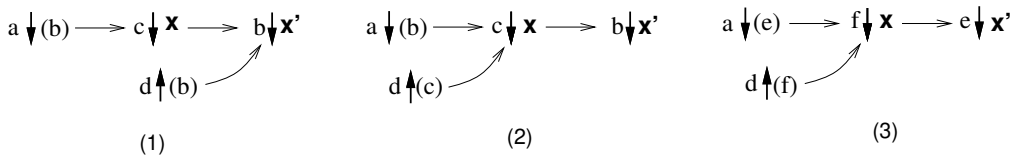

FIGURE 1. Graph Types

The following two kinds of graph types are important for our theory.

2.5. **Definition.** (safe and pointed graph types)

(i) (safety) A graph type G is *safe*, if, for any distinct $\mathbf{n}_1, \mathbf{n}_2 \in N(G)$, $\mathsf{fn}(\mathbf{n}_1) = \mathsf{fn}(\mathbf{n}_2)$ implies either $\mathbf{n}_1 \to^+ \mathbf{n}_2$ or $\mathbf{n}_2 \to^+ \mathbf{n}_1$.

(ii) (pointedness) A graph type G is *pointed*, if: (1) There is a unique head in G. (2) All name occurrences except the subject of the head are bound, and (3) G is safe. We write $G_{\langle a \rangle}$ if G is pointed with a free.

Notice G is safe (resp. pointed) iff $\overline{G}$ is safe (resp. pointed). In a safe graph type, two nodes with the same subject are strictly ordered so that, while the same subject may be used several times in interaction, at any one time, only one input and one output are active, making interaction deterministic. In Figure 2, (1) is not safe because there is no ordering between $c\uparrow\mathbf{x}$ and $c\uparrow(e)$.

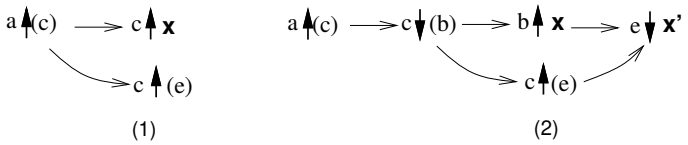

FIGURE 2. Unsafe and Safe Types

The pointedness says that an interaction starts at a single interacting point, and that, after the initial interaction, the communications are always done using private channels, so that they are free from the interference of the third party. An example of a pointed (and hence also safe) graph type is shown in (2) of Figure 2.

2.6. **Composition Function.** We next discuss composition of graph types. Write $\mathsf{dom}(\Psi)$ for the domain of a partial function Ψ. Given G and G', assume $\mathbf{n}, \mathbf{n}_i \in N(G)$ and $\mathbf{n}', \mathbf{n}'_i \in N(G')$. Then a *composition function* Ψ from G to G' is a partial injection $\Psi : N(G) \rightharpoonup N(G')$ such that:

(duality) If $\mathbf{n} \in \mathsf{dom}(\Psi)$, then $\Psi(\mathbf{n}) = \overline{\mathbf{n}}$.

(acyclicity) If $\mathbf{n}_1, \mathbf{n}_2 \in \mathsf{dom}(\Psi)$, both $\mathbf{n}_1 \to^+ \mathbf{n}_2$ and $\Psi(\mathbf{n}_2) \to^+ \Psi(\mathbf{n}_1)$ cannot hold.

(covering) (1) if $a \in \mathsf{fn}(G) \cap \mathsf{fn}(G')$, then $a \in \mathsf{fn}(\mathbf{n})$ implies $\mathbf{n} \in \mathsf{dom}(\Psi)$ (resp. $a \in \mathsf{fn}(\mathbf{n}')$ implies $\mathbf{n}' \in \mathsf{dom}(\Psi^{-1})$). (2) if there is any binding chain $\mathbf{n}_1 \succcurlyeq \mathbf{n}_2 \succcurlyeq \cdots \succcurlyeq \mathbf{n}_n$ in G and $\mathbf{n}_i \in \mathsf{dom}(\Psi)$ for some $i \leq n$, then for all $1 \leq j \leq n$, $\mathbf{n}_j \in \mathsf{dom}(\Psi)$, and then $\Psi(\mathbf{n}_1) \succcurlyeq \Psi(\mathbf{n}_2) \succcurlyeq \cdots \succcurlyeq \Psi(\mathbf{n}_n)$. Similarly for any binding chain in G' and Ψ^{-1}.

Observe that, when G and G' are safe, there can be at most one composition function from G to G'. We write $G_1 \asymp G_2$ if, for some $G_1' \equiv_\alpha G_1$ and $G_2' \equiv_\alpha G_2$, there is a composition function from G_1' to G_2'. For convenience we assume, when we write $G_1 \asymp G_2$, they are already suitably α-converted so that the composition function relates their nodes directly. We now introduce two operations on graph types.

2.7. **Operations on Graph Types.**

(i) (prefix) "$\mathbf{n} \to G$" denotes the result of adding a new node (occurrence) of $\mathbf{n}$ to G together with edges from the new node to all heads of G.

(ii) (parallel composition) Given safe graph types G_1 and G_2 s.t. $G_1 \asymp G_2$ and the corresponding composition function Ψ, the *composition* of G_1 and G_2, denoted by $G_1 \odot G_2$, is a graph type obtained by: first identifying Ψ-related nodes of $G_1 \uplus G_2$, getting a graph, say, G', and then by eliminating every identified node, preserving and reflecting $\to^*$ of G', deleting all redundant edges if any.

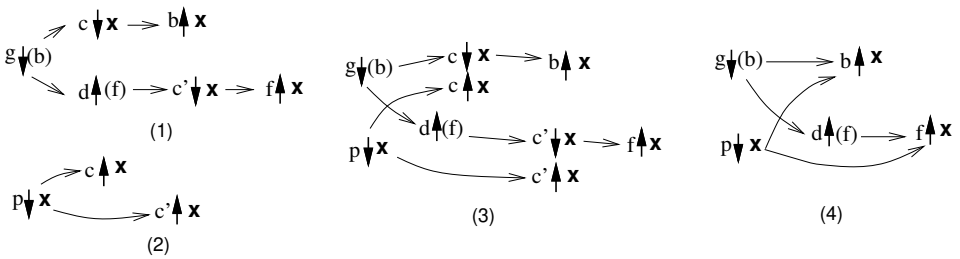

FIGURE 3. Cut Elimination

Figure 3 shows a simple example of parallel composition. Given two safe graphs in (1) and (2), first we identify the same nodes as in $\langle c\downarrow\mathbf{x},\ c\uparrow\mathbf{x}\rangle$ and $\langle c'\downarrow\mathbf{x},\ c'\uparrow\mathbf{x}\rangle$ in (3), next eliminate one by one (the order does not matter). Then we have the final graph (4). In this way the original dependency relations are merged to generate a more complex one.

We can easily show if G is safe then $\mathbf{n} \to G$ is also (normal and) safe, similarly if G_1 and G_2 are safe and $G_1 \asymp G_2$, then $G_1 \odot G_2$ is again (normal and) safe. Another notable fact is that any graph type can be constructed from graph types with a unique node by repeatedly applying these two operations.

3. The Typing System and Its Basic Properties

3.1. **General Idea.** This section introduces a typing system based on graph-types for monadic π-terms and establishes its basic syntactic properties, including the subject reduction property. For simplicity of presentation we do not treat recursive types, whose incorporation is straightforward, cf. 6.2. The typing judgment for a term P has a form:

$$\Gamma \vdash P \triangleright_A G$$

where Γ is an *environment* assigning protocol types introduced in 3.2 to some of free names in P, P is a π-term, A is a *hidden name set*, and G is a graph type. Roughly this judgment tells us: "P has a linear behaviour specified in G, and otherwise obeys the protocols specified in Γ." The separation between linear behaviour and "shared"

interaction is essential for tractability of type inference. A is an auxiliary name set to record the linear usage of names. We start from introducing the idea of *protocol types*, which are essentially pairs of pointed types, and which generalise the notion of sorting.

3.2. **Protocol Types.** Write "$\lambda a.G_{\langle a \rangle}$" for the *name abstraction* of a in a pointed graph $G_{\langle a \rangle}$, with the usual notion of binding (We only need this restricted abstraction in this paper[2]). Then a *protocol type* of $\lambda a.G_{\langle a \rangle}$ is the set $\{\lambda a.G_{\langle a \rangle},\ \lambda a.\overline{G}_{\langle a \rangle}\}$.

Using this notion, we fix the set of types we shall use in our typing system. Given any base $\mathcal{V}'$, we get the set of atomic types, from which we get the set of graph types, and then that of protocol types. The corresponding set functions are written $\psi_A(\mathcal{V}')$, $\psi_G(\mathcal{V}')$ and $\psi_P(\mathcal{V}')$, respectively. Then we define, given an initial base $\mathcal{V}$ (say $\{\mathsf{nat}\}$), $\mathbf{A}_\mathcal{V}$, $\mathbf{G}_\mathcal{V}$ and $\mathbf{P}_\mathcal{V}$ as the smallest sets closed under

$$\mathbf{P}_\mathcal{V} \ \supseteq\ \mathcal{V} \cup \psi_P(\mathbf{P}_\mathcal{V}), \quad \mathbf{A}_\mathcal{V} \ \supseteq\ \psi_A(\mathbf{P}_\mathcal{V}) \quad \text{and} \quad \mathbf{G}_\mathcal{V} \ \supseteq\ \psi_G(\mathbf{P}_\mathcal{V}).$$

$\alpha, \beta, \ldots$ range over $\mathbf{P}_\mathcal{V}$. We shall use these sets for our typing system. Some notations:

- "$\updownarrow(x) \to G$" denotes abstraction $\lambda a.(a\updownarrow(x) \to G)$.
- For $\alpha \in \mathbf{P}_\mathcal{V}$, we write (α) for $\{\lambda a.a\downarrow\alpha,\ \lambda a.a\uparrow\alpha\}$.

The notation "(α)" comes from the monadic sorting [21], and it indeed functions as such (representing one-time usage of name passing). Protocol types in general, however, carry more dynamic information, by which we can specify refined communication behaviour a process may be engaged in. A protocol type (similar to a pointed graph (2) in Figure 2) is given in Figure 4.

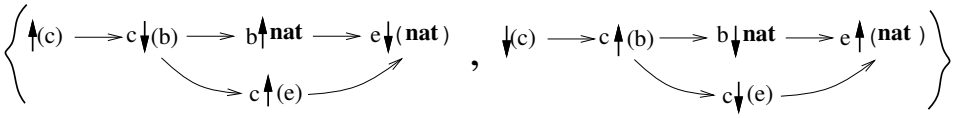

FIGURE 4. Protocol Type

3.3. **Typing Function.** A *typing function* is a function from a finite subset of $\mathcal{N}$ to a set of protocol types. The set of typing functions are ranged over by $\Gamma, \Delta, \ldots$. We often regard Γ as a finite set of elements of form $a : \alpha$. $\mathsf{fn}(\Gamma)$ denotes $\{a \mid a : \alpha \in \Gamma\}$. Let $A, A', \ldots$ denote a finite subset of $\mathcal{N}$. "$\Gamma \setminus A$" denotes $\{a : \alpha \in \Gamma \mid a \notin A\}$, while "$\Gamma \lceil A$" denotes $\{a : \alpha \in \Gamma \mid a \in A\}$. "$a : \alpha, \Gamma$" means $\Gamma \cup \{a : \alpha\}$ together with the assumption $a \notin \mathsf{fn}(\Gamma)$. "$\Gamma \asymp \Delta$" represents, if $x : \alpha \in \Gamma$ and $x : \beta \in \Delta$, we have $\alpha = \beta$. If we get Γ by α-conversion of some protocol type in Δ, then we write $\Gamma \equiv_\alpha \Delta$.

3.4. **Sequent and Typing System.** The sequent has a form $\Gamma \vdash P \triangleright_A G$, which we read: "$P$ has a type G under Γ with hidden names A". Γ is called the *basis* of the sequent, P is its *subject*, A is the *hidden name set*, and G is its *type*. We write $\Gamma \vdash P \triangleright G$ if $A = \emptyset$ (read "P has a type G under Γ"), $\Gamma \vdash P \triangleright_A$ if $G = \emptyset$, and $\Gamma \vdash P$ if both are empty. The typing system, denoted π_{G}, is given in Figure 5, where "$\langle G_1, A_1\rangle \asymp \langle G_2, A_2\rangle$" denotes both $G_1 \asymp G_2$ and $(\mathsf{fn}(G_i) \cup A_i) \cap A_j = \emptyset$ for $i \neq j$. We hereafter write $\Gamma \vdash P \triangleright_A G$ if the sequent is derivable in π_{G}.

[2]It is possible to consider extensions such as a general name abstraction of pointed types with two or more heads (cf. [4]), and with a non-dual pairs (cf. [19]), though we do not treat them here.

(I) Prefix Rules.

$$(\mathrm{in}_1)\ \frac{\Gamma \vdash P \triangleright_A G}{\Gamma \vdash ax.P \triangleright_A a{\downarrow}(x) \to G}\ (1,2) \qquad (\mathrm{out}_1)\ \frac{\Gamma \vdash P \triangleright_A G}{\Gamma \vdash \overline{a}x.P \triangleright_{A\cup\{x\}} a{\uparrow}(x) \to G}\ (1,2)$$

$$(\mathrm{in}_2)\ \frac{x:\alpha, \Gamma \vdash P \triangleright_A G}{\Gamma \vdash ax.P \triangleright_A a{\downarrow}\alpha \to G}\ (1) \qquad (\mathrm{out}_2)\ \frac{x:\alpha, \Gamma \vdash P \triangleright_A G}{x:\alpha, \Gamma \vdash \overline{a}x.P \triangleright_A a{\uparrow}\alpha \to G}\ (1)$$

$$(\mathrm{in}_3)\ \frac{a:\beta, \Gamma \vdash P \triangleright_A G}{a:\beta, \Gamma \vdash ax.P \triangleright_A}\ (2,3) \qquad (\mathrm{out}_3)\ \frac{a:\beta,\ \Gamma \vdash P \triangleright_A \overline{G}}{a:\beta,\ \Gamma \vdash \overline{a}x.P \triangleright_{A\cup\{x\}}}\ (2,3)$$

$$(\mathrm{in}_4)\ \frac{a:(\alpha),\ x:\alpha,\ \Gamma \vdash P \triangleright_A}{a:(\alpha),\ \Gamma \vdash ax.P \triangleright_A} \qquad (\mathrm{out}_4)\ \frac{a:(\alpha),\ x:\alpha, \Gamma \vdash P \triangleright_A}{a:(\alpha),\ x:\alpha, \Gamma \vdash \overline{a}x.P \triangleright_A}$$

(II) Other Rules.

$$(\mathrm{par})\ \frac{\Gamma \vdash P \triangleright_{A_1} G_1 \quad \Gamma \vdash Q \triangleright_{A_2} G_2}{\Gamma \vdash P \mid Q \triangleright_{A_1\cup A_2\cup A} G_1 \odot G_2}\ (4) \qquad (\mathrm{res})\ \frac{\Gamma \vdash P \triangleright_A G}{\Gamma\backslash a \vdash (a)P \triangleright_{A\backslash\{a\}} G}\ (5)$$

$$(\mathrm{alpha})\ \frac{\Gamma \vdash P \triangleright_A G \quad G \equiv_\alpha G' \quad \Gamma \equiv_\alpha \Gamma'}{\Gamma' \vdash P \triangleright_A G'} \qquad (\mathrm{weak})\ \frac{\Gamma \vdash P \triangleright_A G}{a:\alpha,\ \Gamma \vdash P \triangleright_{A\cup\{b\}} G}\ (6)$$

$$(\mathrm{nil})\ \vdash \mathbf{0} \qquad (\mathrm{rep})\ \frac{\Gamma \vdash P}{\Gamma \vdash !P}$$

(1) $a \notin \mathsf{fn}(\Gamma) \cup A$. (2) $x \notin \mathsf{fn}(\Gamma) \cup A$. (3) $\beta = \{{\downarrow}(x) \to G, {\uparrow}(x) \to \overline{G}\}$. (4) $\langle G_1, A_1\rangle \asymp \langle G_2, A_2\rangle$, $A = \mathsf{fn}(G_1) \cap \mathsf{fn}(G_2)$. (5) $a \notin \mathsf{fn}(G)$. (6) $a \notin A \cup \mathsf{fn}(G)$, $b \notin \mathsf{fn}(\Gamma) \cup \mathsf{fn}(G)$.

FIGURE 5. Typing System π_{G}

3.5. **Comments on Typing Rules.** First notice there are four pairs of (in) and (out) rules corresponding to what kinds of atomic types will be introduced.

($\mathbf{in}_1$, $\mathbf{out}_1$): In (in_1), "$a{\downarrow}(x) \to G$" represents the behaviour of a process of getting a fresh name x and then acting with behaviour G (corresponding to $ax.P \xrightarrow{a(x)} P$ in the late transition [22]). Similarly (out_1) represents a "bound output" $(x)\overline{a}x.P \xrightarrow{\overline{a}(x)} P$. In the latter x is recorded as a hidden name to be later eliminated by (res) rule.

($\mathbf{in}_2$, $\mathbf{out}_2$): These rules specify the situation where a process gets/emits a name of a protocol type α, and then acts like G.

($\mathbf{in}_3$, $\mathbf{out}_3$): If we have successfully constructed a protocol type β by finally putting $a \updownarrow (x)$ to G, then we may remove the whole pointed graph type to the shared environment by these rules. In (out_3), we memorize "x" as a hidden name. Notice that, by using (in_1) and (out_1) instead of (in_3) and (out_3), we can keep $a \updownarrow (x) \to G$ in the linear realm to construct a more complex protocol type. Notice also this shows the usual notion of principal type scheme does not exist in π_{G}.

($\mathbf{in}_4$, $\mathbf{out}_4$): These rules correspond to monadic sorting, so that (together with (par) and other rules) this system subsumes monadic sorting. By ($\mathrm{in}_{3,4}$) and ($\mathrm{out}_{3,4}$), we make it possible to infer types of a term which contains a shared name at which multiple agents with consistent protocols may interact.

(par): This is the key rule which allows us to extract versatile forms of interaction structures from π-terms. Notice that, even if G_1 and G_2 are sequential (i.e. $\to^*$ is a total order), we may get a non-sequential graph (but still representing a linear behaviour, see 4.5). In the rule we memorise the set of names lost by cut-elimination: this prevents further connection to these names so that reduction at these points is

deterministic (cf. 4.3). Finally it is notable that, if we only allow composition of disjoint graphs, we lose the subject reduction property we establish later: thus the present rule not only extends the typability of terms but also is essential to the fundamental property as a typing system.

(res): The combination of this rule and other rules that manipulate the hidden name set such as ($\mathrm{out}_{1,3}$) and (par) enables us to record the linear usage of names during the type inference before the actual occurrence of the scope restriction.

Other rules are standard. By (alpha), we hereafter safely consider graph types and basis modulo their respective α-equality.

3.6. **Example.** Below we give a simple example of typing. Let $P_1 \stackrel{\mathrm{def}}{=} z_1 y.cx.\overline{z_1}c$, $P_2 \stackrel{\mathrm{def}}{=} z_2 x.\overline{c}w$ and $P \stackrel{\mathrm{def}}{=} az_1.z_1 z_2.(c)\ (P_1 \mid P_2)$. Then we have:

$$w : \mathsf{nat} \vdash P_1 \triangleright z_1\!\downarrow\!\mathsf{nat} \to c\!\downarrow\!\mathsf{nat} \to z_1\!\uparrow\!(\mathsf{nat}) \quad \text{and} \quad w : \mathsf{nat} \vdash P_2 \triangleright z_2\!\downarrow\!\mathsf{nat} \to c\!\uparrow\!\mathsf{nat}$$

Note z_1 carries two different types. By (par) rule a proper graph structure arises.

$$w : \mathsf{nat} \vdash (P_1 \mid P_2) \ \triangleright_{\{c\}} \ \begin{array}{l} z_1\!\downarrow\!\mathsf{nat} \to z_1\!\uparrow\!(\mathsf{nat}). \\ z_2\!\downarrow\!\mathsf{nat} \nearrow \end{array}$$

After restricting and prefixing, we have the final state.

$$v : \mathsf{nat}, w : \mathsf{nat} \vdash P \ \triangleright \ \begin{array}{l} a\!\downarrow\!(z_1) \to z_1\!\downarrow\!(z_2) \to z_1\!\downarrow\!\mathsf{nat} \to z_1\!\uparrow\!(\mathsf{nat}). \\ \qquad\qquad\qquad\qquad \searrow z_2\!\downarrow\!\mathsf{nat} \nearrow \end{array}$$

Note that the final graph type is a-pointed, so that we could have moved them using (in_3) to the l.h.s. as protocol types.

The basic properties of the typing system follow. Notice, by (i,ii) below, we safely assume binding occurrences in types and a term of $\Gamma \vdash P \triangleright_A G$ are all distinct and disjoint from free occurrences.

3.7. **Lemma.**

(i) (properties of names) *Suppose* $\Gamma \vdash P \triangleright_A G$. *Then:* $\mathsf{fn}(\Gamma) \cap \mathsf{fn}(G) = A \cap \mathsf{fn}(G) = A \cap \mathsf{fn}(\Gamma) = \emptyset$, $\mathsf{fn}(P) \subset \mathsf{fn}(G) \cup A \cup \mathsf{fn}(\Gamma)$, $\mathsf{fn}(G) \subset \mathsf{fn}(P)$, *and* $\Gamma\lceil \mathsf{fn}(P) \vdash P \triangleright_{A \cap \mathsf{fn}(P)} G$.

(ii) (renaming) *Suppose* $\Gamma \vdash P \triangleright_A G$ *and* $b \notin \mathsf{fn}(\Gamma) \cup A \cup \mathsf{fn}(G)$. *Then we have* $\Gamma\{b/a\} \vdash P\{b/a\} \triangleright_{A\{b/a\}} G\{b/a\}$.

(iii) ($\equiv$) *If* $\Gamma \vdash P \triangleright_A G$ *and* $P \equiv Q$, *then* $\Gamma \vdash Q \triangleright_A G$.

(iv) (substitution) *If* $a : \beta, b : \beta, \Gamma \vdash P \triangleright_A G$, *then* $b : \beta, \Gamma \vdash P\{b/a\} \triangleright_A G$.

Using Lemma above, we can establish the subject reduction theorem, the key property of the typing system.

3.8. **Theorem.** (Subject Reduction).

If $\Gamma \vdash P \triangleright_A G$ *and* $P \longrightarrow\!\!\!\!\!\rightarrow P'$, *then we have* $\Gamma \vdash P' \triangleright_A G$.

It is notable that not only the environment Γ (which corresponds to sorting) but also the graph type of a term is invariant under reduction. This is because all possible reduction concerning the (names of the) graph type is already consumed when (par) rule is applied: and reduction concerning the (names of the) basis only creates dual graphs which eliminate each other to leave the graph type unchanged.

4. Behavioural Equality over Typed π-terms

This section introduces the basic constructions of typed behavioural equality over monadic π-terms following the untyped construction in [15]. Using this equality we clarify relationship between types and process behaviour they guarantee.

4.1. **Typed Terms and Relation.** A typed term is a tuple $\langle \Gamma,\ P,\ A,\ G\rangle$ such that $\Gamma \vdash P \triangleright_A G$. We simply write $\Gamma \vdash P \triangleright_A G$ for a typed term $\langle \Gamma,\ P,\ A,\ G\rangle$. A relation over typed terms, ranged over $\mathcal{R}, \mathcal{R}'...$, is called a *typed relation*, if:

(a) $\Gamma \vdash P \triangleright_A G\ \mathcal{R}\ \Delta \vdash Q \triangleright_B G' \ \Rightarrow\ A = B,\ G = G'$, and $\Gamma = \Delta$, and
(b) $\Gamma \vdash P \triangleright_A G\ \mathcal{R}\Gamma \vdash Q \triangleright_A G \ \Rightarrow\ (a\colon \alpha,\ \Gamma \vdash P \triangleright_{A\cup\{b\}} G)\ \mathcal{R}\ (a\colon \alpha,\ \Gamma \vdash P \triangleright_{A\cup\{b\}} G)$ with (6) of Fig.5.

We often write $\Gamma \vdash P\ \mathcal{R}\ Q \triangleright_A G$ for a tuple of a typed relation, or just $P\ \mathcal{R}\ Q$ if the type assignment is obvious from the context.

Notice $\equiv$ and $\longrightarrow$ restricted to typed terms are typed relations (by Lemma 3.7 (iii) and Theorem 3.8 respectively). Several important classes of typed relations are:

(i) A typed relation is *reflexive* if it contains the structural equality over typed π-terms. The *transitive/symmetric relations* are defined in a similar way. If a typed relation is reflexive, transitive and symmetric, then it is an *equivalence*.

(ii) A typed relation $\mathcal{R}$ is a *congruence* if $\mathcal{R}$ is an equivalence and moreover is closed under the following rules, for each inference rule (and in (par) for each selection of one side of the antecedents) of Figure 5: $\Gamma \vdash P_1\ \mathcal{R}\ P_2 \triangleright_A G \ \Rightarrow\ \Gamma' \vdash C[P_1]\ \mathcal{R}\ C[P_2] \triangleright_{A'} G'$ if, in an instance of the inference rule, $\Gamma' \vdash C[P_i] \triangleright_{A'} G'$ is the conclusion with an antecedent $\Gamma \vdash P_i \triangleright_A G$, for $i = 1, 2$, respectively. We write $\cong, \cong', ...$ for congruence relations.

4.2. **Behavioural Equality.** For the behavioural equality over typed π-terms, we use the reduction-based semantics which can be equally applicable to both untyped and typed settings. The formulation is extensively studied for untyped systems in [15, 13, 14]. First we set: $a \in \mathcal{AN}_+(P)$ iff $P \equiv (\tilde{c})(\overline{a}v.Q \mid R)$ with $a \notin \{\tilde{c}\}$, and $a \in \mathcal{AN}_-(P)$ iff $P \equiv (\tilde{c})(a x.Q \mid R)$ with $a \notin \{\tilde{c}\}$. With θ ranging over $+$ and $-$, we define a family of *action predicates*: $P \Downarrow_{a^\theta} \stackrel{\text{def}}{\Leftrightarrow} \exists P'.\ P \longrightarrow\!\!\!\!\!\!\rightarrow\ P' \ \wedge\ a \in \mathcal{AN}_\theta(P')$. Then $\Gamma \vdash P \Downarrow_{a^\theta} \triangleright_A G \stackrel{\text{def}}{\Leftrightarrow} (\Gamma \vdash P \triangleright_A G \ \wedge\ P \Downarrow_{a^\theta} \ \wedge a \notin A)$. Next we define:

- A typed congruence $\cong$ is *reduction-closed*, i.e., whenever $\Gamma \vdash P \cong Q \triangleright_A G$, $P \longrightarrow\!\!\!\!\!\!\rightarrow P'$ implies, for some Q', $Q \longrightarrow\!\!\!\!\!\!\rightarrow Q'$ and $\Gamma \vdash P' \cong Q' \triangleright_A G$.
- A typed relation $\mathcal{R}$ *respects the action predicates* iff both $\Gamma \vdash P\ \mathcal{R}\ Q \triangleright_A G$ and $\Gamma \vdash P \Downarrow_{a^\theta} \triangleright_A G$ implies $\Gamma \vdash Q \Downarrow_{a^\theta} \triangleright_A G$.

We say a congruence $\cong$ is $\Downarrow_{a^\theta}$*-sound*, or simply *sound*, if it is reduction closed and respects the action predicates. Then, by taking the congruent closure of the whole family of sound congruences, we immediately know there is the maximum sound congruence, denoted by $=_\pi$, within the family of all sound congruences.[3]

We now clarify basic behavioural properties types ensure for typed π-terms. We start from the property of reduction corresponding to cut elimination.

---

[3]We notice that we get the same equality if we use other observability predicates: $P \Downarrow_a \stackrel{\text{def}}{\Leftrightarrow} \exists\theta.\ P \Downarrow_{a^\theta}$ [30] and $P \Downarrow \stackrel{\text{def}}{\Leftrightarrow} \exists a.\ P \Downarrow_a$ [23], or even if we use the notion of insensitivity of [15], combined with reduction-closure. Another possible equational framework would be the equality called *barbed congruence*, which does not require reduction-closure at the congruence level. This construction may accompany difficulties when combined with algebraic notions such as quotient, though we believe that the same full abstraction result can be gained with it.

4.3. **Definition.** (β-reduction) The β-reduction, $\rightarrow_\beta$, is the relation generated by:

(β_1) $\Gamma \vdash ax.P \mid \overline{a}v.Q \rightarrow_\beta P\{v/x\} \mid Q \triangleright_A G \quad (a \in A)$

(β_2) $\Gamma \vdash (a)(ax.P \mid \overline{a}v.Q) \rightarrow_\beta (a)(P\{v/x\} \mid Q) \triangleright_A G$

(par) $\Gamma \vdash P \rightarrow_\beta P' \triangleright_{A_1} G_1 \ \wedge\ \Gamma \vdash Q \triangleright_{A_2} G_2 \ \Rightarrow\ \Gamma \vdash P \mid Q \rightarrow_\beta P' \mid Q \triangleright_{A'} G_1 \odot G_2$

(res) $\Gamma \vdash P \rightarrow_\beta P' \triangleright_A G \ \Rightarrow\ \Gamma\backslash a \vdash (a)P \rightarrow_\beta (a)P' \triangleright_{A\backslash\{a\}} G$

(str) $P \equiv P' \quad \Gamma \vdash P' \rightarrow_\beta Q' \triangleright_A G \quad Q \equiv Q' \ \Rightarrow\ \Gamma \vdash P \rightarrow_\beta Q \triangleright_A G$

with $A' = A_1 \cup A_2 \cup A$ plus (4) of Fig. 5 in (par), and (5) of Fig. 5 in (res), respectively. $\twoheadrightarrow_\beta$ is reflexive transitive closure of $\rightarrow_\beta$, while $=_\beta$ is the congruent closure of $\rightarrow_\beta$.

We note $\rightarrow_\beta$, which is a subset of $\longrightarrow$, generalises the *linear reduction* studied in [18, 10, 17] in that it allows the case when one name occurs many times in a process but at any one time only one input/output pair occurs actively (cf. Figure 3, see also [13, 14]). The basic properties of $\twoheadrightarrow_\beta$ and $=_\beta$ follow. (ii) is easy from (i-a,b).

4.4. **Proposition.**

(i) (a) *If* $P \twoheadrightarrow_\beta Q_1$ *and* $P \longrightarrow\!\!\!\!\twoheadrightarrow Q_2$, *then there exists* Q' *s.t.* $Q_1 \longrightarrow\!\!\!\!\twoheadrightarrow Q'$ *and* $Q_2 \twoheadrightarrow_\beta Q'$. (b) $P \twoheadrightarrow_\beta Q$ *and* $P \Downarrow_{a^\theta}$ *implies* $Q \Downarrow_{a^\theta}$.

(ii) $=_\beta$ *is a sound congruence, hence* $P =_\beta Q$ *implies* $P =_\pi Q$.

The above proposition shows $\rightarrow_\beta$ can be semantically neglected. This is essential to the understanding of the following proposition, which clarifies the relationship between types and the behaviour they guarantee. (i–v) give the case for "linear behaviour" corresponding to a graph type: a term does act as its graph type specifies, following the activation ordering in the graph modulo $\rightarrow_\beta$. Combining with the behaviour of "protocol type" shown in (vi), we know that once a term performs an atomic action $a\updownarrow(x)$ in a protocol type $a\updownarrow(x) \rightarrow G$, then it should automatically do the rest of action G (because G is now in the linear space). (vii) corresponds to the behaviour of sorting.

4.5. **Proposition.** (behaviour of typed terms) *Write* $\mathsf{Hd}(G)$ *for the set of heads in* G *and* $G\backslash\mathbf{n}$ *for the elimination of the head* $\mathbf{n}$ *from* G. *Then* $\Gamma \vdash P \triangleright_A G$ *implies:*

(i) $a\downarrow(x) \in \mathsf{Hd}(G) \ \Rightarrow\ P \twoheadrightarrow_\beta \stackrel{ax}{\longrightarrow} P'$ *and* $\Gamma \vdash P' \triangleright_A G\backslash a\downarrow(x)$ (1)

(ii) $a\uparrow(x) \in \mathsf{Hd}(G) \ \Rightarrow$
$$\begin{cases} P \twoheadrightarrow_\beta \stackrel{\overline{a}(x)}{\longrightarrow} P' \text{ and } \Gamma \vdash P' \triangleright_A G\backslash a\uparrow(x) & (1) \\ P \twoheadrightarrow_\beta \stackrel{\overline{a}x'}{\longrightarrow} P' \text{ and } \Gamma \vdash P' \triangleright_{A\backslash\{x'\}} (G\backslash a\uparrow(x))\{x'/x\} & (2) \end{cases}$$

(iii) $a\downarrow\alpha \in \mathsf{Hd}(G) \ \Rightarrow\ P \twoheadrightarrow_\beta \stackrel{ax}{\longrightarrow} P'$ *and* $\{x:\alpha\} \cup \Gamma \vdash P' \triangleright_A G\backslash a\downarrow\alpha$ (3)

(iv) $a\uparrow\alpha \in \mathsf{Hd}(G) \ \Rightarrow$
$$\begin{cases} P \twoheadrightarrow_\beta \stackrel{\overline{a}(x)}{\longrightarrow} P' \quad \text{and} \quad \{x:\alpha\} \cup \Gamma \vdash P' \triangleright_A G\backslash a\uparrow\alpha & (3) \\ P \twoheadrightarrow_\beta \stackrel{\overline{a}x}{\longrightarrow} P' \quad \text{and} \quad \Gamma \vdash P' \triangleright_A G\backslash a\uparrow\alpha & (4) \end{cases}$$

(v) $a \in \mathsf{fn}(G) \ \wedge\ a \notin \mathsf{fn}(\mathsf{Hd}(G)) \cup A \ \Rightarrow\ \neg\exists l.\ P \stackrel{l}{\Longrightarrow}$ *with* $a \in \mathsf{fn}(l)$.

(vi) *Assume* $a:\{\downarrow(x) \rightarrow G',\ \uparrow(x) \rightarrow \overline{G'}\} \in \Gamma$. *Then:*

(a) $a \in \mathcal{AN}_-(P) \ \Rightarrow\ P \stackrel{ax}{\longrightarrow} P'$ *and* $\Gamma \vdash P' \triangleright_A G \uplus G'$ (1)

(b) $a \in \mathcal{AN}_+(P) \ \Rightarrow$
$$\begin{cases} P \stackrel{\overline{a}(x)}{\longrightarrow} P' \quad \text{and} \quad \Gamma \vdash P' \triangleright_A G \uplus \overline{G} & (1) \\ P \stackrel{\overline{a}x'}{\longrightarrow} P' \quad \text{and} \quad \Gamma \vdash P' \triangleright_{A\backslash\{x'\}} (G \uplus \overline{G}'\{x'/x\}) & (2) \end{cases}$$

(vii) *Assume* $a:(\alpha) \in \Gamma$. *Then:*

(a) $a \in \mathcal{AN}_-(P) \ \Rightarrow\ P \stackrel{ax}{\longrightarrow} P'$ *and* $\{x:\alpha\} \cup \Gamma \vdash P' \triangleright_A G$ (3)

$$\text{(b)}\ \ a \in \mathcal{AN}_+(P) \ \Rightarrow \begin{cases} P \xrightarrow{\overline{a}(x)} P' \quad \textit{and} \quad \{x:\alpha\} \cup \Gamma \vdash P' \triangleright_A G & (3) \\ P \xrightarrow{\overline{a}x} P' \quad \textit{and} \quad \Gamma \vdash P' \triangleright_A G & (4) \end{cases}$$

where (1) x *fresh* (2) $x' \in A$ (3) $\{x:\alpha\} \asymp \Gamma$ *and* $x \notin (\mathsf{fn}(G) \cup A)$ (4) $x:\alpha \in \Gamma$.

5. Fully Abstract Encoding of Polyadic π-calculus

This section establishes the full abstraction result for the standard encoding of polyadic π-calculus into monadic π-calculus in terms of a basic behavioural equality in each setting, as a non-trivial application of our type discipline. We first give a brief review of polyadic π-calculus and its sorting, including behavioural equality. Then, after introducing the encodings, we make it clear what is the essential problems for such an equational embedding. Then we go into the main technical discussions, establishing the full abstraction result.

5.1. **Review of Polyadic π-calculus.** The syntax of the polyadic π-calculus is given below [21, 30].

$$P \quad ::= \quad a(x_1..x_n).P \ \mid\ \overline{a}\langle v_1..v_n\rangle.P \ \mid\ P \,|\, Q \ \mid\ (a)P \ \mid\ !P \ \mid\ \mathbf{0}$$

$a(x_1..x_n).P$ and $\overline{a}\langle v_1..v_n\rangle.P$ are input and output processes respectively, with n being the arity of the input (output) prefix. The set of *sortings* ranged over by S_1, S_2,..., is given by the following grammar:

$$S ::= \mathbf{x} \mid (S_1 S_2 ... S_n)$$

with $n \geq 0$ (we omit the recursive sorting for simplicity of presentation, cf. 6.2). $\Gamma, \Gamma', \ldots$ denote typing functions, i.e. functions from a finite subset of $\mathcal{N}$ to the above set. The typing system in [34], essentially equivalent to the system in [21], gives the sequent of form $\vdash P \succ \Gamma$, which we write $\Gamma \vdash P$ here. [34] shows the typing system satisfies the subject reduction property. To avoid ambiguity, we often write $\Gamma_{\mathrm{P}}, \Delta_{\mathrm{P}}, \ldots$ for typing functions, $\rightarrow_{\mathrm{P}}$ for the reduction, and $\xrightarrow{l}_{\mathrm{P}}$ for the early transition relation on polyadic π-calculus. The maximum sound equality over sorted polyadic π-terms is defined as in 4.2, which we denote by $=_{\mathrm{P}}$.

The following mapping of polyadic π-terms into monadic π-terms is standard except we present the mapping at the level of *types* too (which plays the central role for our main result).

5.2. **Definition.** (translation from polyadic to monadic π-calculus, [22, 21])

- Mapping for terms (with c, z fresh).

$$\begin{array}{lll} [\![\overline{a}\langle v_1..v_n\rangle.P]\!] \stackrel{\mathrm{def}}{=} \overline{a}(c).\overline{c}v_1...\overline{c}v_n.[\![P]\!] & [\![P_1|P_2]\!] \stackrel{\mathrm{def}}{=} [\![P_1]\!]|[\![P_2]\!] & [\![\mathbf{0}]\!] \stackrel{\mathrm{def}}{=} \mathbf{0} \\ [\![a(x_1..x_n).P]\!] \stackrel{\mathrm{def}}{=} az.zx_1...zx_n.[\![P]\!] & [\![(a)P]\!] \stackrel{\mathrm{def}}{=} (a)[\![P]\!] & [\![!P]\!] \stackrel{\mathrm{def}}{=} ![\![P]\!] \end{array}$$

- Mapping for types (with x fresh).

$$[\![\mathbf{x}]\!] \stackrel{\mathrm{def}}{=} \mathbf{x} \qquad [\![(S_1...S_n)]\!] \stackrel{\mathrm{def}}{=} \{\downarrow(x) \rightarrow x\downarrow[\![S_1]\!] \cdots \rightarrow x\downarrow[\![S_n]\!],\ \uparrow(x) \rightarrow x\uparrow[\![S_1]\!] \cdots \rightarrow x\uparrow[\![S_n]\!]\}$$

Notice a sorting is mapped to a protocol type which consists only of consecutive inputs or outputs. Indeed we get the following result, addressing the issue Milner raised [21].

5.3. **Proposition.** $\Gamma_{\rm P} \vdash P \quad \Leftrightarrow \quad [\![\Gamma_{\rm P}]\!] \vdash [\![P]\!]$.

This shows that, *at least at the syntactic level*, the embedding of types closely reflects the embedding of terms (which by the way seems the first result in this generality with respect to polyadic name passing, cf. [28, 27, 10, 17]). The objective of the present section is to show that this faithfulness extends to the semantic level, i.e. at the level of $=_\pi$ and $=_{\rm P}$. Before going into technical development, we need some discussions on the background of such equational embeddings.

5.4. **The Problem in Equational Embedding.** The embedding $[\![\ ]\!]$ (the term part) is a most basic example of embedding the high-level operational structures into π-calculus, cf. [22, 20, 21, 16], and, as such, presents the key issue involved in this kind of embedding in a simple setting. The power of π-calculus in encoding various calculi and programming languages lies in the use of two kinds of name passing, free and bound, to simulate one single action of a high-level operation by *a sequence of interactions which is semantically atomic*. For example, with the above encoding, we get:

$$[\![a(x_1..x_n).P]\!] \mid [\![\overline{a}\langle v_1..v_n\rangle.Q]\!] \ \longrightarrow =^u_\pi \ [\![P\{v_1..v_n/x_1..x_n\}]\!] \mid [\![Q]\!]$$

where $=^u_\pi$ is some weak behavioural equivalence, say the maximum sound equality constructed as $=_\pi$ in the untyped setting. This shows that if two encoded terms interact together, the original interaction structure is preserved in the strongest possible operational sense. Indeed the above property easily leads to the following adequacy result:

$$[\![P]\!] =^u_\pi [\![Q]\!] \quad \Rightarrow \quad P =^u_{\rm P} Q$$

where $=^u_{\rm P}$ is the basic behavioural equality in the polyadic calculus (say $=_{\rm P}$ itself). Such an embedding result motivates us to study various operational structures through their encodings into π-calculus. However, in many situations including polyadic name passing, the embedding cannot be wholly satisfactory, in the sense that the converse of the above implication does *not* hold: in another word, the translation is not equationally fully abstract. To see this, we take two terms $P \stackrel{\rm def}{=} \overline{a}\langle v\rangle.\overline{a}\langle v\rangle.\mathbf{0}$ and $Q \equiv \overline{a}\langle v\rangle.\mathbf{0} \mid \overline{a}\langle v\rangle.\mathbf{0}$ and translate them by $[\![\cdot]\!]$. Seen as polyadic terms, P and Q are strongly congruent so we safely say $P =^u_{\rm P} Q$. However $[\![P]\!] =^u_\pi [\![Q]\!]$ usually fails: With $C[\ ] \stackrel{\rm def}{=} (a)([\ ] \mid ax.ax.\overline{e}.\mathbf{0})$ where e is fresh, $C[[\![Q]\!]]$ gives an observable at e while $C[[\![P]\!]]$ does not. There are other examples. This context also explains why the completeness ($\Leftarrow$) in general fails: *if an encoding interacts with a monadic term which is not itself an encoding, the protocol can be violated, leading to the detection of difference of two originally equated terms.*

We now observe that, in the present setting too, where sorted polyadic terms are mapped to typed monadic terms, an encoding of a polyadic term *can indeed be composed to interact* with a term which is *not* the encoding itself. For example, let R be given by:

$$R \equiv (b_1 b_2)(\overline{a}(c).(b)(\overline{b}v \mid \overline{c}v_1.bz.\overline{c}v_2) \mid \overline{e}b_1.\overline{f}b_2)$$

R is clearly not in the form of the encoding, but R is typable in the way $[\![a : (\mathbf{x}_1\mathbf{x}_2)]\!] \vdash R \triangleright e\uparrow(b_1) \to f\uparrow(b_2)$, and therefore can be composed to, say, $[\![a : (\mathbf{x}_1\mathbf{x}_2)]\!] \vdash [\![a(x_1x_2).\mathbf{0}]\!]$. This shows that the completeness result, if we ever get it, may not be an easy task, since we cannot rely only on the properties of the encodings: at the same time, the above example suggests there is some hope for the completeness, since it does seem that R will interact with any encoded term *as if it were itself an encoded term*, which is explicitly specified as R's protocol types.

In the following, we give the basic technical development to validate this prospect, i.e. to establish the equational full abstraction of the standard encoding. The completeness part crucially relies on two essential roles of the types in the present setting,

i.e. as information to control the composition of terms and as the guarantee of terms' behaviour.

We start from the following operational correspondence.

5.5. **Proposition.** (operational correspondence)

(i) *If* $\Gamma_{\mathrm{P}} \vdash P \to_{\mathrm{P}} Q$, *then* $[\![\Gamma_{\mathrm{P}}]\!] \vdash [\![P]\!] \longrightarrow\!\!\!\!\!\rightarrow_{\beta} [\![Q]\!]$.

(ii) *If* $[\![\Gamma_{\mathrm{P}}]\!] \vdash [\![P]\!] \longrightarrow P'$, *then for some* Q, $[\![\Gamma_{\mathrm{P}}]\!] \vdash P' \twoheadrightarrow_{\beta} [\![Q]\!]$ *with* $\Gamma_{\mathrm{P}} \vdash P \to_{\mathrm{P}} Q$.

(iii) $\Gamma_{\mathrm{P}} \vdash P \Downarrow_{a^{\theta}} \;\Leftrightarrow\; [\![\Gamma_{\mathrm{P}}]\!] \vdash [\![P]\!] \Downarrow_{a^{\theta}}$.

For finer analysis of behaviour of typed terms, we use the following mapping of labels.

$$[\![\tau]\!] \stackrel{\text{def}}{=} \tau \qquad [\![a\langle x_1 ... x_n\rangle]\!] \stackrel{\text{def}}{=} az \cdot zx_1 \cdots zx_n \qquad [\![(\tilde{b})\overline{a}\langle v_1 ... v_n\rangle]\!] \stackrel{\text{def}}{=} \overline{a}(c) \cdot (\tilde{b}_1)cv_1 \cdots (\tilde{b}_n)cv_n$$

where c, z are fresh, $\tilde{b}_i \stackrel{\text{def}}{=} \tilde{b} \setminus v_1..v_{i-1}$ (the result of taking off each $v_1..v_{i-1}$ from $\tilde{b}$), and $(\tilde{b}_i)cv_i \stackrel{\text{def}}{=} c(v_i)$ if $v_i \in \{\tilde{b}_i\}$. The decomposition of labeled transition relation becomes:

$$\xrightarrow{[\![l]\!]} \;\stackrel{\text{def}}{\Leftrightarrow}\; \xrightarrow{l_1}\xrightarrow{l_2}\cdots\xrightarrow{l_n} \qquad\qquad \stackrel{[\![l]\!]}{\Longrightarrow} \;\stackrel{\text{def}}{\Leftrightarrow}\; \longrightarrow\!\!\!\!\!\rightarrow\xrightarrow{l_1}\longrightarrow\!\!\!\!\!\rightarrow\xrightarrow{l_2}\longrightarrow\!\!\!\!\!\rightarrow\cdots\longrightarrow\!\!\!\!\!\rightarrow\xrightarrow{l_n}\longrightarrow\!\!\!\!\!\rightarrow$$

where $[\![l]\!] = l_1 \cdot l_2 \cdots l_n$. In particular, if all $\longrightarrow\!\!\!\!\!\rightarrow$ is $\twoheadrightarrow_{\beta}$ in the second line, we write $\stackrel{[\![l]\!]}{\Longrightarrow}_{\beta}$.

We easily observe: $P \xrightarrow{l}_{\mathrm{P}} P' \;\Leftrightarrow\; [\![P]\!] \xrightarrow{[\![l]\!]} [\![P']\!]$ for $l \neq \tau$. The following key lemma for our completeness result shows how typed terms which are composable with the encodings never violate protocols the encoded terms are engaged in. For the proof, we use the behavioural properties over typed terms shown in Proposition 4.5.

5.6. **Lemma.** (Composition Lemma) *First assume* $\Gamma_{\mathrm{P}} \vdash P$. *Next suppose, with* $\Gamma_0 \supset [\![\Gamma_{\mathrm{P}}]\!]$, *we have a derivation:*

$$\frac{\Gamma_0 \vdash [\![P]\!] \qquad \Gamma_0 \vdash R \triangleright_A G}{\Gamma_0 \vdash [\![P]\!] \mid R \triangleright_A G} \; (par)$$

Then if $[\![P]\!] \mid R \longrightarrow P_0$, *one of the following is satisfied.*

(1) $P_0 \equiv [\![P]\!] \mid R'$ *with* $R \longrightarrow R'$. (2) $P_0 \twoheadrightarrow_{\beta} [\![P']\!] \mid R$ *with* $P \to_{\mathrm{P}} P'$.

(3) $P_0 \twoheadrightarrow_{\beta} (\tilde{c})([\![P']\!] \mid R')$ *with* $[\![P]\!] \xrightarrow{[\![l]\!]} [\![P']\!]$ *and* $P \xrightarrow{l}_{\mathrm{P}} P'$, *such that either:*
$l = a\langle v_1 ... v_n\rangle$ *and* $R \stackrel{[\![(\tilde{c})\overline{a}\langle v_1 ... v_n\rangle]\!]}{\Longrightarrow_{\beta}} R'$, *or* $l = (\tilde{c})\overline{a}\langle v_1 ... v_n\rangle$ *and* $R \stackrel{[\![a\langle v_1 ... v_n\rangle]\!]}{\Longrightarrow_{\beta}} R'$.

We also use the following technical lemma concerning sound congruences. We say a typed relation $\mathcal{R}$ is *substitution closed* if it is closed under: (1) $\Gamma \vdash P \;\mathcal{R}\; Q \triangleright_A G$ and $b \notin \mathsf{fn}(\Gamma) \cup A \cup \mathsf{fn}(G)$, then $\Gamma\{b/a\} \vdash P\{b/a\} \;\mathcal{R}\; Q\{b/a\} \triangleright_{A\{b/a\}} G\{b/a\}$, and (2) $a : \beta, b : \beta, \Gamma \vdash P \;\mathcal{R}\; Q \triangleright_A G$, then $b : \beta, \Gamma \vdash P\{b/a\} \;\mathcal{R}\; Q\{b/a\} \triangleright_A G$. We need a notation: $C_r[\,]_{\Delta, A_0, G_0}$ denotes a typed reduction context, i.e. a context whose hole is not under prefix nor replication, with the type of the hole specified.

5.7. **Lemma.** *Suppose* $=_s$ *is a substitution-closed sound congruence, and* $\mathcal{R}$ *is substitution-closed and is closed under parallel composition, and satisfies the following property:*

(i) *whenever* $\Gamma \vdash P \;\mathcal{R}\; Q \triangleright_A G$, $P \longrightarrow\!\!\!\!\!\rightarrow P'$ *implies, for some* Q', $Q \longrightarrow\!\!\!\!\!\rightarrow Q'$, $P' =_s C_r[P'_0]_{\Delta, A_0, G_0}$, $\Delta \vdash P'_0 \;\mathcal{R}\; Q'_0 \triangleright_{A_0} G_0$, *and* $C_r[Q'_0]_{\Delta, A_0, G_0} =_s Q'$, *and the converse.*

(ii) $\mathcal{R}$ *respects the action predicates.*

Then the congruent closure of $=_s \cup \mathcal{R}$ *is sound and substitution closed.*

For the proof, see Section 6 of [36]. Now we show the main result of this paper.

5.8. **Theorem.** (**Main Theorem, Full Abstraction**)

$$\Gamma_{\mathrm{P}} \vdash P =_{\mathrm{P}} Q \quad \Leftrightarrow \quad [\![\Gamma_{\mathrm{P}}]\!] \vdash [\![P]\!] =_{\pi} [\![Q]\!].$$

Proof outline: ($\Leftarrow$) direction is straightforward by Propositions 5.5 and 4.4. For ($\Rightarrow$) direction, we first construct the relation $\cong^{-}$ as the smallest typed relation including the following base equations and closed under parallel composition.

$$\Gamma_{\mathrm{P}} \vdash P =_{\mathrm{P}} Q \quad \Rightarrow \quad [\![\Gamma_{\mathrm{P}}]\!] \vdash [\![P]\!] \cong^{-} [\![Q]\!]$$

Clearly $\cong^{-}$ is neither congruent nor reduction-closed. However we can show $\cong^{-}$ satisfies the conditions of $\mathcal{R}$ in Lemma 5.7, taking $=_{\beta}$ for $=_{s}$. Notice $\Gamma_0 \vdash P_1 \cong^{-} Q_2 \triangleright_A G$ implies $P_1 \equiv [\![P]\!] \mid R$ and $Q_1 \equiv [\![Q]\!] \mid R$ with $\Gamma_{\mathrm{P}} \vdash P =_{\mathrm{P}} Q$ and $[\![\Gamma_{\mathrm{P}}]\!] \subset \Gamma_0$ by induction of the derivations. From this the substitution-closure of $\cong^{-}$ is easily shown, together with that of $=_{\beta}$ and $=_{\mathrm{P}}$, so that we only have to show $\cong^{-}$ satisfies (i,ii) of Lemma 5.7. For the clause (i), it is enough to prove the case when $P_1 \equiv [\![P]\!]|R$ reduces by one step. There are three cases, corresponding to (1,2,3) in Lemma 5.6. The case (1) is trivial. (2) is by Proposition 5.5 (i). In (3), the case for input is direct from an observability property of $=_{\mathrm{P}}$ (cf. 4.13 of [15]). Since values in output can not be observed (cf. [15]), the case of output is non-trivial. But we can easily find a simple context C_r which satisfies Lemma 5.7. For (ii), we again use Lemma 5.6 together with a certain context formation. For detailes, please consult [36]. □

6. Discussion

6.1. **Related Works.** While there have been many studies on types for mobile processes [21, 5, 28, 9, 34, 32, 19, 17, 10, 33], as far as we know, the present work is the first one which captures the structure of protocol construction as types and establishes the full abstraction results for the basic encoding such as polyadic name passing.

(types with sequencing): A study to give a type-abstraction in the monadic asynchronous π-calculus with the same aim had been tried by Honda [8] in 1992, whose idea is reflected to the present construction to some extent. However, his formulation (especially binding construction) was very complex and hard to grasp. Independently, Pierce discovered the way to type-check a class of encodings of polyadic name passing in (a variant of) monadic π-calculus in which two local names are used only for input and output, respectively [27]. The differences are (1) a name can only carry one kind of types so typable terms are restricted, and (2) however the length of sequence can be indefinite. In both studies, the basic results as we established here, such as the subject reduction theorem and the full abstraction, are not presented.

(linearity in processes): The notion of linearity is studied by Kobayashi and others [17] in Pict language and by [10] in an algebraic framework, though dynamic communication structures are not captured. In the former, the idea of "linearizability," based on the above mentioned Pierce's work, was proposed as an extension of their linear channel types. However structures of polyadic name passing may not be typable when a name carries different types. Another difference is our explicit treatment of name transmission in a type, which we believe to be important for the kind of results we have obtained here.

(linear logic): Some ideas of Linear Logic [7] and Interaction Categories [1] are related (e.g. "multi-cuts" in [6]). Prasad [31] studies a term assignment to a generalisation of linear logic, where the distinction between "linear" and "classical" realms exists, which is also related.

(true concurrency): The causality of communication in process calculi has been studied in the setting of true concurrency, cf. [25, 35]. The order relation in our safe graph is a concise way to encapsulate linear interaction behaviour rather than to describe general dependency among communication events. To capture more complex causality (such as labeled event structure in [35]) may be one possibility, though we may lose a simple combinatorial expression as we have in the present paper.

6.2. **Extensions and Further Issues.** There are several possible extensions and further issues of the present typing system, which we outline below.

(recursive type): For simplicity of presentation we do not form recursive protocol types. Its incorporation is standard by introducing the expressions $\mu\mathbf{x}.\alpha$ and consider them modulo their tree unfoldings, cf. [34]. Immediately all results in the paper hold with this extension w.r.t. recursive sortings.

(full abstraction): Since Prop. 4.4 and 4.5 about behaviours of typed terms do not depend on the standard encoding $[\![\ ]\!]$, we can easily extend the full abstraction result in the present paper to calculi with more complex communication structures, such as those with mixed input/output and parallel communication. Such examples are discussed in [36]. It is interesting how the present construction and its extensions can be applied to various significant translations into π-calculus, as found in [20, 16, 26, 4].

(ν and its combinators): In this paper we have exclusively dealt with graph types for synchronous monadic π-calculus. It is however possible to extend the present type discipline so that we can type ν-calculus (asynchronous monadic π-calculus) and their combinators [12, 2, 15, 13, 14] (cf. [26, 29]), which are smaller than the present system but as expressive. Because there is no direct sequencing in output, the same notion of "safety condition" cannot be used. However, an extension of the notion of graph types and the safety condition can be done so that we can form a typing system for the asynchronous calculus along the same line as in Section 5. This will be treated in the sequel to present paper.

(other typing constructs): The construction of present types can be combined with various refinements on sorting studied by e.g. in [28, 9, 32, 19, 17, 10, 33] in either monadic or polyadic setting, which will be an interesting subject of study.

(algorithm): The present system does not induce the typing algorithm because a name can be used in several ways in linear space and classical space (cf. [17]). Finding some analogue of the principal type schemes would be an interesting subject of study.

(semantics): Apart from practical viewpoints, one of the most essential issues for further study is to seek the canonical notion of semantics of types in process calculi, like "arrow types" well studied in λ-calculi and combinators [24]. Since our framework suggests one way to build up a significant computational causal chains in the basic π-calculus like functional types, we hope this work becomes a stepping stone in this line of research, cf. [11].

Acknowledgments. The author is indebted to Kohei Honda for his invaluable suggestion and continuous discussions on this work. Naoki Kobayashi provided helpful comments on the full version. She thanks Samson Abramsky, Cliff Jones, Benjamin Pierce, Kenjiro Taura and the members of Interaction Club at Edinburgh for their discussions, and Mario Tokoro for his encouragements.

References

[1] Abramsky, S., Gay, S. and Nagarajan, R., Interaction Categories and Foundations of Typed Concurrent Computing. *Deductive Program Design*, Springer-Verlag, 1995.

[2] Boudol, G., *Asynchrony and π-calculus.* INRIA Report 1702, INRIA, Sophia Antipolis, 1992.
[3] Davey, B.A. and Priestley, H.A., *Introduction to Lattices and Order*, CUP, 1990.
[4] Fournet, C. and Gonthier, G., The reflexive CHAM and the join-calculus, *POPL'96*, pp.372–385, ACM Press, 1996.
[5] Gay, S., A Sort Inference Algorithm for the Polyadic π-Calculus. *POPL'93*, ACM Press, 1993.
[6] Gay, S. and Nagarajan, R., A Typed Calculus of Synchronous Processes. *LICS'95*, IEEE, pp.210–220, 1995.
[7] Girard, J.-Y., Linear Logic, *TCS*, Vol. 50, pp.1–102, North-Holland, 1987.
[8] Honda, K., Pre-types in mobile processes, *a manuscript*, 1992.
[9] Honda, K., Types for Dyadic Interaction. *CONCUR'93*, LNCS 715, pp.509–523, Springer, 1993.
[10] Honda, K., Composing Processes, *POPL'96*, pp.344–357, ACM Press, 1996.
[11] Honda, K., Name-Passing Games (I): Semantics of Sortings, *a type-script*, July, 1996.
[12] Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication. *ECOOP'91*, LNCS 512, pp.133–147, Springer-Verlag 1991.
[13] Honda, K. and Yoshida, N., Combinatory Representation of Mobile Processes, *POPL'94*, pp.348–360, ACM Press, 1994.
[14] Honda, K. and Yoshida, N., Replication in Concurrent Combinators, *TACS'94*, LNCS 789, pp.786–805, Springer, 1994.
[15] Honda, K. and Yoshida, N., On Reduction-Based Process Semantics. *TCS*, pp.437–486, No.151, North-Holland, December, 1995.
[16] Jones, C.B., *Process-Algebraic Foundations for an Object-Based Design Notation.* UMCS-93-10-1, Manchester University, 1993.
[17] Kobayashi, N., Pierce, B., and Turner, D., Linear Types and π-calculus, *POPL'96*, pp.358–371, ACM Press, 1996.
[18] Lafont, Y., Interaction Nets, *POPL'90*, pp. 95–108, ACM press, 1990.
[19] Liu, X. and Walker, D., A polymorphic type system for the polyadic pi-calculus, *CONCUR'95*, LNCS 962, pp.103–116, Springer, 1995.
[20] Milner, R., Functions as Processes. *MSCS*, 2(2), pp.119–146, 1992.
[21] Milner, R., Polyadic π-Calculus: a tutorial. *Logic and Algebra of Specification*, Springer, 1992.
[22] Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes, *Information and Computation* 100(1), pp.1–77, 1992.
[23] Milner, R. and Sangiorgi, D., Barbed Bisimulation. *Proc. of ICALP'92*, LNCS 623, pp.685–695, Springer-Verlag, 1992.
[24] Mitchell, J., Type Systems for Programming Languages. *Handbook of Theoretical Computer Science* B, pp.367–458, MIT Press, 1990.
[25] Montanari, U. and Pistore, M., Concurrent Semantics for π-calculus, *MFCS'95* and ENTCS, Vol. 1. Elsevier, 1995.
[26] Odersky, M., Polarized Name Passing. *FSTTCS'15*, LNCS 1026, pp.324–335, Springer, 1995.
[27] Pierce, B.C., Linearized Types for the π-calculus, *a type script*, December, 1994.
[28] Pierce, B.C. and Sangiorgi. D, Typing and subtyping for mobile processes. *LICS'93*, pp.187–215, IEEE, 1993.
[29] Raja, N. and Shyamasundar, R.K., Combinatory Formulations of Concurrent Languages, pp. 156–170, ACSC'95, LNCS 1023, Springer-Verlag, 1995.
[30] Sangiorgi, D., *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms.* Ph.D. Thesis, University of Edinburgh, 1992.
[31] Prasad, S., *Towards a Formulae-as-Types View of Communicating Applicative Processes*, Technical report ECRC-94-32, ECRC, 1994.
[32] Takeuchi, K., Honda, K. and Kubo, M., An Interaction-based Language and its Typing System. *PARLE'94*, LNCS 817, pp.398–413, Springer-Verlag, 1994.
[33] Turner, D., The π-calculus: Types, polymorphism and implementation, Phd Thesis, University of Edinburgh, 1996.
[34] Vasconcelos, V. and Honda, K., Principal Typing Scheme for Polyadic π-Calculus. *CONCUR'93*, LNCS 715, pp.524-538, Springer-Verlag, 1993.
[35] Winskel, G., An Introduction to Event Structures. *In Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, pp. 364–397,
[36] Yoshida, N., Graph Types for Mobile Process Calculi I. The full version of this paper. CS technical report, Keio University, 1996. Available from `ftp.dcs.ed.ac.uk/export/ny/pub/graph1.ps.Z.`