Termination in a Pi-calculus with Subtyping

Ioana Cristescu, Daniel Hirschkoff

Termination in a π-calculus with Subtyping

Ioana Cristescu Daniel Hirschkoff
ENS Lyon, Université de Lyon, CNRS, INRIA, France

We present a type system to guarantee termination of π-calculus processes that exploits input/output capabilities and subtyping, as originally introduced by Pierce and Sangiorgi, in order to analyse the usage of channels.

We show that our system improves over previously existing proposals by accepting more processes as terminating. This increased expressiveness allows us to capture sensible programming idioms. We demonstrate how our system can be extended to handle the encoding of the simply typed λ-calculus, and discuss questions related to type inference.

1 Introduction

Although many concurrent systems, such as servers, are supposed to run forever, termination is an important property in a concurrent setting. For instance, one would like a request to a server to be eventually answered; similarly, the access to a shared resource should be eventually granted. Termination can be useful to guarantee in turn lock-freedom properties [8].

In this work, we study termination in the setting of the π-calculus: concurrent systems are specified as π-calculus processes, and we would like to avoid situations in which a process can perform an infinite sequence of internal communication steps. Despite its conciseness, the π-calculus can express complex behaviours, such as reconfiguration of communication topology, and dynamic creation of channels and threads. Guaranteeing termination is thus a nontrivial task.

More specifically, we are interested in methods that provide termination guarantees statically. There exist several type-based approaches to guarantee termination in the π-calculus [5, 15, 12, 3, 4]. In these works, any typable process is guaranteed to be reactive, in the sense that it cannot enter an infinite sequence of internal communications: it eventually terminates computation, or ends up in a state where an interaction with the environment is required.

The type systems in the works mentioned above have different expressive powers. Analysing the expressiveness of a type system for termination amounts to studying the class of processes that are recognised as terminating. A type system for termination typically rules out some terminating terms, because it is not able to recognise them as such (by essence, an effective type system for termination defines an approximation of this undecidable property). When improving expressiveness, one is interested in making the type system more flexible: more processes should be deemed as terminating. An important point in doing so is also to make sure that (at least some of) the 'extra processes' make sense from the point of view of programming.

Type systems for termination in the π-calculus. Existing type systems for termination in the π-calculus build on simple types [13], whereby the type of a channel describes what kind of values it can carry. Two approaches, that we shall call 'level-based' and 'semantics-based', have been studied to guarantee termination of processes. We discuss below the first kind of methods, and return to semantics-based approaches towards the end of this section. Level-based methods for the termination of processes

B. Luttik and F. D. Valencia (Eds.): 18th International Workshop on
Expressiveness in Concurrency (EXPRESS 2011)
EPTCS 64, 2011, pp. 44–58, doi:10.4204/EPTCS.64.4

originate in [5], and have been further analysed and developed in [3]. They exploit a stratification of names, obtained by associating a *level* (given by a natural number) to each name. Levels are used to insure that at every reduction step of a given process, some well-founded measure defined on processes decreases.

Let us illustrate the level-based approach on some examples. In this paper, we work in the asynchronous π-calculus, and replication can occur only on input prefixes. As in previous work, adding features like synchrony or the sum operator to our setting does not bring any difficulty.

According to level-based type systems, the process $!a(x).\overline{b}\langle x\rangle$ is well-typed provided $\mathsf{lvl}(a)$, the level of a, is strictly greater than $\mathsf{lvl}(b)$. Intuitively, this process trades messages on a (that 'cost' $\mathsf{lvl}(a)$) for messages on b (that cost less). Similarly, $!a(x).(\overline{b}\langle x\rangle \mid \overline{b}\langle x\rangle)$ is also well-typed, because none of the two messages emitted on b will be liable to trigger messages on a ad infinitum. More generally, for a process of the form $!a(x).P$ to be typable, we must check that all messages occurring in P are transmitted on channels whose level is strictly smaller than $\mathsf{lvl}(a)$ (more accurately, we only take into account those outputs that do not occur under a replication in P — see Section 2).

This approach rules out a process like $!a(x).\overline{b}\langle x\rangle \mid !b(y).\overline{a}\langle y\rangle$ (which generates the unsatisfiable constraint $\mathsf{lvl}(a) > \mathsf{lvl}(b) > \mathsf{lvl}(a)$), as well as the other obviously 'dangerous' term $!a(x).\overline{a}\langle x\rangle$ — note that neither of these processes is diverging, but they lead to infinite computations as soon as they are put in parallel with a message on a.

The limitations of simple types. The starting point of this work is the observation that since existing level-based systems rely on simple types, they rule out processes that are harmless from the point of view of termination, essentially because in simple types, all names transmitted on a given channel should have the same type, and hence, in our setting, the same level as well.

If we try for instance to type the process $P_0 \stackrel{def}{=} !a(x).\overline{x}\langle t\rangle$, the constraint is $\mathsf{lvl}(a) > \mathsf{lvl}(x)$, in other words, the level of the names transmitted on a must be smaller than a's level. It should therefore be licit to put P_0 in parallel with $\overline{a}\langle p\rangle \mid \overline{a}\langle q\rangle$, provided $\mathsf{lvl}(p) < \mathsf{lvl}(a)$ and $\mathsf{lvl}(q) < \mathsf{lvl}(a)$. Existing type systems enforce that p and q have *the same type* for this process to be typable: as soon as two names are sent on the same channel (here, a), their types are unified. This means that if for some reason (for instance, if the subterm $!p(z).\overline{q}\langle z\rangle$ occurs in parallel) we must have $\mathsf{lvl}(p) > \mathsf{lvl}(q)$, the resulting process is rejected, although it is terminating.

We would like to provide more flexibility in the handling of the level of names, by relaxing the constraint that p and q from the example above should have the same type. To do this while preserving soundness of the type system, it is necessary to take into account the way names are used in the continuation of a replicated input. In process P_0 above, x is used in output in the continuation, which allows one to send on a any name (of the appropriate simple type) of level *strictly smaller* than $\mathsf{lvl}(a)$. If, on the other hand, we consider process $P_1 \stackrel{def}{=} !b(y).!y(z).\overline{c}\langle z\rangle$, then typability of the subterm $!y(z).\overline{c}\langle z\rangle$ imposes $\mathsf{lvl}(y) > \mathsf{lvl}(c)$, which means that any name of level *strictly greater* than $\mathsf{lvl}(c)$ can be sent on b. In this case, P_1 uses the name y that is received along b in input. We can remark that divergent behaviours would arise if we allowed the reception of names having a bigger (resp. smaller) level in P_0 (resp. P_1).

Contributions of this work. These observations lead us to introduce a new type system for termination of mobile processes based on Pierce and Sangiorgi's system for *input/output types* (i/o-types) [11]. I/o-types are based on the notion of *capability* associated to a channel name, which makes it possible to grant only the possibility of emitting (the output capability) or receiving (the input capability) on a given channel. A subtyping relation is introduced to express the fact that a channel for which both capabilities

are available can be coerced to a channel where only one is used. Intuitively, being able to have a more precise description of how a name will be used can help in asserting termination of a process: in P_0, only the output capability on x is used, which makes it possible to send a name of smaller level on a; in P_1, symmetrically, y can have a bigger level than expected, as only the input capability on y is transmitted.

The overall setting of this work is presented in Section 2, together with the definition of our type system. This system is strictly more expressive than previously existing level-based systems. We show in particular that our approach yields a form of 'level polymorphism', which can be interesting in terms of programming, by making it possible to send several requests to a given *server* (represented as a process of the form $!f(x).P$, which corresponds to the typical idiom for functions or servers in the π-calculus) with arguments that must have different levels, because of existing dependencies between them.

In order to study more precisely the possibility to handle terminating functions (or servers) in our setting, we analyse an encoding of the λ-calculus in the π-calculus. We have presented in [3] a counterexample showing that existing level-based approaches are not able to recognise as terminating the image of the simply-typed λ-calculus (STλ) in the π-calculus (all processes computed using such an encoding terminate [13]). We show that this counterexample is typable in our system, but we exhibit a new counterexample, which is not. This shows that despite the increased expressiveness, level-based methods for the termination of π-calculus processes fail to capture terminating sequential computation as expressed in STλ.

To accommodate functional computation, we exploit the work presented in [4], where an *impure* π-calculus is studied. Impure means here that one distinguishes between two kinds of names. On one hand, *functional names* are subject to a certain discipline in their usage, which intuitively arises from the way names are used in the encoding of STλ in the π-calculus. On the other hand, *imperative names* do not obey such conditions, and are called so because they may lead to forms of stateful computation (for instance, an input on a certain name is available at some point, but not later, or it is always available, but leads to different computations at different points in the execution).

In [4], termination is guaranteed in an impure π-calculus by using a level-based approach for imperative names, while functional names are dealt with separately, using a semantics-based approach [15, 12]. We show that that type system, which combines both approaches for termination in the π-calculus, can be revisited in our setting. We also demonstrate that the resulting system improves in terms of expressiveness over [4], from several points of view.

Several technical aspects in the definition of our type systems are new with respect to previous works. First of all, while the works we rely on for termination adopt a presentation à la Church, where every name has a given type a priori, we define our systems à la Curry, in order to follow the approach for i/o-types in [11]. As we discuss below, this has some consequences on the soundness proof of our systems. Another difference is in the presentation of the impure calculus: [4] uses a specific syntactical construction, called `def`, and akin to a `let .. in` construct, to handle functional names. By a refinement of i/o-types, we are able instead to enforce the discipline of functional names without resorting to a particular syntactical construct, which allows us to keep a uniform syntax.

We finally discuss type inference, by focusing on the case of the *localised* π*-calculus* (Lπ). Lπ corresponds to a certain restriction on i/o-types. This restriction is commonly adopted in implementations of the π-calculus. We describe a sound and complete type inference procedure for our level-based system in Lπ. We also provide some remarks about inference for i/o-types in the general case.

Paper outline. Section 2 presents our type system, and shows that it guarantees termination. We study its expressiveness in Section 3. Section 4 discusses type inference, and we give concluding remarks in

Section 5. For lack of space, several proofs are omitted from this version of the paper.

2 A Type System for Termination with Subtyping

2.1 Definition of the Type System

Processes and types. We work with an infinite set of *names*, ranged over using $a,b,c,\ldots,x,y,\ldots$. Processes, ranged over using $P,Q,R,\ldots$, are defined by the following grammar ($\star$ is a constant, and we use v for values):

$$P \;::=\; \mathbf{0} \mid P_1|P_2 \mid \overline{a}\langle v\rangle \mid (\boldsymbol{\nu} a)\,P \mid a(x).P \mid !a(x).P \qquad\qquad v \;::=\; \star \mid a \;\;.$$

The constructs of restriction and (possibly replicated) input are binding, and give rise to the usual notion of α-conversion. We write $\mathrm{fn}(P)$ for the set of free names of process P, and $P[b/x]$ stands for the process obtained by applying the capture-avoiding substitution of x with b in P.

We moreover implicitly assume, in the remainder of the paper, that all the processes we manipulate are written in such a way that all bound names are pairwise distinct and are different from the free names. This may in particular involve some implicit renaming of processes when a reduction is performed.

The grammar of types is given by:

$$T \;::=\; \sharp^k T \mid \mathsf{i}^k T \mid \mathsf{o}^k T \mid \mathbb{U} \;\;,$$

where k is a natural number that we call a *level*, and $\mathbb{U}$ stands for the `unit` type having $\star$ as only value. A name having type $\sharp^k T$ has level k, and can be used to send or receive values of type T, while type $\mathsf{i}^k T$ (resp. $\mathsf{o}^k T$) corresponds to having only the input (resp. output) capability.

Figure 1 introduces the subtyping and typing relations. We note $\leq$ both for the subtyping relation and for the inequality between levels, as no ambiguity is possible. We can remark that the input (resp. output) capability is covariant (resp. contravariant) w.r.t. $\leq$, but that the opposite holds for levels: input requires the supertype to have a smaller level.

Γ ranges over typing environments, which are partial maps from names to types – we write $\Gamma(a) = T$ if Γ maps a to T. $\mathrm{dom}(\Gamma)$, the domain of Γ, is the set of names for which Γ is defined, and $\Gamma, a : T$ stands for the typing environment obtained by extending Γ with the mapping from a to T, this operation being defined only when $a \notin \mathrm{dom}(\Gamma)$.

The typing judgement for processes is of the form $\Gamma \vdash P : w$, where w is a natural number called the *weight* of P. The weight corresponds to an upper bound on the maximum level of a channel that is used in output in P, without this output occurring under a replication. This can be read from the typing rule for output messages (notice that in the first premise, we require the output capability on a, which may involve the use of subtyping) and for parallel composition. As can be seen by the corresponding rules, non replicated input prefix and restriction do not change the weight of a process. The weight is controlled in the rule for replicated inputs, where we require that the level of the name used in input is strictly bigger than the weight of the continuation process. We can also observe that working in a synchronous calculus would involve a minor change: typing a synchronous output $\overline{a}\langle v\rangle.P$ would be done essentially like typing $\overline{a}\langle v\rangle \,|\, P$ in our setting (with no major modification in the correctness proof for our type system).

As an abbreviation, we shall omit the content of messages in prefixes, and write a and $\overline{a}$ for $a(x)$ and $\overline{a}\langle\star\rangle$ respectively, when a's type indicates that a is used to transmit values of type $\mathbb{U}$.

Example 1 *The process $!a(x).\overline{x}\langle t\rangle \,|\, \overline{a}\langle p\rangle \,|\, \overline{a}\langle q\rangle \,|\, !p(z).\overline{q}\langle z\rangle$ from Section 1 can be typed in our type system: we can set $a : \sharp^3\mathsf{o}^2 T, p : \sharp^2 T, q : \mathsf{o}^1 T$. Subtyping on levels is at work in order to typecheck the subterm $\overline{a}\langle q\rangle$. We provide a more complex term, which can be typed using similar ideas, in Example 15 below.*

Subtyping $\leq$ is the least relation that is reflexive, transitive, and satisfies the following rules:

$$\frac{}{\sharp^k T \leq \mathsf{i}^k T} \qquad \frac{}{\sharp^k T \leq \mathsf{o}^k T} \qquad \frac{T \leq S \qquad k_1 \leq k_2}{\mathsf{i}^{k_2} T \leq \mathsf{i}^{k_1} S} \qquad \frac{T \leq S \qquad k_1 \leq k_2}{\mathsf{o}^{k_1} S \leq \mathsf{o}^{k_2} T}$$

Typing values

$$\frac{}{\Gamma \vdash \star : \mathbb{U}} \qquad \frac{\Gamma(a) = T}{\Gamma \vdash a : T} \qquad \frac{\Gamma \vdash a : T \qquad T \leq U}{\Gamma \vdash a : U}$$

Typing processes

$$\frac{}{\Gamma \vdash \mathbf{0} : 0} \qquad \frac{\Gamma \vdash a : \mathsf{o}^k T \qquad \Gamma \vdash v : T}{\Gamma \vdash \overline{a}\langle v \rangle : k} \qquad \frac{\Gamma \vdash a : \mathsf{i}^k T \qquad \Gamma, x : T \vdash P : w}{\Gamma \vdash a(x).P : w}$$

$$\frac{\Gamma \vdash a : \mathsf{i}^k T \qquad \Gamma, x : T \vdash P : w \qquad k > w}{\Gamma \vdash !a(x).P : 0} \qquad \frac{\Gamma, a : T \vdash P : w}{\Gamma \vdash (\boldsymbol{\nu} a) P : w} \qquad \frac{\Gamma \vdash P_1 : w_1 \qquad \Gamma \vdash P_2 : w_2}{\Gamma \vdash P_1 | P_2 : \max(w_1, w_2)}$$

Figure 1: Typing and Subtyping Rules

$$\frac{}{a(x).P \mid \overline{a}\langle v \rangle \longrightarrow P[v/x]} \qquad \frac{}{!a(x).P \mid \overline{a}\langle v \rangle \longrightarrow !a(x).P \mid P[v/x]}$$

$$\frac{P \longrightarrow P'}{P|Q \longrightarrow P'|Q} \qquad \frac{P \longrightarrow P'}{(\boldsymbol{\nu} a) P \longrightarrow (\boldsymbol{\nu} a) P'} \qquad \frac{Q \equiv P \qquad P \longrightarrow P' \qquad P' \equiv Q'}{Q \longrightarrow Q'}$$

Figure 2: Reduction of Processes

Reduction and termination. The definition of the operational semantics relies on a relation of structural congruence, noted $\equiv$, which is the smallest equivalence relation that is a congruence, contains α-conversion, and satisfies the following axioms:

$$P|(Q|R) \equiv (P|Q)|R \qquad P|Q \equiv Q|P \qquad P|\mathbf{0} \equiv P$$

$$(\boldsymbol{\nu} a)\mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu} a)(\boldsymbol{\nu} b) P \equiv (\boldsymbol{\nu} b)(\boldsymbol{\nu} a) P \qquad (\boldsymbol{\nu} a)(P|Q) \equiv P \,|\, (\boldsymbol{\nu} a) Q \text{ if } a \notin \mathrm{fn}(P)$$

Note in particular that there is no structural congruence law for replication.

Reduction, written $\longrightarrow$, is defined by the rules of Figure 2.

Definition 2 (Termination) *A process P* diverges *if there exists an infinite sequence of processes $(P_i)_{i \geq 0}$ such that $P = P_0$ and for any i, $P_i \longrightarrow P_{i+1}$. P* terminates *(or P is terminating) if P does not diverge.*

2.2 Properties of the Type System

We first state some (mostly standard) technical properties satisfied by our system.

Lemma 3 *If $\Gamma \vdash P : w$ and $w \neq 0$ then for any $w' \geq w$, $\Gamma \vdash P : w'$.*

Proposition 4 (Narrowing) *If $\Gamma, x : T \vdash P : w$ and $T' \leq T$, then $\Gamma, x : T' \vdash P : w'$ for some $w' \leq w$.*

Lemma 5 *If $P \equiv Q$, then $\Gamma \vdash P : w$ iff $\Gamma \vdash Q : w$.*

Lemma 6 *If $\Gamma, x : T \vdash P : w$, $\Gamma \vdash b : T'$ and $T' \leq T$ then $\Gamma \vdash P[b/x] : w'$, for some $w' \leq w$.*

Proof (sketch). *This is a consequence of Lemma 4, as we replace x by a name of smaller type.*

Theorem 7 (Subject reduction) *If $\Gamma \vdash P : w$ and $P \longrightarrow P'$, then $\Gamma \vdash P' : w'$ for some $w' \leq w$.*

Proof (sketch). *By induction over the derivation of $P \longrightarrow P'$. The most interesting case corresponds to the case where $P = !a(x).P_1 \mid \overline{a}\langle v \rangle \longrightarrow P' = !a(x).P_1 \mid P_1[v/x]$. By typability of P, we have $\Gamma \vdash !a(x).P_1 : 0$. Let $T_a = \Gamma(a)$. Typability of P gives $\Gamma, x : T \vdash P_1 : w_1$ for some T and w_1 such that $T_a \leq \mathsf{i}^k T$ and $w_1 < k \leq \mathsf{lvl}(a)$. Typability of $\overline{a}\langle v \rangle$ gives $T_a \leq \mathsf{o}^{k'} U$ for some $k' \geq \mathsf{lvl}(a)$, with $w = k'$ and $\Gamma \vdash v : U$. The two constraints on T_a entail $T \leq U$, and hence, by Lemma 6, $\Gamma \vdash P_1[v/x] : w_2$ for some $w_2 \leq w_1 \leq \mathsf{lvl}(a) \leq k'$. We then conclude $\Gamma \vdash P' : w_2$.*

Termination. Soundness of our type system, that is, that every typable process terminates, is proved by defining a measure on processes that decreases at each reduction step. A typing judgement $\Gamma \vdash P : w$ yields the weight w of process P, but this notion is not sufficient (for instance, $\overline{a} \mid \overline{a} \mid a \longrightarrow \overline{a}$, and the weight is preserved). We instead adapt the approach of [1], and define the measure as a multiset of natural numbers. This is done by induction over the derivation of a typing judgement for the process. We will use $\mathscr{D}$ to range over typing derivations, and write $\mathscr{D} : \Gamma \vdash P : w$ to mean that $\mathscr{D}$ is a derivation of $\Gamma \vdash P : w$.

To deduce termination, we rely on the multiset extension of the well-founded order on natural numbers, that we write $>_{mul}$. $M_2 >_{mul} M_1$ holds if $M_1 = N \uplus N_1$, $M_2 = N \uplus N_2$, N being the maximal such multiset ($\uplus$ is multiset union), and for all $e_1 \in N_1$ there is $e_2 \in N_2$ such that $e_1 < e_2$. The relation $>_{mul}$ is well-founded. We write $M_1 \geq_{mul} M_2$ if $M_1 >_{mul} M_2$ or $M_1 = M_2$.

Definition 8 *Suppose $\mathscr{D} : \Gamma \vdash P : w$. We define a multiset of natural numbers, noted $\mathscr{M}(\mathscr{D})$, by induction over $\mathscr{D}$ as follows:*

$$\text{If } \mathscr{D} : \Gamma \vdash \mathbf{0} \text{ then } \mathscr{M}(\mathscr{D}) = \emptyset \qquad \text{If } \mathscr{D} : \Gamma \vdash \overline{a}\langle b \rangle : k \text{ then } \mathscr{M}(\mathscr{D}) = \{lvl(a)\}$$

$$\text{If } \mathscr{D} : \Gamma \vdash !a(x).P : 0 \text{ then } \mathscr{M}(\mathscr{D}) = \emptyset$$

$$\text{If } \mathscr{D} : \Gamma \vdash a(x).P : w, \text{ then } \mathscr{M}(\mathscr{D}) = \mathscr{M}(\mathscr{D}_1), \text{ where } \mathscr{D}_1 : \Gamma, x : T \vdash P : w$$

$$\text{If } \mathscr{D} : \Gamma \vdash (\boldsymbol{\nu} a)\, P : w, \text{ then } \mathscr{M}(\mathscr{D}) = \mathscr{M}(\mathscr{D}_1), \text{ where } \mathscr{D}_1 : \Gamma, a : T \vdash P : w$$

$$\text{If } \mathscr{D} : \Gamma \vdash P_1 | P_2 : \max(w_1, w_2), \text{ then } \mathscr{M}(\mathscr{D}) = \mathscr{M}(\mathscr{D}_1) \uplus \mathscr{M}(\mathscr{D}_2), \text{ where } \mathscr{D}_i : \Gamma \vdash P_i, i = 1, 2$$

Given Γ and P, we define $\mathscr{M}_\Gamma(P)$, the measure of P with respect to Γ, as follows:

$$\mathscr{M}_\Gamma(P) = \min(\mathscr{M}(\mathscr{D}), \mathscr{D} : \Gamma \vdash P : w \quad \text{for some } w) \ .$$

Note that in the case of output in the above definition, we refer to $\mathsf{lvl}(a)$, which is the level of a according to Γ (that is, without using subtyping). We have that if $\Gamma \vdash P : w$, then $\forall k \in \mathscr{M}_\Gamma(P), k \leq w$.

Lemma 9 *Suppose $\Gamma \vdash P : w$, $\Gamma(x) = T$, $\Gamma(v) = T'$ and $T' \leq T$. Then $\mathscr{M}_\Gamma(P) \geq_{mul} \mathscr{M}_\Gamma(P[v/x])$.*

Proof *Follows from Lemma 6, and by definition of $\mathscr{M}_\Gamma(\cdot)$.* □

Lemma 10 *If $\Gamma \vdash P : w$ and $P \equiv Q$, then $\Gamma \vdash Q : w'$ for some w' and $\mathscr{M}_\Gamma(P) = \mathscr{M}_\Gamma(Q)$.*

We are now able to derive the essential property of $\mathscr{M}_\Gamma(\cdot)$:

Lemma 11 *If $\Gamma \vdash P : w$ and $P \longrightarrow P'$, then $\mathscr{M}_\Gamma(P) >_{mul} \mathscr{M}_\Gamma(P')$.*

Theorem 12 (Soundness) *If $\Gamma \vdash P : w$, then P terminates.*

Proof *Suppose that P diverges, i.e., there is an infinite sequence $(P_i)_{i \in N}$, where $P_i \longrightarrow P_{i+1}$, $P = P_0$. According to Theorem 7 every P_i is typable. Using Lemma 11 we have $\mathscr{M}_\Gamma(P_i) >_{mul} \mathscr{M}_\Gamma(P_{i+1})$ for all i, which yields a contradiction.* □

Remark 13 (à la Curry vs à la Church) *Our system is presented à la Curry. Existing systems for termination [5, 4] are à la Church, while the usual presentations of i/o-types [11] are à la Curry. The latter style of presentation is better suited to address type inference (see Section 4). This has however some technical consequences in our proofs. Most importantly, the measure on processes (Definition 8) would be simpler when working à la Church, because we could avoid to consider all possible derivations of a given judgement. We are not aware of Church-style presentations of i/o-types.*

3 Expressiveness of our Type System

For the purpose of the discussions in this section, we work in a polyadic calculus. The extension of our type system to handle polyadicity is rather standard, and brings no particular difficulty.

3.1 A More Flexible Handling of Levels

Our system is strictly more expressive than the original one by Deng and Sangiorgi [5], as expressed by the two following observations (Lemma 14 and Example 15):

Lemma 14 *Any process typable according to the first type system of [5] is typable in our system.*

Proof *The presentation of [5] differs slightly from ours. The first system presented in that paper can be recast in our setting by working with the # capability only (thus disallowing subtyping), and requiring type $\sharp^k T$ for a in the first premise of the rules for output, finite input and replicated input. We write $\Gamma \vdash_D P : w$ for the resulting judgement. We establish that $\Gamma \vdash_D P : w$ implies $\Gamma \vdash P : w$ by induction over the derivation of $\Gamma \vdash_D P : w$.* □

We now present an example showing that the flexibility brought by subtyping can be useful to ease programming. We view replicated processes as servers, or functions. Our example shows that it is possible in our system to invoke a server by passing names having different levels, provided some form of coherence (as expressed by the subtyping relation) is guaranteed. This form of "polymorphism on levels" is not available in previous type systems for termination in the π-calculus.

Example 15 (Level-polymorphism) *Consider the following definitions (in addition to polyadicity, we accommodate the first-order type of natural numbers, with corresponding primitive operations):*

$$\begin{array}{rcl} F_1 & = & !f_1(n,r).\overline{r}\langle n*n\rangle \\ F_2 & = & !f_2(m,r).(\boldsymbol{\nu} s)\left(\overline{f}_1\langle m+1,s\rangle \mid s(x).\overline{r}\langle x+1\rangle\right) \\ Q & = & !g(p,x,r).(\boldsymbol{\nu} s)\left(\overline{p}\langle x,s\rangle \mid s(y).\overline{p}\langle y,r\rangle\right) \end{array}$$

F_1 is a server, running at f_1, that returns the square of a integer on a continuation channel r (which is its second argument). F_2 is a server that computes similarly $(m+1)^2+1$, by making a call to F_1 to compute $(m+1)^2$. Both F_1 and F_2 can be viewed as implementations of functions of type `int -> int`.

Q is a "higher-order server": its first argument p is the address of a server acting as a function of type `int -> int`*, and Q returns the result of calling twice the function located at p on its argument (process Q thus somehow acts like Church numeral 2).*

Let us now examine how we can typecheck the process

$$F_1 \mid F_2 \mid Q \mid \overline{g}\langle f_1, 4, t_1\rangle \mid \overline{g}\langle f_2, 5, t_2\rangle \ .$$

F_2 contains a call to f_1 under a replicated input on f_2, which forces $\mathsf{lv}(f_2) > \mathsf{lv}(f_1)$. In the type systems of [5], this prevents us from typing the processes above, since f_1 and f_2 should have the same type (and hence in particular the same level), both being used as argument in the outputs on g. We can type this process in our setting, thanks to subtyping, for instance by assigning the following types: $g : \mathsf{o}^{k_g}(\mathsf{o}^{k_2}T, U, V)$, $f_2 : \sharp^{k_2}T$, $f_1 : \sharp^{k_1}T$, with $k_1 < k_2$.

It can be shown that this example cannot be typed using any of the systems of [5]. It can however be phrased (and hence recognised as terminating) in the "purely functional π-calculus" of [4], that is, using a semantics-based approach — see also Section 3.3. It should however not be difficult to present a variation on it that forces one to rely on levels-based type systems.

3.2 Encoding the Simply-Typed λ-calculus

We now push further the investigation of the ability to analyse terminating functional behaviour in the π-calculus using our type system, and study an encoding of the λ-calculus in the π-calculus.

We focus on the following *parallel call-by-value* encoding, but we believe that the analogue of the results we present here also holds for other encodings. A λ-term M is encoded as $[\![M]\!]_p$, where p is a name which acts as a parameter in the encoding. The encoding is defined as follows:

$$[\![\lambda x.M]\!]_p \stackrel{def}{=} (\boldsymbol{\nu} y)\,(!y(x,q).[\![M]\!]_q \mid \overline{p}\langle y\rangle) \qquad\qquad [\![x]\!]_p \stackrel{def}{=} \overline{p}\langle x\rangle$$

$$[\![MN]\!]_p \stackrel{def}{=} (\boldsymbol{\nu} q, r)\,\big([\![M]\!]_q \mid [\![N]\!]_r \mid q(f).r(z).\overline{f}\langle z, p\rangle\big)$$

We can make the following remarks:

- A simply-typed λ-term is encoded into a simply-typed process (see [13]). Typability for termination comes into play in the translation of λ-abstractions.
- The target of this encoding is $L\pi$, the *localised π-calculus* in which only the output capability is transmitted (see also Section 4.1).

[3] provides a counterexample to typability of this encoding for the first type system of [5] (the proof of this result also entails that typability according to the other, more expressive, type systems due to Deng and Sangiorgi also fails to hold). Let us analyse this example:

Example 16 (From [3]) *The λ-term $M_1 \stackrel{def}{=} f\ (\lambda x.(f\ u\ (u\ v))$ can be typed in the simply typed λ-calculus, in a typing context containing the hypotheses $f : (\sigma \longrightarrow \tau) \longrightarrow \tau \longrightarrow \tau, v : \sigma, u : \sigma \longrightarrow \tau$.*

Computing $[\![M_1]\!]_p$ yields the process:

$$
\begin{array}{l}
(\boldsymbol{\nu} q, r) \\
\quad (\boldsymbol{\nu} y)\,\big(\, \overline{r}\langle y\rangle \\
\qquad \mid\; !y(x,q').(\boldsymbol{\nu} q_1, r_1, q_2, r_2, q_3, r_3) \\
\qquad\quad \big(\; \overline{q}_2\langle f\rangle \mid \overline{r}_2\langle u\rangle \mid q_2(f_2).r_2(z_2).\overline{f}_2\langle z_2, q_1\rangle \qquad [\![f\ u]\!]_{q_1} \\
\qquad\quad \mid\; \overline{q}_3\langle u\rangle \mid \overline{r}_3\langle v\rangle \mid q_3(f_3).r_3(z_3).\overline{f}_3\langle z_3, r_1\rangle \qquad [\![u\ v]\!]_{r_1} \\
\qquad\quad \mid\; q_1(f_1).r_1(z_1).\overline{f}_1\langle z_1, q'\rangle \;\big)\,\big) \\
\quad \mid \overline{q}\langle f\rangle \mid q(f').r(z).\overline{f}'\langle z, p\rangle
\end{array}
\qquad \Big]\ [\![\lambda x.\,(f\ u\ (u\ v))]\!]_r
$$

If we try and type this term using the first type system of [5], we can reason as follows:

1. *By looking at the line corresponding to $[\![f\ u]\!]_{q_1}$, we deduce that the types of f and f_2 are unified, and similarly for z_2 and u.*
2. *Similarly, the next line ($[\![u\ v]\!]_{r_1}$) implies that the types of f_3 and u are unified.*
3. *The last line above entails that the types assigned to f and f' must be unified, and the same for the types of z and y (because of the output $\overline{r}\langle y\rangle$).*

If we write $\sharp^k\langle T_1, T_2\rangle$ for the type (simple) assigned to f, we have by remark 1 that u has type T_1, and the same holds for y by remark 3. In order to typecheck the replicated term, we must have $\mathsf{lvl}(y) > \mathsf{lvl}(f_3) = \mathsf{lvl}(u)$ by remark 2, which is impossible since y and u have the same type.

While $[\![M_1]\!]_p$ cannot be typed using the approach of [5], it can be using the system of Section 2. Indeed, in that setting y and u need not have the same levels, so that we can satisfy the constraint $\mathsf{lvl}(y) > \mathsf{lvl}(u)$. The last line above generates an output $\overline{f}\langle y, p\rangle$, which can be typed directly, without use of subtyping. To typecheck the output $\overline{f}\langle u, q_1\rangle$, we "promote" the level of u to the level of y thanks to subtyping, which is possible because only the output capability on u is transmitted along f.

It however appears that our system is not able to typecheck the image of STλ, as the following (new) counterexample shows:

Example 17 *We first look at the following rather simple π-calculus process:*

$$(\boldsymbol{\nu} u)\,\big(\, !u(x).\overline{x} \mid (\boldsymbol{\nu} v)\,(!v.\overline{u}\langle t\rangle \mid \overline{u}\langle v\rangle)\,\big)\ .$$

This process is not typable in our type system, although it terminates. Indeed, we can assign a type of the form $\sharp^k \mathsf{o}^n \mathbb{U}$ to u, and $\sharp^m \mathbb{U}$ to v. Type-checking the subterm $!v.\overline{u}\langle t\rangle$ imposes $k < m$, and type-checking $!u(x).\overline{x}$ imposes $k > n$. Finally, type-checking $\overline{u}\langle v\rangle$ gives $m \leq n$, which leads to an inconsistency.

We can somehow 'expand' this process into the encoding of a λ-term: consider indeed

$$M_2 \ \stackrel{def}{=}\ \big(\lambda u.\,((\lambda v.(u\ v))\ (\lambda y.(u\ t)))\big)\ \ (\lambda x.\,(x\ a))\ .$$

We do not present the (rather complex) process corresponding to $[\![M_2]\!]_p$. We instead remark that there is a sequence of reductions starting from $[\![M_2]\!]_p$ and leading to

$$!y_1(u,q_1).\big(\; !y_3(v,q_4).\overline{u}\langle v,q_4\rangle \mid !y_5(y,q_5).\overline{u}\langle t,q_5\rangle \mid \overline{y}_3\langle y_5,q_1\rangle\;\big) \;\mid\; !y_2(x,q_2).\overline{x}\langle a,q_2\rangle \;\mid\; \overline{y}_1\langle y_2,p\rangle\ .$$

These first reduction steps correspond to 'administrative reductions' (which have no counterpart in the original λ-calculus term). We can now perform some communications that correspond to β-reductions, and obtain a process which contains a subterm of the form

$$\overline{\mathbf{u}}\langle \mathbf{v}, p\rangle \mid !\mathbf{v}(y,q_5).\overline{\mathbf{u}}\langle \mathbf{t}, q_5\rangle \mid !\mathbf{u}(\mathbf{x},q_2).\overline{\mathbf{x}}\langle a, q_2\rangle\ .$$

Some channel names appear in boldface in order to stress the similarity with the process seen above: for the same reasons, this term cannot be typed. By subject reduction (Theorem 7), a typable term can only reduce to a typable term. This allows us to conclude that $[\![M_2]\!]_p$ is not typable in our system.

$$\frac{\Gamma, x : T \bullet - \vdash P : w \qquad k \geq w}{\Gamma \bullet f : \mathsf{o}^k T \vdash !f(x).P : 0} \qquad \frac{\Gamma, f : \mathsf{o}^k T \vdash a : \mathsf{o}^n U \qquad \Gamma, f : \mathsf{o}^k T \vdash v : U}{\Gamma \bullet f : \mathsf{o}^k T \vdash \overline{a}\langle v \rangle : n}$$

$$\frac{\Gamma \vdash c : \mathsf{i}^n T \qquad \Gamma, x : T, f : \mathsf{o}^k U \bullet - \vdash P : w \qquad n > w}{\Gamma \bullet f : \mathsf{o}^k U \vdash c(x).P : 0}$$

$$\frac{\Gamma \vdash c : \mathsf{i}^n T \qquad \Gamma, x : T, f : \mathsf{o}^k U \bullet - \vdash P : w \qquad n > w}{\Gamma \bullet f : \mathsf{o}^k U \vdash !c(x).P : 0} \qquad \frac{\Gamma \bullet f : \mathsf{o}^k T \vdash P_1 \qquad \Gamma \bullet f : \mathsf{o}^k T \vdash P_2}{\Gamma \bullet f : \mathsf{o}^k T \vdash P_1 | P_2}$$

$$\frac{\Gamma, g : \mathsf{o}^k T \bullet f : \mathsf{o}^n U \vdash P : w}{\Gamma \bullet g : \mathsf{o}^k T \vdash (\boldsymbol{\nu} f) P : w} \qquad \frac{\Gamma, c : \sharp^n T \bullet f : \mathsf{o}^k U \vdash P : w}{\Gamma \bullet f : \mathsf{o}^k U \vdash (\boldsymbol{\nu} c) P : w}$$

Figure 3: Typing Rules for an Impure Calculus

3.3 Subtyping and Functional Names

In order to handle functional computation as expressed by STλ, we extend the system of Section 2 along the lines of [4]. The idea is to classify names into *functional* and *imperative* names. Intuitively, functional names arise through the encoding of STλ. For termination, these are dealt with using an appropriate method — the 'semantics-based' approaches discussed in Section 1, and introduced in [15, 12]. For imperative names, we resort to (an adaptation of) the rules of Section 2.

Our type system is à la Curry, and the kind of a name, functional or imperative, is fixed along the construction of a typing derivation. Typing environments are of the form $\Gamma \bullet f : \mathsf{o}^k T$ — the intuition is that we isolate a particular name, f. f is the name which can be used to build replicated inputs where f is treated as a functional name. The typing rules are given on Figure 3. There are two rules to typecheck a restricted process, according to whether we want to treat the restricted name as functional (in which case the isolated name changes) or imperative (in which case the typing hypothesis is added to the Γ part of the typing environment).

The typing rules of Figure 3 rely on i/o-capabilities and the isolated name to enforce the usage of functional names as expressed in [12]. In [4], a specific syntactical construct is instead used: we manipulate processes of the form $\texttt{def}\ f\ =\ (x)P_1\ \texttt{in}\ P_2$ (that can be read as $(\boldsymbol{\nu} f)\,(!f(x).P_1 \,|\, P_2)$), where f does not occur in P_1 and occurs in output position only in P_2.

Let us analyse how our system imposes these constraints. In the rule for restriction on a functional name, the name g, that occurs in 'isolated position' in the conclusion of the rule, is added in the 'non isolated' part of the typing environment in the premise, with a type allowing one to use it in output only.

In the rules for input on an imperative name (replicated or not), the typing environment is of the form $\Gamma \bullet -$ in the premise where we typecheck the continuation process: this has to be understood as $\Gamma \bullet d : \mathsf{o}^k T$, for some dummy name d that is not used in the process being typed. We write '$-$' to stress the fact that we disallow the construction of replicated inputs on functional names. The functional name f appears in the aforementioned premise in the 'non isolated' part of the typing environment, with only the output rights on it. Forbidding the creation of replicated inputs on functional names under input

prefixes is necessary because of diverging terms like the following (c is imperative, f is functional):

$$c(x).!f(y).\overline{x}\langle y\rangle \;\mid \overline{c}\langle f\rangle \mid \overline{f}\langle v\rangle \;\;.$$

Note also that typing non replicated inputs (on imperative names) involves the same constraints as for replicated inputs, like in [4]: the relaxed control over functional names requires indeed to be more restrictive on all usages of imperative names.

The notation $\Gamma \bullet -$ is also used in the rule to type a replicated input on a functional name, and we can notice that in this case f cannot be used at all in the premise, to avoid recursion.

In addition to the gain in expressiveness brought by subtyping, we can make the following remark:

Remark 18 (Expressiveness) *As in [4], our system allows one to typecheck the encoding of a STλ term, by treating all names as functional, and assigning them level* 0*.*

Moreover, our type system makes it possible to typecheck processes where several replicated inputs on the same functional name coexist, provided they occur 'at the same level' in the term. For instance, a term of the form $(\boldsymbol{\nu} f)\,(!f(x).P \mid !f(y).Q \mid R)$ *can be well-typed with* f *acting as a functional name. This is not possible using the* `def` *construct of [4].*

Another form of expressiveness brought by our system is given by typability of the following process: $!u(x).\overline{x} \mid !v.\overline{u}\langle t\rangle \mid \overline{u}\langle v\rangle \mid c(y).\overline{u}\langle c\rangle$. Here, name c must be imperative while name v must be functional, and both are emitted on u. This is impossible in [4], where every channel carries either a functional or an imperative name. In our setting, only the output capability on c is transmitted along u, so in a sense c is transmitted 'as a functional name'.

Because of the particular handling of restrictions on functional names, the analogue of Lemma 5 does not hold for this type system: typability is not preserved by structural congruence. Accordingly, the subject reduction property is stated in the following way:

Theorem 19 (Subject reduction) *If* $\Gamma \bullet f : \mathsf{o}^k T \vdash P : w$ *and* $P \longrightarrow P'$*, then there exist* Q *and* $w' \leq w$ *s.t.* $P' \equiv Q$ *and* $\Gamma \bullet f : \mathsf{o}^k T \vdash Q : w'$*.*

Theorem 20 (Soundness) *If* $\Gamma \bullet f : \mathsf{o}^k T \vdash P : w$*, then* P *terminates.*

Proof (sketch). *The proof has the same structure as the corresponding proof in [4]. An important aspect of that proof is that we exploit the termination property for the calculus where all names are functional without looking into it. To handle the imperative part, we must adapt the proof along the lines of the termination argument for Theorem 12.*

4 Type Inference

We now study type inference, that is, given a process P, the existence of Γ, w such that $\Gamma \vdash P : w$. There might a priori be several such Γ (and several w: see Lemma 3). Type inference for level-based systems has been studied in [2], in absence of i/o-types. We first present a type inference procedure in a special case of our type system, and then discuss this question in the general case.

4.1 Type Inference for Termination in the Localised π-calculus

In this section, we concentrate on the *localised π-calculus*, Lπ, which is defined by imposing that channels transmit only the output capability on names: a process like $a(x).x(y).\mathbf{0}$ does not belong to Lπ, as it makes use of the input capability on x. From the point of view of implementations, the restriction to Lπ

makes sense. For instance, the language JoCaml [10] implements a variant of the π-calculus that follows this approach: one can only use a received name in output. Similarly, the communication primitives in Erlang [9] can also be viewed as obeying to the discipline of Lπ: asynchronous messages can be sent to a PiD (process id), and one cannot create dynamically a receiving agent at that PiD: the code for the receiver starts running as soon as the PiD is allocated.

Technically, Lπ is introduced by allowing the transmission of o-types only. We write $\Gamma \vdash^{L\pi} P : w$ if $\Gamma \vdash P : w$ can be derived in such a way that in the derivation, whenever a type of the form $\eta^k \eta'^{k'} T$ occurs, we have $\eta' = \mathsf{o}$ (types of the form $\mathsf{i}^k T$ and $\sharp^k T$ appear only when typechecking input prefixes and restrictions). Obviously, typability for $\vdash^{L\pi}$ entails typability for $\vdash$, hence termination. It can also be remarked that in restricting to Lπ, we keep an important aspect of the flexibility brought by our system. In particular, the examples we have discussed in Section 3 — Example 15, and the encoding of the λ-calculus — belong to Lπ.

We now describe a type inference procedure for $\vdash^{L\pi}$. For lack of space, we do not provide all details and proofs.

We first check typability when levels are not taken into account. For this, we rely on a type inference algorithm for simple types [14], together with a simple syntactical check to verify that no received name is used in input. When this first step succeeds, we replace $\sharp T$ types with $\mathsf{o}T$ types appropriately in the outcome of the procedure for simple types (a type variable may be assigned to some names, as, e.g., to name x in process $a(x).\overline{b}\langle x\rangle$).

What remains to be done is to find out whether types can be decorated with levels in order to ensure termination. As mentioned above, we suppose w.l.o.g. that we have a term P in which all bound names are pairwise distinct, and distinct from all free names. We define the following sets of names:

- $\mathrm{names}(P)$ stands for the set of all names, free and bound, of P;
- $\mathrm{bn}(P)$ is the set of names that appear bound (either by restriction or by input) in P;
- $\mathrm{rcv}(P)$ is the set of names that are bound by an input prefix in P ($x \in \mathrm{rcv}(P)$ iff P has a subterm of the form $a(x).Q$ or $!a(x).Q$ for some a, Q);
- $\mathrm{res}(P)$ stands for the set of names that are restricted in P ($a \in \mathrm{res}(P)$ iff P has a subterm of the form $(\boldsymbol{\nu} a)\, Q$ for some Q).

We have $\mathrm{bn}(P) = \mathrm{rcv}(P) \uplus \mathrm{res}(P)$ (where $\uplus$ stands for disjoint union), and $\mathrm{names}(P) = \mathrm{bn}(P) \uplus \mathrm{fn}(P)$. Moreover, for any $x \in \mathrm{rcv}(P)$, there exists a unique $a \in \mathrm{fn}(P) \cup \mathrm{res}(P)$ such that P contains the prefix $a(x)$ or the prefix $!a(x)$: we write in this case $a = \mathrm{father}(x)$ ($a \in \mathrm{fn}(P) \cup \mathrm{res}(P)$, because we are in L$\pi$).

We build a graph as follows:

- For every name $n \in \mathrm{fn}(P) \cup \mathrm{res}(P)$, create a node labelled by n, and create a node labelled by $\mathrm{son}(n)$. Intuitively, if n has type $\sharp^k S$ of $\mathsf{o}^k S$, $\mathrm{son}(n)$ has type S. In case type inference for simple types returns a type of the form α, where α is a type variable, for n, we just create the node n.
- For every $x \in \mathrm{rcv}(P)$, let $a = \mathrm{father}(x)$, add x as a label to $\mathrm{son}(a)$.

Example 21 *We associate to the process* $P = a(x).(\boldsymbol{\nu} b)\,\overline{x}\langle b\rangle \mid !a(y).(\overline{c}\langle y\rangle \mid d(z).\overline{y}\langle z\rangle)$ *the following set of 8 nodes with their labels:* $\{a\}, \{\mathrm{son}(a), x, y\}, \{b\}, \{\mathrm{son}(b)\}, \{c\}, \{\mathrm{son}(c), y\}, \{d\}, \{\mathrm{son}(d), z\}$.

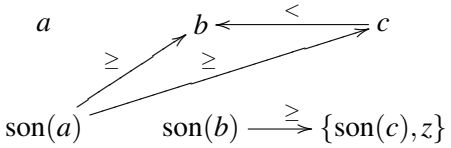
The next step is to insert edges in our graph, to represent the constraints between levels.

- For every output of the form $\overline{n}\langle m\rangle$, we insert an edge labelled with "$\geq$" from $\mathrm{son}(n)$ to m.
- For every subterm of P of the form $!a(x).Q$, and for every output of the form $\overline{n}\langle m\rangle$ that occurs in Q without occurring under a replication in Q, we insert an edge $a \xrightarrow{\geq} n$.

Example 22 *The graph associated to process* $!c(z).\overline{b}\langle z\rangle \mid \overline{a}\langle c\rangle \mid \overline{a}\langle b\rangle$ *has nodes*

$$\{a\},\{\mathrm{son}(a)\},\{b\},\{\mathrm{son}(b)\},\{c\},\{\mathrm{son}(c),z\} \ ,$$

and can be depicted as follows:



The last phase of the type inference procedure consists in looking for an assignment of levels on the graph: this is possible as long as there are no cycles involving at least one $\stackrel{>}{\to}$ edge in the graph.

At the beginning, all nodes of the graph are unlabelled; we shall label them using natural numbers.

1. We go through all nodes of the graph, and collect those that have no outgoing edge leading to an unlabelled node in a set $\mathscr{S}$.
2. If $\mathscr{S}$ is not empty, we label every node n in $\mathscr{S}$ as follows: we start by setting n's label to 0.

   We then examine all outgoing edges of n. For every $n \stackrel{\geq}{\to} m$, we replace n's label, say k, with $\max(k,k')$, where k' is m's label, and similarly for $n \stackrel{>}{\to} m$ edges, with $\max(k,k'+1)$.

   We then empty $\mathscr{S}$, and start again at step 1.
3. If $\mathscr{S} = \emptyset$, then either all nodes of the graph are labelled, in which case the procedure terminates, or the graph contains at least one oriented cycle. If this cycle contains at least one $\stackrel{>}{\to}$ edge, the procedure stops and reports failure. Otherwise, the cycle involves only $\stackrel{\geq}{\to}$ edges: we compute the level of each node of the cycle along the lines of step 2 (not taking into account nodes of the cycle among outgoing edges), and then assign the maximum of these labels to all nodes in the cycle. We start again at step 1.

This procedure terminates, since each time we go back to step 1, strictly more nodes are labelled.

Example 23 *On the graph of Example 22, the procedure first assigns level* 0 *to nodes* a,b *and* $\{\mathrm{son}(c),z\}$. *In the second iteration,* $\mathscr{S} = \{\mathrm{son}(b),c\}$*; level* 0 *is assigned to* $\mathrm{son}(b)$*, and* 1 *to* c*. Finally, level* 1 *is assigned to* $\mathrm{son}(a)$*. This yields the typing* $b : \mathsf{o}^0\mathsf{o}^0T, c : \sharp^1\mathsf{o}^0T, a : \mathsf{o}^0\mathsf{o}^1\mathsf{o}^0T$ *for the process of Example 22.*

As announced above, for lack of space we have described only the main steps of our type inference procedure. Establishing that the latter has the desired properties involves the introduction of an auxiliary typing judgement (that characterises $\vdash^{L\pi}$), and explaining how types are reconstructed at the end of the procedure. This finally leads to the following result:

Theorem 24 *There is a type inference procedure that given a process* P*, returns* Γ, w *s.t.* $\Gamma \vdash^{L\pi} P : w$ *iff there exists* Γ', w' *s.t.* $\Gamma' \vdash^{L\pi} P : w'$.

4.2 Discussion: Inferring i/o-Types

If we consider type inference for the whole system of Section 2, the situation is more complex. We start by discussing type inference without taking the levels into account. If a process is typable using simple types (that is, with only types of the form $\sharp T$), one is interested in providing a more informative typing derivation, where input and output capabilities are used.

For instance, the process $a(x).\overline{x}\langle t\rangle$ can be typed using different assignments for a: $\mathsf{io}T$, $\sharp\mathsf{o}T$, $\mathsf{i}\sharp T$, and $\sharp\sharp T$ — if we suppose $t : T$. Among these, $\mathsf{io}T$ is the most informative (intuitively, types featuring

'less #' seem preferable because they are more precise). Moreover, it is a supertype of all other types, thus acting as a 'candidate' if we were to look for a notion of principal typing. Actually, in order to infer i/o-types, one must be able to compute lubs and glbs of types, using equations like $glb(\mathsf{i}T, \mathsf{i}U) = \mathsf{i}\, glb(T,U)$, $glb(\mathsf{i}T, \mathsf{o}U) = \sharp glb(T,U)$, and $glb(\mathsf{o}T, \mathsf{o}U) = \mathsf{o}\, lub(T,U)$. The contravariance of o suggests the introduction of an additional capability, that we shall note $\uparrow$, which builds a supertype of input and output capabilities (more formally, we add the axioms $\mathsf{i}T \leq \uparrow T$ and $\mathsf{o}T \leq \uparrow T$).

[7] presents a type inference algorithm for (an enrichment of) i/o-types, where such a capability $\uparrow$ is added to the system of [11] (the notations are different, but we adapt them to our setting for the sake of readability). The use of $\uparrow$ can be illustrated on the following example process:

$$Q_1 \stackrel{def}{=} a(t).b(u).\big(\ !t(z).\overline{u}\langle z\rangle \mid \overline{c}\langle t\rangle \mid \overline{c}\langle u\rangle\ \big)\ .$$

To typecheck Q_1, we can see that the input (resp. output) capability on t (resp. u) needs to be received on a (resp. b), which suggests the types $a : \mathsf{i}\mathsf{i}T, b : \mathsf{i}\mathsf{o}T$. Since t and u are emitted on the same channel c, and because of contravariance of output, we compute a *supertype* of $\mathsf{i}T$ and $\mathsf{o}T$, and assign type $\mathsf{o} \uparrow T$ to c.

Operationally, the meaning of $\uparrow$ is "no i/o-capability at all" (note that this does not prevent from comparing names, which may be useful to study behavioural equivalences [6]): in the typing we just described, since we only have the input capability on t and the output capability on u, we must renounce to all capabilities, and t and u are sent without the receiver to be able to do anything with the name except passing it along. Observe also that depending on how the context uses c, a different typing can be introduced. For instance, Q_1 can be typed by setting $a : \mathsf{i}\sharp T, b : \mathsf{i}\mathsf{o}T, c : \mathsf{o}\mathsf{o}T$. This typing means that the output capability on u is received, used, and transmitted on c, and both capabilities on t are received, the input capability being used locally, while the output capability is transmitted on c.

The first typing, which involves $\uparrow$, is the one that is computed by the procedure of [7]. It is "minimal", in the terminology of [7]. Depending on the situations, a typing like the second one (or the symmetrical case, where the input capability is transmitted on c) might be preferable.

If we take levels into account, and try and typecheck Q_1 (which contains a replicated subterm), the typings mentioned above can be adapted as follows: we can set $a : \mathsf{i}^0\sharp^1 T, b : \mathsf{i}^0\mathsf{o}^0 T, c : \mathsf{o}^0\mathsf{o}^1 T$, in which case subtyping on levels is used to deduce $u : \mathsf{o}^1 T$ in order to typecheck $\overline{c}\langle u\rangle$. Symmetrically, we can also set $a : \mathsf{i}^0\mathsf{i}^1 T, b : \mathsf{i}^0\sharp^0 T, c : \mathsf{o}^0\mathsf{i}^0 T$, and typecheck $\overline{c}\langle t\rangle$ using subsumption to deduce $t : \mathsf{i}^0 T$.

It is not clear to us how levels should be handled in relation with the $\uparrow$ capability. One could think that since $\uparrow$ prevents any capability to be used on a name, levels have no use, and one could simply adopt the subtyping axioms $\mathsf{i}^k T \leq \uparrow T$ and $\mathsf{o}^k T \leq \uparrow T$. This would indeed allow us to typecheck Q_1.

Further investigations on a system for i/o-types with $\uparrow$ and levels is left for future work, as well as the study of inference for such a system.

5 Concluding Remarks

In this paper, we have demonstrated how Pierce and Sangiorgi's i/o-types can be exploited to refine the analysis of the simplest of type systems for termination of processes presented in [5]. Other, more complex systems are presented in that work, and it would be interesting to study whether they would benefit from the enrichment with capabilities and subtyping. One could also probably refine the system of Section 2 by distinguishing between *linear* and *replicated input capabilities*, as only the latter must be controlled for termination (if a name is used in linear input only, its level is irrelevant).

The question of type inference for our type systems (differently from existing proposals, these are presented à la Curry, which is better suited for the study of type inference) can be studied further. It would

be interesting to analyse how the procedure of Section 4.1 could be ported to programming languages that obey the discipline of Lπ for communication, like Erlang or JoCaml. For the moment, we only have preliminary results for a type inference procedure for the system of Section 2, and we would like to explore this further. Type inference for the system of Section 3.3 is a challenging question, essentially because making the distinction between functional and imperative names belongs to the inference process (contrarily to the setting of [4], where the syntax of processes contains this information).

Acknowledgements. Romain Demangeon, as well as anonymous referees, have provided insightful comments and suggestions on this work. We also acknowledge support by ANR projects ANR-08-BLANC-0211-01 "COMPLICE", ANR-2010-BLANC-0305-02 "PiCoq" and CNRS PEPS "COGIP".

References

[1] R. Demangeon (2010): *Terminaison des systèmes concurrents*. Ph.D. thesis, ENS Lyon.

[2] R. Demangeon, D. Hirschkoff, N. Kobayashi & D. Sangiorgi (2007): *On the Complexity of Termination Inference for Processes*. In: *Proc. of TGC'07*, *LNCS* 4912, Springer, pp. 140–155, doi:10.1007/978-3-540-78663-4_11.

[3] R. Demangeon, D. Hirschkoff & D. Sangiorgi (2009): *Mobile Processes and Termination*. In: *Semantics and Algebraic Specification*, *LNCS* 5700, Springer, pp. 250–273, doi:10.1007/978-3-642-04164-8_13.

[4] R. Demangeon, D. Hirschkoff & D. Sangiorgi (2010): *Termination in Impure Concurrent Languages*. In: *Proc. of CONCUR'10*, *LNCS* 6269, Springer, pp. 328–342, doi:10.1007/978-3-642-15375-4_23.

[5] Y. Deng & D. Sangiorgi (2006): *Ensuring termination by typability*. *Inf. Comput.* 204(7), pp. 1045–1082, doi:10.1016/j.ic.2006.03.002.

[6] M. Hennessy & J. Rathke (2004): *Typed behavioural equivalences for processes in the presence of subtyping*. *Math. Str. in Comp. Sc.* 14(5), pp. 651–684, doi:10.1017/S0960129504004281.

[7] A. Igarashi & N. Kobayashi (2000): *Type Reconstruction for Linear -Calculus with I/O Subtyping*. *Inf. Comput.* 161(1), pp. 1–44, doi:10.1006/inco.2000.2872.

[8] N. Kobayashi & D. Sangiorgi (2010): *A hybrid type system for lock-freedom of mobile processes*. *ACM Trans. Program. Lang. Syst.* 32(5), doi:10.1145/1745312.1745313.

[9] Ericsson Computer Science Laboratory (2011): *Erlang Programming Language Website*. `http://www.erlang.org`.

[10] L. Mandel & L. Maranget (2010): *The JoCaml programming language*. `http://jocaml.inria.fr/`.

[11] B. C. Pierce & D. Sangiorgi (1996): *Typing and Subtyping for Mobile Processes*. *Math. Structures in Comput. Sci.* 6(5), pp. 409–453.

[12] D. Sangiorgi (2006): *Termination of Processes*. *Math. Structures in Comput. Sci* 16(1), pp. 1–39, doi:10.1017/S0960129505004810.

[13] D. Sangiorgi & D. Walker (2001): *The π-calculus: a Theory of Mobile Processes*. Cambridge Univ. Press.

[14] V. T. Vasconcelos & K. Honda (1993): *Principal Typing Schemes in a Polyadic pi-Calculus*. In: *Proc. of CONCUR'93*, *Lecture Notes in Computer Science* 715, Springer, pp. 524–538, doi:10.1007/3-540-57208-2_36.

[15] N. Yoshida, M. Berger & K. Honda (2004): *Strong Normalisation in the Pi-Calculus*. *Information and Computation* 191(2), pp. 145–202, doi:10.1016/j.ic.2003.08.004.