Sequence Types for the π-calculus

Sergio Maffeis [1]

Department of Computing,
Imperial College London,
London SW7 2AZ, United Kingdom.

Abstract

We introduce *channel sequence types* to study finitary polymorphism in the context of mobile processes modelled in the π-calculus. We associate to each channel a set of exchange types, and we require that output processes send values of one of those types, and input processes accept values of any type in the set. Our type assignment system enjoys subject reduction and guarantees the absence of communication errors. We give several examples of polymorphism, and we encode the λ-calculus with the strict intersection type discipline.

Key words: Polymorphism, Mobile Processes, Intersection types, Union Types, Encodings.

1 Introduction

Intersection types were introduced in the late 1970s by Coppo and Dezani-Ciancaglini [5], with the aim of defining a typing system for the λ-calculus where types are preserved by β-conversion, and where every term possessing a normal form has a meaningful type. Since then, intersection types have been studied extensively and have found many applications, starting from the work by Reynolds on polymorphism for the Forsythe programming language [17].

From a programming language perspective, intersection types express a finite, yet unbounded, amount of information about a program, called *finitary polymorphism*. For example, an arithmetic function could be given the type $Int \to Int \cap Real \to Real$, meaning that it is either generic on its arguments (*universal* polymorphism) or that it provides two different implementations in case of the arguments (and result) being either integers or real numbers (*ad-hoc* polymorphism). We argue that a similar approach (specifying the set of types of messages that a channel can exchange) can be adopted to type

[1] Email: maffeis@doc.ic.ac.uk. Supported by Microsoft Research Cambridge and an EPSRC e-Science Grant.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

communication in process calculi, and yields more flexibility than the usual existential/universal approach to polymorphism.

We study finitary polymorphism for the π-calculus [13], a fundamental formalism to reason about concurrent and mobile processes. A recent study by Berger, Honda and Yoshida [3] on genericity, highlights how the two endpoints of a communication channel in π-calculus play a dual role with respect to universal and existential type quantification. Accordingly, we have found that intersection and union types for channels cannot be disentangled, and must be considered at once as what we call *sequence* types, resorting to the original name used by Coppo and Dezani-Ciancaglini. We associate to each channel a set of exchange types, and we require that output processes send values of one of those types, and input processes accept values of all the types in the set. We consider also the case when the set is empty, which corresponds to the constant ω of the intersection type discipline: a channel to which is associated the empty set cannot be used for communication.

Related Work. Some important intersection type assignment systems are proposed in [5,7,2]. Strict intersection types are introduced in [19], and their principal typing properties are studied in [20]. A comparison between some of these approaches can be found in [21]. Intersection types have been used to define filter-models for the λ-calculus [2], the π-calculus [8], and a variant of Mobile Ambients [6]. Anyway, the approach and the aims of [8] are different to the ones of this paper. In particular the types for the filter model are assigned to processes, and reflect the process structure quite closely (containing constructors for replication, input, matching, parallel composition, etc.), whereas we give type to channels only, and our types are simply sets representing the possible exchange types for a channel. Our type system aims at preventing communication errors, rather than precisely describing the behaviour of terms.

Polymorphism in its various flavours has been studied extensively in the π-calculus: universal [9,24], predicative [23], existential [18,16] and generic [3]. A different approach is the one of [10], which defines a sort inference system based on tuples of input and output sorts (which could resemble our sequences of types). Their system is incomparable to ours, since it adopts a *nominal* rather than a *structural* approach, it does not enjoy subject reduction, and in order to prevent communication errors it resorts to an auxiliary notion of *consistent types* which rules out, for example, the possibility for a channel to exchange types with different arities, which can be acceptable in our system.

As far as we are aware, this paper represents the first attempt to study finitary polymorphism in the context of the π-calculus.

Structure of the Paper. In Section 2 we present the polyadic π-calculus and its basic types. In Section 3 we introduce sequence types through a type assignment system which enjoys subject reduction and which guarantees the absence of communication errors, and we give examples of finitary polymor-

phism. In Section 4 we give a type-preserving encoding of the call-by-name λ-calculus with the strict intersection type discipline into the π-calculus with sequence types. In Section 5 we conclude with an extended example showing how sequence types can be used to type inductively-defined polymorphic data structures.

2 The π-calculus

2.1 Syntax and Semantics

For simplicity, we use the polyadic variant of the π-calculus [13] without the silent action prefix and mixed choice. Let $\mathcal{N}$ be a denumerable set of channel names, ranged over by x, y, z, w. We use the notation $\tilde{x}$ for the tuple $x_1, \ldots, x_n$, and we write $\tilde{x}_n$ when the specific n is important. Processes are given by

$$P ::= \mathbf{0} \mid P \mid P \mid\ !P \mid (\nu x)P \mid y(\tilde{x}).P \mid \overline{y}\langle\tilde{x}\rangle.P$$

The empty process is denoted by $\mathbf{0}$, $P \mid Q$ is the parallel composition of P and Q, $!P$ (replication) denotes an unbounded number of copies of P in parallel, $(\nu x)P$ restricts the name x in P, $y(\tilde{x}).P$ denotes the process ready to perform an input along channel y, binding the variables $\tilde{x}$ in P, and $\overline{y}\langle\tilde{x}\rangle.P$ is the output process ready to pass $\tilde{x}$ along y. Processes are identified up-to the structural congruence relation $\equiv$, defined as the least congruence relation satisfying alpha-conversion of bound names, the commutative monoidal laws for $(\mid, \mathbf{0})$, and the axioms reported below

$$\begin{array}{rr} (\nu z)\mathbf{0} \equiv \mathbf{0} & (\nu z)(\nu w)P \equiv (\nu w)(\nu z)P \\ z \notin fn(P) \Rightarrow (\nu z)(P \mid Q) \equiv P \mid (\nu z)Q & !P \equiv P \mid !P \end{array}$$

We use the standard syntactic conventions of omitting trailing zeros in processes and writing $\overline{x}.P$ and $x.P$ for output and input of the empty tuple along channel x. The reduction semantics is the least binary relation on processes $\longrightarrow$ satisfying the rules given in Table 1. The axiom (Com) is responsible for the communication of the values $\tilde{z}_n$ along channel y.

A communication error condition is verified when two processes attempt to communicate on two end-points of the same channel with different arities.

Definition 2.1 We define the error predicate on processes $-\dagger$ as

$$P\dagger \iff P \equiv (\nu\tilde{z})(P_1 \mid y(\tilde{x}_n).R \mid \overline{y}\langle\tilde{w}_m\rangle.Q)$$

where $m \neq n$. If it is not the case that $P\dagger$, we write $P \not\dagger$.

2.2 Basic Type Assignment System

The basic Curry-style type assignment system for the π-calculus, as presented in [18], is based on an environment Γ (a partial function) associating an ex-

$$(\textsc{Com})\ y(\tilde{x}_n).P \mid \overline{y}\langle \tilde{z}_n \rangle.Q \longrightarrow P\{\tilde{z}_n/\tilde{x}_n\} \mid Q$$

$$(\textsc{C-Res})\ \frac{P \longrightarrow Q}{(\nu x)P \longrightarrow (\nu x)Q} \qquad (\textsc{C-Par})\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

$$(\textsc{Struct})\ \frac{N \equiv N' \qquad N' \longrightarrow M' \qquad M' \equiv M}{N \longrightarrow M}$$

Table 1
Reduction Rules

$$(\textsc{Par})\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad (\textsc{Nil})\frac{}{\Gamma \vdash \mathbf{0}}$$

$$(\textsc{Rep})\frac{\Gamma \vdash P}{\Gamma \vdash\ !P} \qquad (\textsc{Res})\frac{\Gamma, x : T \vdash P}{\Gamma \vdash (\nu x)P}$$

$$(\textsc{B-In})\frac{\Gamma(z) = [\tilde{T}] \qquad \Gamma, \tilde{x} : \tilde{T} \vdash P}{\Gamma \vdash z(\tilde{x}).P} \quad (\textsc{B-Out})\frac{\Gamma(z) = [\tilde{T}] \qquad \Gamma(\tilde{x}) = \tilde{T} \qquad \Gamma \vdash P}{\Gamma \vdash \overline{z}\langle \tilde{x} \rangle.P}$$

Table 2
Basic type assignment system for the π-calculus.

change type to each channel, describing the objects that can be communicated. The formal definition of basic types and environments is

$$\begin{aligned} (\textsc{B-Types}) \qquad & S, T ::= [\tilde{T}] \\ (\textsc{Type environments}) \qquad & \Gamma ::= \Gamma, x : T \mid \emptyset \end{aligned}$$

The statement $x : [\tilde{T}]$ means that x is a channel exchanging tuples of values with the types specified by $\tilde{T}$. If $\tilde{x} = x_1, \ldots, x_n$ and $\tilde{T} = T_1, \ldots T_n$, we use the shorthand notation $\tilde{x} : \tilde{T}$ for $x_1 : T_1, \ldots, x_n : T_n$. If Γ contains $\tilde{x} : \tilde{T}$ we write $\Gamma(\tilde{x}) = \tilde{T}$. The type assignment rules are given in Table 2, and are defined up-to alpha-conversion of bound names.

It is a well-established result that the basic type system satisfies the standard property of preserving types under reduction, and guarantees that well-typed processes will not incur in communication errors.

Theorem 2.2 ([18]) *(i) If* $\Gamma \vdash P$ *and* $P \rightarrow Q$*, then* $\Gamma \vdash Q$*. (ii) If* $\Gamma \vdash P$ *then* $P \not\rightarrow$*.*

Note that from the Theorem 2.2 and rule (Par), follows that, if P does not

contain a communication error in an environment Γ, neither do its derivatives, nor the systems obtained by composing P in parallel with other processes well-typed with respect to the same environment.

Throughout the paper we will re-use the symbols $\vdash, \Gamma, S, T$: the precise meaning will always be clear from the context.

3 Channel Sequence Types

3.1 Sequence Type Assignment System

The sequence type assignment system for the π-calculus is a proper extension of the basic type assignment system. In this new system a channel x can be assigned a set of types $\circ\{[\tilde{T}_1]; \ldots; [\tilde{T}_n]\}$, with the dual interpretation that the type of x is sometimes the *intersection* $[\tilde{T}_1] \cap \ldots \cap [\tilde{T}_n]$ and sometimes the *union* $[\tilde{T}_1] \cup \ldots \cup [\tilde{T}_n]$ of the types in the sequence. This apparently strange property can be understood considering the duality between the types of a name when used as an input or as an output studied in [3], and by comparing the rules (S-In) and (S-Out) below with the classical elimination rules for union and intersection types in the λ-calculus.

The formal definition of channel sequence types is given by

$$\begin{array}{rc} (\text{S-Types}) & S, T ::= \circ\{[\tilde{T}_1]; \ldots; [\tilde{T}_n]\},\ n \geq 0 \\ (\text{Type environments}) & \Gamma ::= \Gamma, x : T \mid \emptyset \end{array}$$

where to each name x is assigned a (possibly empty) set $\circ\{[\tilde{T}_1]; \ldots; [\tilde{T}_n]\}$ of exchange types, which in turn are channel sequence types. For readability, we write $[T]$ for the singleton sequence type $\circ\{[T]\}$, and we use the reserved name ω for sequences over the empty set $\circ\emptyset$, with the intuitive meaning that the corresponding channel *cannot* be used for communication. Types of the form $[\tilde{T}, [\tilde{S}]]$ play a special role in encoding the functional behaviour of channels, and for readability will sometimes be abbreviated by $\tilde{T} \to \tilde{S}$.

The type assignment system for sequence types is given by the rules of Table 2, where the rules (B-In) and (B-Out) are replaced by

$$(\text{S-In})\ \frac{\Gamma(z) = \circ\{[\tilde{T}_1]; \ldots; [\tilde{T}_n]\} \qquad \forall i \in 1..n \,.\, \Gamma, \tilde{x} : \tilde{T}_i \vdash P}{\Gamma \vdash z(\tilde{x}).P}$$

$$(\text{S-Out})\ \frac{\Gamma(z) = \circ\{[\tilde{T}_1]; \ldots; [\tilde{T}_n]\} \qquad \exists i \in 1..n \,.\, \Gamma(\tilde{x}) = \tilde{T}_i \qquad \Gamma \vdash P}{\Gamma \vdash \overline{z}\langle\tilde{x}\rangle.P}$$

Rule (S-In) can be intuitively explained as applying rule (B-In) to every possible type for channel z, and rule (S-Out) is the dual, obtained using (B-Out) on at least one of the possible types for the objects of communication.

The type system preserves types under reduction and guarantees that well-typed processes will not incur in communication errors.

Theorem 3.1 *(i) If* $\Gamma \vdash P$ *and* $P \to Q$*, then* $\Gamma \vdash Q$*. (ii) If* $\Gamma \vdash P$ *then* $P \not\downarrow$*.*

3.2 Examples

We show how ω can be used to type terms which contain the potential for errors only in points not reachable during an execution.

Example 3.2 (The relevance of ω) The process $x(y).(\overline{z}\langle y\rangle \mid z(w,y).Q)$ can be typed if $\Gamma(x) = \omega$. Under this assumption it is never possible to type an output prefix with subject x, as rule (S-Out) requires the existence of at least one exchange type for x. Consequently, the process can never reduce to one containing a top-level communication error. Conversely, the process $\overline{x}\langle y\rangle \mid \overline{x}\langle y,y\rangle$ can be typed for example for $\Gamma = y : T, x : \circ\{[T];[T,T]\}$, but no process attempting an input on x can be composed to this process, since rule (S-In) imposes two incompatible conditions on the number of variables bound by channel x. Therefore, the communication error condition cannot be verified. Both processes above cannot be typed in the usual type systems for the π-calculus which rule out communication errors.

It should come as no surprise that sequence types can be used to express polymorphism. Below, we give two concrete examples.

Example 3.3 (Generic polymorphism) The standard definition of a process `Id` behaving like the identity function in π-calculus is given by

$$!\mathtt{Id}(x,y).\overline{y}\langle x\rangle$$

A process can send any value v along with a fresh channel z, and receive the same value v on z. The term above has a well-defined semantics with respect to both the invocations below

$$(\nu z)\overline{\mathtt{Id}}\langle a,z\rangle.z(w).P \qquad (\nu z)\overline{\mathtt{Id}}\langle b,z\rangle.z(w).Q$$

but if a and b happen to have different types, then their parallel composition cannot be typed in the basic type system. Intuitively, channel `Id` should be typable for all types T such that $x : T$ and $y : [T]$. Using sequence types, we can give it a type for any (finite) set of types. For example, we show that given $\Gamma = \{a : T, b : S, \mathtt{Id} : \circ\{T \to T; S \to S\}\}$ the composition of the three processes given above is well-typed:

$$\Gamma \vdash (\nu z)\overline{\mathtt{Id}}\langle a,z\rangle.z(w).P \mid (\nu z)\overline{\mathtt{Id}}\langle b,z\rangle.z(w).Q \mid !\mathtt{Id}(x,y).\overline{y}\langle x\rangle \qquad (1)$$

We show the interesting part of the derivation:

$$\Gamma \vdash (\nu z)\overline{\mathtt{Id}}\langle a, z\rangle.z(w).P \quad \text{by (T-Par)} \quad (2)$$

$$\Gamma_1 = \Gamma, z : [T], \quad \Gamma_1 \vdash \overline{\mathtt{Id}}\langle a, z\rangle.z(w).P \quad \text{by (T-Res)} \quad (3)$$

$$\left.\begin{array}{r} \Gamma_1(\mathtt{Id}) = \circ\{T \to T; S \to S\} \\ \Gamma_1 \vdash a : T, z : [T] \\ \Gamma_1 \vdash z(w).P \end{array}\right\} \quad \text{by (S-Out)} \quad (4)$$

$$\Gamma \vdash (\nu z)\overline{\mathtt{Id}}\langle b, z\rangle.z(w).Q \quad \text{by (T-Par)} \quad (5)$$

$$\Gamma_2 = \Gamma, z : [S], \quad \Gamma_2 \vdash \overline{\mathtt{Id}}\langle b, z\rangle.z(w).Q \quad \text{by (T-Res)} \quad (6)$$

$$\left.\begin{array}{r} \Gamma_2(\mathtt{Id}) = \circ\{T \to T; S \to S\} \\ \Gamma_2 \vdash b : S, z : [S] \\ \Gamma_2 \vdash z(w).Q \end{array}\right\} \quad \text{by (S-Out)} \quad (7)$$

$$\Gamma \vdash !\mathtt{Id}(x, y).\overline{y}\langle x\rangle \quad \text{by (T-Par)} \quad (8)$$

$$\Gamma \vdash \mathtt{Id}(x, y).\overline{y}\langle x\rangle \quad \text{by (T-Rep)} \quad (9)$$

$$\left.\begin{array}{r} \Gamma(\mathtt{Id}) = \circ\{T \to T; S \to S\} \\ \Gamma, x : T, y : [T] \vdash \overline{y}\langle x\rangle \\ \Gamma, x : S, y : [S] \vdash \overline{y}\langle x\rangle \end{array}\right\} \quad \text{by (S-In)} \quad (10)$$

The crucial points are (4), (7) and (10). In the former two cases we need only check the arguments of communication along $\mathtt{Id}$ for one of the possible tuples $T, [T]$ and $S, [S]$, and in the latter case we have to check that the final output $\overline{y}\langle x\rangle$ is well-typed for both types assumed for x, y, again $T, [T]$ and $S, [S]$.

Example 3.4 (Ad-hoc polymorphism) In this example we show that a value received on a polymorphic channel y can then be used in two different ways, depending on its type. In order to do so, we must provide two different implementations (in this case of the output on z) and guard each of them with an input (respectively on $\mathtt{one}$ and $\mathtt{two}$) used to recognise the specific type of the polymorphic value (here represented by w). The process

$$(\nu\, \mathtt{one}, \mathtt{two})(y(z, x).(\overline{x}\langle z\rangle \mid \mathtt{one}(z).\overline{z}\langle a\rangle \mid \mathtt{two}(z).\overline{z}\langle a, a\rangle) \mid \overline{y}\langle b, \mathtt{one}\rangle \mid \overline{y}\langle c, \mathtt{two}\rangle)$$

can be typed in the environment

$$\Gamma = a : T, b : [T], c : [T, T], y : \circ\{[T] \to [T]; [T, T] \to [T, T]\}$$

Both processes given in Example 3.3 and Example 3.4 can be typed in the polymorphic system of Turner [18]; in Section 5 we show an example which cannot be typed in that system.

4 Encoding of the λ-calculus

4.1 Strict intersection types for the λ-calculus

We report below a definition of the strict intersection type assignment system of van Bakel for the λ-calculus, along the lines of [19]. We adapt the definition from the natural deduction style, and we use sets of types in the intersections to avoid an explicit definition of equivalence between types. Below, let α be a type variable.

$$\begin{aligned} &(\textsc{Strict Types}) && \tau ::= \alpha \mid \cap\{\tau_1;\ldots;\tau_n\} \to \tau \quad n \geq 0 \\ &(\textsc{Strict Intersection Types}) && \sigma = \cap\{\tau_1;\ldots;\tau_n\} \quad n \geq 0 \\ &(\textsc{Basis}) && B ::= B, x:\sigma \mid \emptyset \end{aligned}$$

We denote with ω the empty intersection, and we write τ for $\cap\{\tau\}$. We denote a basis by B, which is a partial function from variables to strict intersection types. Strict derivations are:

$$(\to \text{E})\ \frac{B \vdash M : \cap\{\tau_1;\ldots;\tau_n\} \to \tau \quad N : \tau_1 \ldots N : \tau_n}{B \vdash M\ N : \tau}\ (n \geq 0)$$

$$(\to \text{I})\ \frac{B, x:\sigma \vdash M : \tau}{B \vdash \lambda x.M : \sigma \to \tau} \qquad (\cap \text{E})\ \frac{B(x) = \cap\{\tau_1;\ldots;\tau_n\}}{B \vdash x : \tau_i}\ (n \geq 1, i \in [1..n])$$

Note that in rule $(\to \text{I})$ x can have a strict intersection type σ. A term M has type σ with respect to a base B, written $B \vdash_S M : \sigma$, if $\sigma = \omega$, or if $\sigma = \cap\{\tau_1;\ldots;\tau_n\}$ and for all $i \in [1..n]$, $B \vdash M : \tau_i$. We report below some interesting properties of this type system (from [19]):

(i) A term M has a normal form if and only if there exists B, σ, not containing ω, such that $B \vdash_S M : \sigma$.

(ii) A term M has a head normal form if and only if there exists B, τ such that $B \vdash M : \tau$.

(iii) A term M is strongly normalizable if and only if there exists B, σ such that $B \vdash_S M : \sigma$ and ω does not appear in B, σ, or in any step of the derivation.

4.2 Encoding into the π-calculus

Below we report the untyped encoding of the call-by-name λ-calculus into the π-calculus, as proposed by Ostheimer and Davie [14]. The encoding is parametric on a continuation channel a used to communicate the result of evaluating a term.

$$\begin{aligned} [\![x]\!]_a &= \overline{x}\langle a\rangle \\ [\![\lambda x.M]\!]_a &= (\nu f)(\overline{a}\langle f\rangle \mid !f(x,b).[\![M]\!]_b) \\ [\![M\ N]\!]_a &= (\nu b, x)([\![M]\!]_b \mid b(f).(\overline{f}\langle x, a\rangle \mid !x(c).[\![N]\!]_c)) \end{aligned}$$

We give a translation of strict types into sequence types which is a proper extension of the one for basic types given by Turner [18] (for the purpose of the encoding we assume to have type variables α also in our type system).

$$\langle\!\langle \alpha \rangle\!\rangle = \alpha$$
$$\langle\!\langle \cap\{\tau_1; \ldots; \tau_n\} \to \tau \rangle\!\rangle = [\circ\{[[\langle\!\langle \tau_1 \rangle\!\rangle]]; \ldots; [[\langle\!\langle \tau_n \rangle\!\rangle]]\}, [\langle\!\langle \tau \rangle\!\rangle]]$$

A type variable remains unchanged, whereas a function type is translated in a channel type containing the encoding of the intersection of the argument types into a sequence, where each type is a doubly nested channel containing the encoding of the corresponding argument, and a channel containing the encoding of the result type. The translation of basis and judgments is given by

$$\langle\!\langle B, x : \cap\{\tau_1; \ldots; \tau_n\} \rangle\!\rangle = \langle\!\langle B \rangle\!\rangle, x : \circ\{[[\langle\!\langle \tau_1 \rangle\!\rangle]]; \ldots; [[\langle\!\langle \tau_n \rangle\!\rangle]]\}$$
$$[\![B \vdash M : \tau]\!]_a = \langle\!\langle B \rangle\!\rangle, a : [\langle\!\langle \tau \rangle\!\rangle] \vdash [\![M]\!]_a$$

The translation preserves types, as stated by the theorem below.

Theorem 4.1 *If $B \vdash M : \tau$ then $[\![B \vdash M : \tau]\!]_a$.*

Proof. By structural induction on M.

(i) x: if $B \vdash x : \tau$ then rule (∩ E) has been applied, and we know that $B = B', x : \cap\{\tau_1; \ldots; \tau_n\}$ with $\tau = \tau_i$ for some i. We need to show that $\langle\!\langle B' \rangle\!\rangle, x : \circ\{[[\langle\!\langle \tau_1 \rangle\!\rangle]]; \ldots; [[\langle\!\langle \tau_n \rangle\!\rangle]]\}, a : [\langle\!\langle \tau \rangle\!\rangle] \vdash \overline{x}\langle a \rangle$, which follows by rule (S-Out).

(ii) $\lambda x.M$: if $B \vdash \lambda x.M : \cap\{\tau_1; \ldots; \tau_n\} \to \tau$ then rule (→ I) has been applied, and we know that $B, x : \cap\{\tau_1; \ldots; \tau_n\} \vdash M : \tau$. By inductive hypothesis we have $\langle\!\langle B \rangle\!\rangle, x : \circ\{[[\langle\!\langle \tau_1 \rangle\!\rangle]]; \ldots; [[\langle\!\langle \tau_n \rangle\!\rangle]]\}, b : [\langle\!\langle \tau \rangle\!\rangle] \vdash [\![M]\!]_b$. We need to show that $\Gamma \vdash (\nu f)(\overline{a}\langle f \rangle \mid !f(x, b).[\![M]\!]_b)$ for $\Gamma = \langle\!\langle B \rangle\!\rangle, a : [[\circ\{[[\langle\!\langle \tau_1 \rangle\!\rangle]]; \ldots; [[\langle\!\langle \tau_n \rangle\!\rangle]]\}, [\langle\!\langle \tau \rangle\!\rangle]]$. By (Res) it suffices to show that $\Gamma_1 = \Gamma, f : [\circ\{[[\langle\!\langle \tau_1 \rangle\!\rangle]]; \ldots; [[\langle\!\langle \tau_n \rangle\!\rangle]]\}, [\langle\!\langle \tau \rangle\!\rangle]] \vdash \overline{a}\langle f \rangle \mid !f(x, b).[\![M]\!]_b$. By rules (Par) and (S-Out) the term on the left is well typed. By rules (Par) and (Rep) we need to show that $\Gamma_1 \vdash f(x, b).[\![M]\!]_b$. Since f is a unary intersection, applying rule (S-In) it is enough to show that $\Gamma_1, x : \circ\{[[\langle\!\langle \tau_1 \rangle\!\rangle]]; \ldots; [[\langle\!\langle \tau_n \rangle\!\rangle]]\}, b : [\langle\!\langle \tau \rangle\!\rangle] \vdash [\![M]\!]_b$, which follows by applying weakening to the inductive hypothesis.

(iii) $M\ N$: if $B \vdash M\ N : \tau$ then rule (→ E) has been applied, and we have both $B \vdash M : \cap\{\tau_1; \ldots; \tau_n\} \to \tau$ and, for all $i \in [1..n]$, $B \vdash N : \tau_i$. The reasoning is similar to the previous case, the non-trivial point being the application of rule (S-In) on the subterm $x(c).[\![N]\!]_c$ where x has the intersection type derived from the function argument. The rule requires to use all the n inductive hypothesis derived from the premises $B \vdash N : \tau_i$. □

Remark 4.2 Theorem 4.1 implies that if $B \vdash_S M : \sigma$ with $\sigma = \cap\{\tau_1; \ldots; \tau_n\}$

and $n \geq 1$, then we can type the encoding of M also in π-calculus, by definition of $\vdash_S$. On the other hand, we cannot type M if $\sigma = \omega$, because for example in the case of a variable term we would not have any binding for x in Γ. As noted by van Bakel, ω is effectively needed to type those subterms which are erased during the reduction of a term towards its normal form, and which cannot be typed otherwise. The encoding reflects this property by assigning ω to input variables which will simply be discarded during reduction since, as explained in Example 3.2, ω can be used in π-calculus to *avoid* typing the continuation of an input which is known never to be evaluated.

The encoding of a lambda-term in π-calculus may receive types which are not in the image of the encoding of λ-calculus types, because of the freedom given by rule (Res). Nevertheless, if we impose the condition that every derivation in the π-calculus system assigns to restricted names arising from the encoding of an application some types in the image of λ-types, we can prove also the converse direction of Theorem 4.1.

Theorem 4.3 *If $[\![B \vdash M : \tau]\!]_a$ with a derivation assigning to the restricted names b, x of the encoding of an application respectively types of the form $[\langle\!\langle \rho \rangle\!\rangle]$ and $\circ\{[[\langle\!\langle \rho_1 \rangle\!\rangle]]; \ldots; [[\langle\!\langle \rho_n \rangle\!\rangle]]\}$, where ρ is a strict type, then $B \vdash M : \tau$.*

Proof. By induction on M, similar to the proof of Theorem 4.1, using the condition on the types that can be assumed for restricted names. □

The previous theorem has the interesting consequence that the properties (1-3) reported above for the strict type assignment system can be transferred to the image of λ-terms in π-calculus. Moreover, since the encoding preserves termination [14], in the case of (3) we obtain a type-based characterisation of a set of terminating π-processes (those in the image of the encoding) *as large as* the set of all the strongly normalising λ-terms. An interesting topic for further research would be a direct characterisation of the π-calculus terms typable with sequence types (with and without ω).

Remark 4.4 We have chosen Ostheimer and Davie's encoding of the call-by-name (CBN) λ-calculus rather than Milner's one because the former is closer to the optimised call-by-value (CBV) encoding used by Turner (already mentioned in [11]). The two encodings induce different types on π-terms in the case of simply typed λ-calculus (see [18]), yet enjoy similar properties. In the case of sequence types, we were not able to define a typed encoding for the optimised CBV encoding. The problem arises with the encoding of the variable term, which is $[\![x]\!]_a = \overline{a}\langle x \rangle$. Given this definition, the types for the judgment $x : \cap\{\tau_1; \ldots; \tau_n\} \vdash x : \tau_i$ are not preserved since the object of communication x has more types than the subject a (compare with point (1) in the proof of Theorem 4.1). We leave it to future work to investigate Milner's original CBV encoding.

5 Polymorphic Inductive Datatypes

It is well known (see Milner [12]) how to encode data structures in the π-calculus. For example, we report below the encoding of lists and list operations as given by Turner [18].

$$\begin{aligned}
&!\mathtt{nil}(r).(\nu l)(\overline{r}\langle l\rangle \mid !l(n,c).\overline{n})\\
&!\mathtt{cons}(h,t,r).(\nu l)(\overline{r}\langle l\rangle \mid !l(n,c).\overline{c}\langle h,t\rangle)\\
&!\mathtt{ccat}(l,m,r).(\nu n,c)(\overline{l}\langle n,c\rangle.(n.\overline{r}\langle m\rangle \mid\\
&\qquad c(h,t).(\nu s)(\overline{\mathtt{ccat}}\langle t,m,s\rangle \mid s(res).\overline{\mathtt{cons}}\langle h,res,r\rangle)))
\end{aligned}$$

The process $(\nu r)\overline{\mathtt{nil}}\langle r\rangle.r(l).P$ creates a new empty list l for the continuation P. If P is in turn $(\nu r)\overline{\mathtt{cons}}\langle h,l,r\rangle.r(m).Q$, now Q knows about a list called m with first element h and with tail the empty list $!l(n,c).\overline{n}$. It is possible to concatenate lists similar to the ones just described using the operation `ccat` given above.

In the usual π-calculus approach, recursive types are necessary to type lists[2]. Moreover, even if all the operations on lists are generic, and work for data objects of any type, a simple recursive type system for the π-calculus could type the operations only for one kind of data at a time. For example, it would rule out the (perfectly reasonable) term

$$(\nu intr, strr)(\overline{\mathtt{ccat}}\langle intl1, intl2, intr\rangle \mid \overline{\mathtt{ccat}}\langle strl1, strl2, strr\rangle \mid P)$$

where P can receive from $intr$ the concatenation of the two lists of integers $intl1, intl2$, and from $strr$ the concatenation of the two lists of strings $strl1, strl2$.

In our type system it is possible to typecheck this term, for example in an environment Γ such that

$$\begin{aligned}
&\Gamma(\mathtt{ccat}) = \circ(\{IL(h,k) \mid 0 \le h \le m, 0 \le k \le n\}\\
&\qquad\qquad \cup \{SL(h,k) \mid 0 \le h \le i, 0 \le k \le j\})\\
&\Gamma(intl1) = IntL_n \quad \Gamma(intl2) = IntL_m\\
&\Gamma(srtl1) = StrL_i \quad \Gamma(strl2) = StrL_j
\end{aligned}$$

where we have used the abbreviations

$$IL(h,k) = (IntL_h, IntL_k) \to IntL_{h+k} \quad SL(h,k) = (StrL_h, StrL_k) \to StrL_{h+k}$$

Sequence types give us both the power to express finitary polymorphism *and* inductive data structures.

[2] Turner shows in [18] that Church-encoded lists (which can hardly be regarded as list themselves, since selecting the tail is not a constant-time operation) can be typed in the polymorphic π-calculus without using recursive types.

6 Conclusions and Future Work

We have introduced the notion of sequence types for π-calculus channels, which play the role of both intersection and union types due to the intrinsic duality of channel typing. We have shown that a Curry-like type assignment system enjoys subject reduction and guarantees the absence of communication errors, and we have given examples of terms typable with this discipline.

We have given a type-preserving encoding of the λ-calculus with strict intersection types, deriving a type-based characterisation of a rather large set (computationally speaking) of terminating π-terms: those resulting from the encoding of strongly normalising λ-terms. This gives a hint of the undecidability of our type system.

We have shown that sequence types can be used to type inductively defined, polymorphic data structures.

We leave it to future work a formal study of principal typing and decidability properties. One way to enforce decidability, following Pierce [15], consists in adopting a *typed system* (Church-style), annotating restriction and input variables with types. Alternatively, rank 2 intersection types have been studied for example in [22] as a decidable restriction of the intersection typing discipline which still enjoys many interesting properties. A corresponding notion of rank 2 for our system is straightforward to define, and rank 2 intersection types can be translated into sequence types of the same rank. We conjecture that an analogous of Theorem 4.1 and Theorem 4.3 holds, but we leave it to future work to study the formal properties of the restricted subsystem.

Our original motivation to study intersection types was the attempt to give an expressive structural type-system for the ${}^{e}\pi$-calculus [4], an extension of the π-calculus where channels are identified by vector of names (*synchronisation vectors*). In ${}^{e}\pi$, a natural choice could be to type a channel with the intersection of the types of the names constituting the synchronisation vector. We expect this line of research to be promising, in view of the ability of the calculus to encode faithfully cryptographic primitives and distributed calculi.

It would also be interesting to study sequence types in the context of Ambient Calculi, in particular comparing with the work of Amtoft *et.al.* [1] on ML-like polymorphism (where principal typing holds only for a restricted fragment of Ambients), and where the use of shape types may be regarded as a form of finitary polymorphism.

Apart from the definitions of filter models [8,6] mentioned in the introduction, this work constitutes the first study of intersection types in process calculi, and we hope that will serve as an inspiration for further research in the area.

Acknowledgments. This work was inspired by a discussion with Steffen van Bakel on polyadic-synchronisation types for the ${}^{e}\pi$-calculus. I am grateful to the anonymous referees for helpful comments and suggestions.

References

[1] Amtoft, T., H. Makholm and J. Wells, *Polya: True type polymorphism for mobile ambients*, Technical Report HW-MACS-TR-0015 School of Mathematical and Computer Sciences Heriot-Watt University. To appear in the Proceedings of IFIP-TCS'04. (2004).

[2] Barendregt, H., M. Coppo and M. Dezani-Ciancaglini, *A filter lambda model and the completeness of type assignment*, Journal of Symbolic Logic **48** (1983), pp. 931–940.

[3] Berger, M., K. Honda and N. Yoshida, *Genericity and the pi-calculus*, , **2620**.

[4] Carbone, M. and S. Maffeis, *On the expressive power of polyadic synchronisation in π-calculus*, Nordic Journal of Computing **10** (2003), pp. 70–98.

[5] Coppo, M. and M. Dezani-Ciancaglini, *An extension of the basic functionality theory for the λ-calculus*, Notre-Dame Journal of Formal Logic **21** (1980), pp. 685–693.

[6] Coppo, M. and M. Dezani-Ciancaglini, *A fully abstract model for higherorder mobile ambients*, in: *VMCAI '02*, LNCS **2294** (2002), pp. 255–271.

[7] Coppo, M., M. Dezani-Ciancaglini and B. Venneri, *Functional characters of solvable terms*, Zeitschrift f ur Mathematische Logik und Grundlagen der Mathematik (1981), pp. 45–58.

[8] Damiani, F., M. Dezani-Ciancaglini and P. Giannini, *A filter model for mobile processes*, Mathematical Structures in Computer Science **9** (1999), pp. 63–102.

[9] Gay, S. J., *A sort inference algorithm for the polyadic π-calculus*, in: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1993), pp. 429–438.

[10] Liu, X. and D. Walker, *A polymorphic type system for the polyadic pi-calculus*, in: *Proceedings of the 6th International Conference on Concurrency Theory* (1995), pp. 103–116.

[11] Milner, R., *Functions as processes*, INRIA RR-1154. (Revised version in *Mathematical Structures in Computer Science* 2(2), 1992). (1990).

[12] Milner, R., *The polyadic π-calculus: A tutorial*, in: *Logic and Algebra of Specification*, Springer-Verlag, Heidelberg, 1993 .

[13] Milner, R., J. Parrow and J. Walker, *A calculus of mobile processes, I and II*, Information and Computation **100** (1992), pp. 1–40,41–77.

[14] Ostheimer, G. K. and A. J. T. Davie, *π-calculus characterisations some practical λ-calculus reduction strategies*, Technical Report CS/93/14, Department of Mathematical and Computing Sciences, University of St Andrews (1993).

[15] Pierce, B. C., "Programming with Intersection Types and Bounded Polymorphism," Ph.D. thesis, Carnegie Mellon University (1991).

[16] Pierce, B. C. and D. Sangiorgi, *Behavioral equivalence in the polymorphic pi-calculus*, J. ACM **47** (2000), pp. 531–584.

[17] Reynolds, J. C., *Preliminary design of the programming language forsythe*, Technical Report CMU-CS-88-159, Carnegie Mellon University (1988).

[18] Turner, D. N., "The Polymorphic Pi-calculus: Theory and Implementation," Ph.D. thesis, University of Edinburgh (1995).

[19] van Bakel, S., *Complete restrictions of the intersection type discipline*, Theoretical Computer Science (1992), pp. 135–163.

[20] van Bakel, S., *Principal type schemes for the strict type assignment system*, Journal of Logic and Computation **3** (1993), pp. 643–670.

[21] van Bakel, S., *Intersection type assignment systems*, Theoretical Computer Science (1995), pp. 385–435.

[22] van Bakel, S., *Rank 2 intersection type assignment in term rewriting systems*, Fundamenta Informatica **2** (1996).

[23] Vasconcelos, V. T., *Predicative polymorphism in pi-calculus*, in: *Parallel Architectures and Languages Europe*, 1994, pp. 425–437.

[24] Vasconcelos, V. T. and K. Honda, *Principal typing schemes in a polyadic pi-calculus*, in: *Proceedings of the 4th International Conference on Concurrency Theory* (1993), pp. 524–538.

A Subject reduction and Safety

Lemma A.1 (Strengthening) *If* $\Gamma, x : T \vdash P$ *and* $x \notin fn(P)$ *then* $\Gamma \vdash P$.

Lemma A.2 (Weakening) *If* $\Gamma \vdash P$ *then* $\Gamma, x : T \vdash P$ *for any type* T *and any* $x \notin dom(\Gamma)$.

Both the previous lemmas follows easily by structural induction on P.

Lemma A.3 (Substitution Lemma) *If* $\Gamma, x : T \vdash P$ *and* $\Gamma(y) = T$ *then* $\Gamma, x : T \vdash P\{y/x\}$.

Proof. By induction on the structure of P.

- 0: base case, trivial.
- $P \mid Q$: using the inductive hypothesis on P and Q and rule (Par).
- $!P$: trivial.
- $(\nu z)P$: Given $\Gamma, x : T \vdash (\nu z)P$ and $\Gamma(y) = T$ first of all we alpha convert z to some fresh name w. By rule (Res), for some type S, we have $\Gamma, x : T, w : S \vdash P\{w/z\}$, and by inductive hypothesis $\Gamma, x : T, w : S \vdash P\{w/z\}\{y/x\}$, which by (Res) and alpha-conversion gives us $\Gamma, x : T \vdash (\nu z)P\{y/x\}$, where we have used the fact that w is fresh.

- $\overline{z}\langle w\rangle.P$: by syntactical reasoning, using the inductive hypothesis on the rightmost precondition for rule (S-Out).
- $z(w).P$: combining the previous two points.

□

Lemma A.4 (Subject Congruence) *If* $\Gamma \vdash P$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q$.

Proof. By induction on the definition of $\equiv$. We show the case for $(\nu z)(P \mid Q) \equiv P \mid (\nu z)Q$ where $z \notin fn(P)$, as the other ones are easy. From $\Gamma \vdash (\nu z)(P \mid Q)$ by rule (Res) we have, for some type S, that $\Gamma, z : S \vdash P \mid Q$, by rule (Par) we have that $\Gamma, z : S \vdash P$ and $\Gamma, z : S \vdash Q$, by Lemma A.1 we obtain $\Gamma \vdash P$. Using (Res) we get $\Gamma \vdash (\nu z)Q$ and by (Par) we conclude with $\Gamma \vdash P \mid (\nu z)Q$. The other direction is symmetric, and uses weakening instead of strengthening. □

Subject Reduction (Theorem 2.2.i). *If* $\Gamma \vdash P$ *and* $P \to Q$, *then* $\Gamma \vdash Q$.

Proof. By induction on the depth of the inference of $P \to Q$.

- (Com): suppose $y(\tilde{x}_n).P \mid \overline{y}\langle\tilde{z}_n\rangle.Q \longrightarrow P\{\tilde{z}_n/\tilde{x}_n\} \mid Q$ and $\Gamma \vdash y(\tilde{x}_n).P \mid \overline{y}\langle\tilde{z}_n\rangle.Q$. By (Par) we have $\Gamma \vdash y(\tilde{x}_n).P$ and $\Gamma \vdash \overline{y}\langle\tilde{z}_n\rangle.Q$. By (S-Out) we have $\Gamma(y) = \circ\{[\tilde{T}_1]; \ldots; [\tilde{T}_m]\}$, $\Gamma(\tilde{z}) = \tilde{T}_i$, and $\Gamma \vdash Q$. By (S-In) we also have $\Gamma, \tilde{x} : \tilde{T}_j \vdash P$ for any j, and in particular for i. By repeatedly applying the substitution lemma (Lemma A.3), we get $\Gamma, \tilde{x} : \tilde{T}_j \vdash P\{\tilde{z}/\tilde{x}\}$, and by strengthening and (Par), we conclude with $\Gamma \vdash P\{\tilde{z}_n/\tilde{x}_n\} \mid Q$.
- (C-Res): by (Res), the inductive hypothesis, and again (Res).
- (C-Par): by (Par), the inductive hypothesis, and again (Par).
- (Struct): by Lemma A.4, the inductive hypothesis, and again Lemma A.4.

□

Safety (Theorem 2.2.ii). *If* $\Gamma \vdash P$ *then* $P \not\dagger$.

Proof. We show that for all P, if $P\dagger$ then there is no Γ such that $\Gamma \vdash P$. By definition of $-\dagger$ we have that $P \equiv (\nu\tilde{z})(P_1 \mid y(\tilde{x}_n).R \mid \overline{y}\langle\tilde{w}_m\rangle.Q)$ where $m \neq n$. We show by contradiction. Suppose $\Gamma \vdash P$, by Lemma A.4 we would have that $\Gamma \vdash (\nu\tilde{z})(P_1 \mid y(\tilde{x}_n).R \mid \overline{y}\langle\tilde{w}_m\rangle.Q)$. Repeatedly using (Res) and (Par) we reach the two sub-derivations for rules (S-In) and (S-Out) with conclusion, respectively $\Gamma \vdash y(\tilde{x}_n).R$ and $\Gamma \vdash \overline{y}\langle\tilde{w}_m\rangle.Q$. The first condition of both rules forces y to have some type $\circ\{[\tilde{T}_1]; \ldots; [\tilde{T}_j]\}$. By the second condition of (S-In) all the $\tilde{T}_i$ must have the same arity, precisely n, in order to match with $\tilde{x}_n$, but by the second condition of (S-Out) there must be an index h such that $\tilde{T}_h$ has arity m, hence a contradiction since $m \neq n$ by hypothesis. If y has type ω, then it is not possible to derive the second precondition for (S-Out). □