Math. Struct. in Comp. Science (2016), *vol.* 26, *pp.* 1395–1432.
doi:10.1017/S0960129514000620 First published online 9 January 2015

Termination in a π-calculus with subtyping

IOANA CRISTESCU[†] and DANIEL HIRSCHKOFF[‡]

[†]*Laboratoire PPS, Université Paris Diderot and CNRS, UMR 7126, F-75205 Paris, France*
[‡]*École Normale Supérieure de Lyon, CNRS, INRIA, UCBL, U. Lyon, France*
Email: daniel.hirschkoff@ens-lyon.fr

Received 1 January 2012; revised 1 February 2013

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.

Our type system is based on Deng and Sangiorgi's level-based analysis of processes. We show that the addition of i/o-types makes it possible to typecheck processes where a form of *level polymorphism* is at work. We discuss to what extent this programming idiom can be treated by previously existing proposals. 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 (Kobayashi and Sangiorgi 2010).

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 (Demangeon *et al.* 2009, 2010; Deng and Sangiorgi 2006; Sangiorgi 2006; Yoshida *et al.* 2004). 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 recognized as terminating. A type system for termination typically rules out some terminating terms, because it is not able to recognize them as such (by essence, an effective type system for termination defines an approximation of

this undecidable property). Improving expressiveness means 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.

1.1. *Type systems for termination in the π-calculus*

Existing type systems for termination in the π-calculus build on simple types (Sangiorgi and Walker 2001), whereby the type of a channel describes the 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 mostly discuss here 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 originate in Deng and Sangiorgi (2006), and have been further analysed and developed in Demangeon *et al.* (2009). 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. Like in previous work, adding features like synchrony or the sum operator to our setting does not bring any difficulty, as we discuss in Section 2.

According to level-based type systems, the process $!a(x).\overline{b}\langle x\rangle$ is well-typed provided $\mathrm{lvl}_\Gamma(a)$, the level of a, is strictly greater than $\mathrm{lvl}_\Gamma(b)$ (Γ is the typing environment, associating types to names). Intuitively, this process trades messages on a (that 'cost' $\mathrm{lvl}_\Gamma(a)$) 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 under the same hypotheses, 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 $\mathrm{lvl}_\Gamma(a)$ (more accurately, we only take into account those messages 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 $\mathrm{lvl}_\Gamma(a) > \mathrm{lvl}_\Gamma(b) > \mathrm{lvl}_\Gamma(a)$), 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.

1.2. *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 according to 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 $\mathrm{lvl}_\Gamma(a) > \mathrm{lvl}_\Gamma(x)$, 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 $\mathrm{lvl}_\Gamma(p) < \mathrm{lvl}_\Gamma(a)$ and $\mathrm{lvl}_\Gamma(q) < \mathrm{lvl}_\Gamma(a)$. 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 $\mathrm{lvl}_\Gamma(p) > \mathrm{lvl}_\Gamma(q)$, then the resulting process is rejected, although it is terminating.

Our goal is to provide more flexibility in the handling of the levels of names, by relaxing the constraint that p and q from the example above should have exactly 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 $\mathrm{lvl}_\Gamma(a)$. 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 $\mathrm{lvl}_\Gamma(y) > \mathrm{lvl}_\Gamma(c)$, which means that any name of level *strictly greater* than $\mathrm{lvl}_\Gamma(c)$ 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).

1.3. *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) (Pierce and Sangiorgi 1996). 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 viewed as a channel where only one is used. Intuitively, being able to have a more precise description of how a name is used can help in asserting termination of a process: P_0 uses only the output capability on x, 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, since only the input capability on y needs to be transmitted.

The overall setting of this work is presented in Section 2, together with the definition of our type system. As expected, our system is strictly more expressive than the original, level-based 'core' type system of Deng and Sangiorgi (2006) – we compare our system with the more refined approaches introduced in Deng and Sangiorgi (2006) in Section 3.3. We show (Section 3.1) 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 (Section 3.2). Demangeon *et al.* (2009) provides a counterexample showing that existing

level-based approaches are not able to recognize as terminating the image of the simply-typed λ-calculus (STλ in short) in the π-calculus – it is known that all processes computed using such an encoding terminate (Sangiorgi and Walker 2001). We prove 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 Demangeon *et al.* (2010), where an *impure* π-calculus is studied. Here, impure means 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 Demangeon *et al.* (2010), termination is guaranteed in an impure π-calculus by using a level-based approach for imperative names, while functional names are dealt separately, using a semantics-based approach (Sangiorgi 2006; Yoshida *et al.* 2004). We show that type system, which combines both approaches for termination in the π-calculus, can be revisited in our setting (Section 4). We also demonstrate that the resulting system improves in terms of expressiveness over (Demangeon *et al.* 2010) from several points of view.

Several technical aspects in the definition of our type systems are new with respect to previous approaches. 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, our core type system, described in Section 2, is defined à la Curry, in order to follow the approach for i/o-types in Pierce and Sangiorgi (1996). As we discuss below, this has some consequences on the soundness proof of our systems. Another technical novelty is in the study of the impure calculus (Section 4). We indeed introduce two type systems. The first one (Section 4.1) follows (Demangeon *et al.* 2010): it is presented à la Church, and relies on a specific syntactical construction, akin to a `let .. in` construct, to handle functional names. We then give a second presentation (Section 4.2), which shows that by a refinement of i/o-types, we are able to enforce the discipline of functional names without resorting to a particular syntactical construct. This allows us to keep a uniform syntax, and to formulate the type system à la Curry, like it is the case in Pierce and Sangiorgi (1996) and for the system of Section 2. The soundness proof for the second type system exploits the soundness of the first one, and is presented in Section 4.2.2.

We finally discuss type inference (Section 5), 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 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.

1.4. *Paper outline*

Section 2 presents our type system, and shows that it guarantees termination. We study its expressiveness in Section 3, and extend it with the handling of functional names in Section 4. Section 5 discusses type inference, and we give concluding remarks in Section 6.

1.5. *On the contents of this paper*

Most of the results presented in this paper first appeared in Cristescu and Hirschkoff (2011). With respect to that version, we provide here more detailed explanations, as well as proofs, that were omitted in Cristescu and Hirschkoff (2011) due to lack of space. Section 4 has been considerably expanded (in particular, Sections 4.1 and 4.2.2 are new).

2. A type system for termination with subtyping

2.1. *Definition of the type system*

2.1.1. *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):

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

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 assume, in the remainder of the paper, that all the processes we manipulate abide a *Barendregt convention*, meaning that all bound names are pairwise distinct and are different from the free names. This may in particular involve some renaming of processes when a reduction is performed (we express this explicitly in Section 4.2.2).

The grammar of types is given by:

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

where k is a natural number that we call a *level*, and $\mathbb{U}$ stands for the `unit` type having $\star$ as the only value. A name having type $\sharp^k T$, $\mathsf{i}^k T$ or $\mathsf{o}^k T$ has level k. A name of type $\sharp^k T$ 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 use the symbol $\leqslant$ 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. $\leqslant$, 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 $a \notin \mathrm{dom}(\Gamma)$.

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

$$\textsc{Subt-\#I}\ \frac{}{\sharp^k T \leq \mathsf{i}^k T} \qquad \textsc{Subt-\#O}\ \frac{}{\sharp^k T \leq \mathsf{o}^k T} \qquad \textsc{Subt-II}\ \frac{T \leq S \qquad k_1 \leq k_2}{\mathsf{i}^{k_2} T \leq \mathsf{i}^{k_1} S} \qquad \textsc{Subt-OO}\ \frac{T \leq S \qquad k_1 \leq k_2}{\mathsf{o}^{k_1} S \leq \mathsf{o}^{k_2} T}$$

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

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

$$\textsc{Rep-S}\ \frac{\Gamma \vdash a : \mathsf{i}^k T \qquad k > w \qquad \Gamma, x : T \vdash P : w}{\Gamma \vdash\ !a(x).P : 0} \qquad \textsc{Res-S}\ \frac{\Gamma, a : T \vdash P : w}{\Gamma \vdash (\nu a)\, P : w}\ T \neq \mathbb{U} \qquad \textsc{Par-S}\ \frac{\Gamma \vdash P_1 : w_1 \qquad \Gamma \vdash P_2 : w_2}{\Gamma \vdash P_1 | P_2 : \max(w_1, w_2)}$$

Fig. 1. Typing and subtyping rules.

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.

Remark 2.1 (extending the calculus). We can consider various variants and enrichments of the π-calculus we focus on in this paper.

A first extension is given by moving to polyadic communication, which makes the presentation of the type system more involved, but brings no difficulty.

Adopting *synchronous outputs* would involve a minor change: typing $\overline{a}\langle v \rangle.P$ 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).

We could also introduce *choice* $(+)$, and type $P + Q$ like $P|Q$. This is indeed compatible with what rule Par-S specifies: the weight of a sum is the maximum of the weight of each summand. More refined analyses could also be made, along the lines of the most refined type systems introduced in Deng and Sangiorgi (2006), where the weight of a process is computed as a multiset. The measure introduced in Definition 2.12 illustrates the idea: while both components of a parallel composition contribute to the weight of a process, we only need to keep the maximum weight for a choice.

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

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

Fig. 2. Reduction of processes.

We may also consider adding *name matching* to our calculus, by considering for instance processes of the form $[x = y]P$ – such a process is triggered only if names x and y are equal. Arguably, to typecheck this process, one must be able to decide *at which type* names x and y are compared for equality. This might involve computing glbs and lubs of types, which, as we discuss in Section 5.2, is not always possible. The analysis of matching in presence of i/o-types deserves further work, possibly in the setting of Igarashi and Kobayashi (2000), where glbs and lubs of i/o-types can be computed.

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 2.2. The process $!a(x).\overline{x}\langle t\rangle \mid \overline{a}\langle p\rangle \mid \overline{a}\langle q\rangle \mid !p(z).\overline{q}\langle z\rangle$ from Section 1 can be typed in our type system: we can set $a : \sharp^3\mathsf{o}^2T, p : \sharp^2T, q : \mathsf{o}^1T$. Subtyping on levels is at work in order to typecheck the subterm $\overline{a}\langle q\rangle$ (to deduce, e.g. $q : \mathsf{o}^2T$). We provide a more complex term, which can be typed using similar ideas, in Example 3.2 below.

2.1.2. *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:

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

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

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

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

Definition 2.3 (termination). A process P *diverges* if there exists an infinite sequence of processes $(P_i)_{i\geqslant 0}$ 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 2.4. If $\Gamma \vdash P : w$ and $w \neq 0$ then for any $w' \geqslant w$, $\Gamma \vdash P : w'$.

Proof. We reason by induction on the typing derivation. Rules Nil-S and Rep-S give weight 0. The cases of rules Inp-S, Res-S and Par-S are handled easily by exploiting the induction hypotheses.

The interesting case corresponds to rule Out-S: we have $P = \overline{a}\langle b\rangle$, with $\Gamma \vdash a : \mathsf{o}^k T$ and $\Gamma \vdash b : T$. We can construct the following derivation, for any $k' \geqslant k$:

$$\dfrac{\dfrac{\Gamma \vdash a : \mathsf{o}^k T \qquad \dfrac{T \leqslant T \qquad k \leqslant k'}{\mathsf{o}^k T \leqslant \mathsf{o}^{k'} T}}{\Gamma \vdash a : \mathsf{o}^{k'} T} \qquad \Gamma \vdash b : T}{\Gamma \vdash \overline{a}\langle b\rangle : k'}.$$

□

Lemma 2.5 (strengthening). If $\Gamma, x : T \vdash P : w$ and $x \notin \mathrm{fn}(P)$, then $\Gamma \vdash P : w$.

Proof. Follows by induction on the typing derivation of P. □

Lemma 2.6 (weakening). If $\Gamma \vdash P : w$, then for all x and T, we have $\Gamma, x : T \vdash P : w$.

Proof. By induction on the typing derivation of P. □

Lemma 2.7. If $\Gamma \vdash P : w$, then $\mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma)$.

Proof. By induction on the structure of P. □

Proposition 2.8 (narrowing). If $\Gamma, x : T \vdash P : w$ and $T' \leqslant T$, then $\Gamma, x : T' \vdash P : w'$ for some $w' \leqslant w$.

Proof. We set $\Gamma_0 = \Gamma, x : T$ and $\Gamma'_0 = \Gamma, x : T'$, and make a first remark: if $\Gamma_0 \vdash n : T_1$ for some n and T_1, with $n \neq x$, then we have $\Gamma'_0 \vdash n : T_1$

We then reason by induction on the derivation of $\Gamma, x : T \vdash P : w$. We only show the interesting cases.

— Case **Out-S**. We have $P = \overline{a}\langle v\rangle$, $\Gamma_0 \vdash \overline{a}\langle v\rangle : k$ with $\Gamma_0 \vdash a : \mathsf{o}^k T_0$ and $\Gamma_0 \vdash v : T_0$. We distinguish several cases according to whether x is equal to a or v:
  - $x \neq a, v$. Follows by the above remark.
  - $x = a$ and $x \neq v$. In this case, $T \leqslant \mathsf{o}^w T_0$ and $\Gamma \vdash v : T_0$. The former yields $T' \leqslant \mathsf{o}^{w'} T_0$ for some $w' \leqslant w$. Thus, $\Gamma'_0 \vdash x : \mathsf{o}^{w'} T_0$, and $\Gamma'_0 \vdash \overline{x}\langle v\rangle : w'$ by applying Out-S.
  - $x = v$ and $x \neq a$. In this case $T_0 = T$. From $\Gamma_0 \vdash a : \mathsf{o}^w T$ and the remark above we have that $\Gamma'_0 \vdash a : \mathsf{o}^w T$. We can use subsumption to deduce $\Gamma'_0 \vdash b : T$ from $\Gamma'_0 \vdash b : T'$, which allows us to apply rule Out-S and deduce $\Gamma'_0 \vdash \overline{a}\langle x\rangle : w$.
  - $x = a = v$. Impossible because a and v's types have different shapes.

— Case **Rep-S**. We have $P = !a(y).P'$, with $\Gamma_0 \vdash a : \mathsf{i}^k T_0$, $\Gamma_0, y : T_0 \vdash P' : k'$ and $k > k'$. We reason as in the previous case, the only interesting situation being $a = x$. In that case, we deduce from $\Gamma'_0 \vdash a : T'$ that $\Gamma'_0 \vdash a : T$, and we get by induction $\Gamma'_0, y : T_0 \vdash P' : k''$ for some $k'' \leqslant k'$. We thus have $k'' < k$, and we can apply rule Rep-S to deduce $\Gamma'_0 \vdash !a(y).P' : 0$. □

Lemma 2.9. If $P \equiv Q$, then $\Gamma \vdash P : w$ iff $\Gamma \vdash Q : w$.

Proof. We reason by induction on the derivation of $P \equiv Q$. We only consider the case where $P \mid (\nu a)Q \equiv (\nu a)(P \mid Q)$. $P \mid (\nu a)Q$ is typed using rules Par-S and Res-S, from hypotheses $\Gamma \vdash P : w_1$ and $\Gamma, a : T \vdash Q : w_2$, with $w = \max(w_1, w_2)$. By applying Lemma 2.6 and rule Par-S, we derive $\Gamma, a : T \vdash (P \mid Q) : w$. Rule Res-S then yields $\Gamma \vdash (\nu a)(P \mid Q) : w$. Following a similar reasoning we can prove that if $\Gamma \vdash (\nu a)(P \mid Q) : w$ then $\Gamma \vdash P \mid (\nu a)Q : w$. □

Lemma 2.10. If $\Gamma, x : T \vdash P : w$, $\Gamma \vdash b : T'$ and $T' \leqslant T$, then $\Gamma \vdash P[b/x] : w'$, for some $w' \leqslant w$.

Proof. The proof follows the lines of the proof of Lemma 2.8. We show the case where $P = \overline{x}\langle v \rangle$. We have that $\Gamma, x : T \vdash \overline{x}\langle v \rangle : w$, hence $T \leqslant \mathsf{o}^w T_0$ for some T_0. Since $T' \leqslant T$, we have $T' \leqslant \mathsf{o}^{w'} T_0$ for some $w' \leqslant w$. This allows us to deduce $\Gamma \vdash b : \mathsf{o}^{w'} T_0$, and then $\Gamma \vdash \overline{b}\langle v \rangle : w'$. □

Theorem 2.11 (subject reduction). If $\Gamma \vdash P : w$ and $P \longrightarrow P'$, then $\Gamma \vdash P' : w'$ for some $w' \leqslant w$.

Proof. By induction over the derivation of $P \longrightarrow P'$. We only consider the following cases:

— $P \longrightarrow P'$ with $Q \longrightarrow Q'$, $P \equiv Q$ and $P' \equiv Q'$. Using induction we derive that $\Gamma \vdash Q : w_q$, $\Gamma \vdash Q' : w'_q$ and $w'_q \leqslant w_q$. From $P \equiv Q$ and Lemma 2.9 we obtain that $\Gamma \vdash P : w_q$. We apply the same reasoning on $P' \equiv Q'$ and therefore obtain $\Gamma \vdash P' : w'_q$ with $w'_q \leqslant w_q$.

— $P = a(x).P_1 \mid \overline{a}\langle v \rangle \longrightarrow P' = P_1[v/x]$.
$\Gamma \vdash P : w$ is derived from $\Gamma \vdash a(x).P_1 : w_1$ and $\Gamma \vdash \overline{a}\langle v \rangle : k$ with $w = \max(w_1, k)$. The former derivation comes from $\Gamma, x : T_x \vdash P_1 : w_1$ and $\Gamma \vdash a : \mathsf{i}^{k_i} T_x$. The latter gives $\Gamma \vdash a : \mathsf{o}^{k_0} T_v$ and $\Gamma \vdash v : T_v$. Thus, $\Gamma(a) = \sharp^k T_a$ with $T_v \leqslant T_a \leqslant T_x$ and $k_i \leqslant k \leqslant k_o$.
Lemma 2.10 finally gives $\Gamma \vdash P_1[v/x] : w'_1$, with $w'_1 \leqslant w_1 \leqslant \max(w_1, k) = w$.

— The case where $P = !a(x).P_1 \mid \overline{a}\langle v \rangle \longrightarrow P' = !a(x).P_1 \mid P_1[v/x]$ is treated similarly.

□

2.3. *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 $\Gamma \vdash P : w$ gives the *weight* w of process P, but this notion cannot be used as a measure to deduce termination (we have for instance $\overline{a}\langle v \rangle \mid \overline{a}\langle v \rangle \mid a(x).\mathbf{0} \longrightarrow \overline{a}\langle v \rangle$, and the weight is preserved). We adapt the approach of Demangeon (2010), and define the measure of a process as a multiset of natural numbers. This is done by induction over the derivation of a typing judgement for the process. We will use $\mathcal{D}$ to range over typing derivations, and write $\mathcal{D} : \Gamma \vdash P : w$ to mean that $\mathcal{D}$ 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 $>_{mul}$. Then $M_1 >_{mul} M_2$ 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 $e_2 \in N_2$ there is $e_1 \in N_1$ such that $e_1 > e_2$. The relation $>_{mul}$ is well-founded. We write $M_1 \geqslant_{mul} M_2$ if $M_1 >_{mul} M_2$ or $M_1 = M_2$. Relation $\geqslant_{mul}$ is total.

Definition 2.12. Suppose $\mathcal{D} : \Gamma \vdash P : w$. We define a multiset of natural numbers, noted $\mathcal{M}(\mathcal{D})$, by induction over $\mathcal{D}$ as follows:

— If $\mathcal{D} : \Gamma \vdash \mathbf{0}$ then $\mathcal{M}(\mathcal{D}) = \varnothing$;
— If $\mathcal{D} : \Gamma \vdash \overline{a}\langle b\rangle : k$ then $\mathcal{M}(\mathcal{D}) = \{\mathrm{lvl}_\Gamma(a)\}$;
— If $\mathcal{D} : \Gamma \vdash !a(x).P : 0$ then $\mathcal{M}(\mathcal{D}) = \varnothing$;
— If $\mathcal{D} : \Gamma \vdash a(x).P : w$, then $\mathcal{M}(\mathcal{D}) = \mathcal{M}(\mathcal{D}_1)$,
where $\mathcal{D}_1 : \Gamma, x : T \vdash P : w$ is the immediate subderivation of $\mathcal{D}$;
— If $\mathcal{D} : \Gamma \vdash (\nu a)\, P : w$, then $\mathcal{M}(\mathcal{D}) = \mathcal{M}(\mathcal{D}_1)$,
where $\mathcal{D}_1 : \Gamma, a : T \vdash P : w$ is the immediate subderivation of $\mathcal{D}$;
— If $\mathcal{D} : \Gamma \vdash P_1 | P_2 : \max(w_1, w_2)$, then $\mathcal{M}(\mathcal{D}) = \mathcal{M}(\mathcal{D}_1) \uplus \mathcal{M}(\mathcal{D}_2)$,
where $\mathcal{D}_i : \Gamma \vdash P_i : w_i$, $i = 1, 2$ are the immediate subderivations of $\mathcal{D}$.

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

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

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

Lemma 2.13. Suppose that we have a derivation $\mathcal{D} : \Gamma \vdash P_1 : w$, with $\Gamma(v) \leqslant \Gamma(x)$. Then there exists a derivation $\mathcal{D}' : \Gamma \vdash P_1[v/x] : w'$, and $\mathcal{M}(\mathcal{D}) \geqslant_{mul} \mathcal{M}(\mathcal{D}')$.

Proof. This lemma is proved similarly to Lemma 2.10. □

Lemma 2.14. If $\Gamma \vdash P : w$ and $P \equiv Q$, then $\Gamma \vdash Q : w$ and $\mathcal{M}_\Gamma(P) = \mathcal{M}_\Gamma(Q)$.

Proof. Lemma 2.9 gives $\Gamma \vdash Q : w$. To prove the equality, we reason by induction on the derivation of $P \equiv Q$. □

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

Lemma 2.15. If $\Gamma \vdash P : w$ and $P \longrightarrow P'$, then $\mathcal{M}_\Gamma(P) >_{mul} \mathcal{M}_\Gamma(P')$.

Proof. We reason by induction on the derivation of $P \longrightarrow P'$.

— $P \longrightarrow P'$ with $P \equiv Q$, $Q \longrightarrow Q'$ and $Q' \equiv P'$. We have by induction $\mathcal{M}_\Gamma(Q) >_{mul} \mathcal{M}_\Gamma(Q')$, and Lemma 2.14 yields the desired property.
— $P = a(x).P_1 \mid \overline{a}\langle v\rangle \longrightarrow P' = P_1[v/x]$. Consider a typing derivation $\mathcal{D}_P : \Gamma \vdash P : w$. $\mathcal{D}_P$ is constructed from $\mathcal{D}_1 : \Gamma \vdash a(x).P_1 : w_1$ and $\mathcal{D}_2 : \Gamma \vdash \overline{a}\langle v\rangle : k$. In turn, $\mathcal{D}_1$ has been obtained from $\mathcal{D}_3 : \Gamma, x : T \vdash P_1 : w_1$, and $\mathcal{M}(\mathcal{D}_1) = \mathcal{M}(\mathcal{D}_3)$.
By Lemma 2.13, there exists a derivation $\mathcal{D}' : \Gamma, x : T \vdash P_1[v/x] : w'$, and $\mathcal{M}(\mathcal{D}_3) \geqslant_{mul} \mathcal{M}(\mathcal{D}')$. We can indeed apply this lemma: let $\Gamma(a) = \sharp^k T_a$ and $\Gamma(v) = T_v$, then, since the input and the output prefixes on a are typable, we have $T_v \leqslant T_a \leqslant T$.
We obtain: $\mathcal{M}(\mathcal{D}_P) = \mathcal{M}(\mathcal{D}_1) \uplus \{\mathrm{lvl}_\Gamma(a)\} = \mathcal{M}(\mathcal{D}_3) \uplus \{\mathrm{lvl}_\Gamma(a)\} >_{mul} \mathcal{M}(\mathcal{D}') = \mathcal{M}(\mathcal{D}_{P'})$, where $\mathcal{D}_{P'}$ is the typing derivation for P'.

Thus for any typing derivation for P, there exists a typing derivation for P' whose measure is strictly smaller for $>_{mul}$. This gives $\mathcal{M}_\Gamma(P) >_{mul} \mathcal{M}_\Gamma(P')$.

— $P = !a(x).P_1 \mid \overline{a}\langle v\rangle \longrightarrow P' = !a(x).P_1 \mid P_1[v/x]$. We reason like above, keeping similar notations, and write $\mathcal{M}(\mathcal{D}_P) = \varnothing \uplus \{\mathrm{lvl}_\Gamma(a)\}$, and $\mathcal{M}(\mathcal{D}_{P'}) = \varnothing \uplus \mathcal{M}(\mathcal{D}')$, where $\mathcal{D}'$ has been obtained like above using Lemma 2.13, to type $P_1[v/x]$.
We moreover observe that the derivation for $\Gamma \vdash !a(x).P_1 : 0$ comes from a typing derivation $\mathcal{D}_3 : \Gamma, x : T \vdash P_1 : w_1$, with $w_1 < \mathrm{lvl}_\Gamma(a)$, and we have, as remarked above, $\mathcal{M}(\mathcal{D}_3) \leqslant_{mul} \{w_1\}$.
We thus obtain $\mathcal{M}(\mathcal{D}_P) = \{\mathrm{lvl}_\Gamma(a)\} >_{mul} \{w_1\} \geqslant_{mul} \mathcal{M}(\mathcal{D}_3) \geqslant_{mul} \mathcal{M}(\mathcal{D}_{P'})$.
Since this reasoning holds for any typing derivation for P, we can deduce $\mathcal{M}_\Gamma(P) >_{mul} \mathcal{M}_\Gamma(P')$.

□

Theorem 2.16 (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 2.11, every P_i is typable. Using Lemma 2.15 we have $\mathcal{M}_\Gamma(P_i) >_{mul} \mathcal{M}_\Gamma(P_{i+1})$ for all i, which yields a contradiction. □

Remark 2.17 (à la Curry vs. à la Church). Our system is presented à la Curry. Existing systems for termination (Demangeon *et al.* 2010; Deng and Sangiorgi 2006) are à la Church, while the usual presentations of i/o-types (Pierce and Sangiorgi 1996) are à la Curry. The latter style of presentation is better suited to address type inference (see Section 5). This has however some technical consequences in our proofs. Most importantly, the measure on processes (Definition 2.12) would be simpler in a setting à la Church, because we could avoid to consider all possible derivations of a given judgement. A Church-style presentation of an extension of the type system we have just studied is given in Section 4.1

3. Expressiveness

For the purpose of the discussions in this section, we work in a polyadic calculus. As discussed above, the extension of our type system to handle polyadicity 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 (2006), as expressed by the two following observations (Lemma 3.1 and Example 3.2):

Lemma 3.1. Any process typable according to the first type system of Deng and Sangiorgi (2006) is typable in our system.

Proof. The presentation of Deng and Sangiorgi (2006) 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, linear 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 $\Gamma \vdash_D P : w$. In the rules for input and output, the appropriate capability is deduced for a by a simple usage of subtyping. □

We discuss the other type systems for termination introduced in Deng and Sangiorgi (2006) in Section 3.3 below.

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 with 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 3.2 (level-polymorphism). Consider the following definitions (in addition to polyadicity, we accommodate the first-order type of natural numbers, with corresponding primitive operations):

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

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 acts like Church numeral 2).

Let us now examine how we can typecheck the process

$$P_1 \stackrel{def}{=} F_1 \mid F_2 \mid Q \mid \overline{g}\langle f_1, 4, t_1\rangle \mid \overline{g}\langle f_2, 5, t_2\rangle.$$

F_2 contains a call to f_1 under a replicated input on f_2, which forces $\mathrm{lvl}_\Gamma(f_2) > \mathrm{lvl}_\Gamma(f_1)$. In the type systems of Deng and Sangiorgi (2006), assigning different levels to f_1 and f_2 prevents us from typing the processes above, since the fact that f_1 and f_2 are emitted on h forces these names to have the same type, and hence in particular the same level (see also the discussion in Section 3.3). 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 $k_1 < k_2$.

Process P_1 can be phrased (and hence recognized as terminating) in the 'purely functional π-calculus' of Demangeon *et al.* (2010), that is, using a semantics-based approach – see Section 4. It is, however, not difficult to present a variation on P_1 that forces one to rely on level-based type systems (for instance by adding a parallel component with a non-replicated input on f_2), thus being outside the scope of Demangeon *et al.* (2010).

3.2. *Encoding the simply-typed λ-calculus*

We now investigate further 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:

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

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

We can make the following remarks:

— A simply-typed λ-term is encoded into a simply-typed process (see Sangiorgi and Walker (2001)). Typability for termination comes into play in the translation of λ-abstractions (that introduces replication).
— The target of this encoding is Lπ, the *localised π-calculus*, in which only the output capability is transmitted (we return to Lπ in Section 5.1).

Demangeon *et al.* (2009) provides a counterexample to typability of this encoding for the first type system of Deng and Sangiorgi (2006) (the proof of this result actually entails that typability according to the other, more expressive, type systems due to Deng and Sangiorgi also fails to hold). Let us study this example in our setting:

Example 3.3 (from Demangeon *et al.* (2009)). 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 typing hypotheses $v : \sigma$, $u : \sigma \longrightarrow \tau$, and $f : (\sigma \longrightarrow \tau) \longrightarrow \tau \longrightarrow \tau$. Computing $[\![M_1]\!]_p$ yields the process:

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

To typecheck this term using the first type system of Deng and Sangiorgi (2006), 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 (simple) type 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 $\mathrm{lvl}_\Gamma(y) > \mathrm{lvl}_\Gamma(f_3) = \mathrm{lvl}_\Gamma(u)$ 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 Deng and Sangiorgi (2006), the system of Section 2 accepts this process. Indeed, in that setting, y and u need not have the same levels, so that we can satisfy the constraint $\mathrm{lvl}_\Gamma(y) > \mathrm{lvl}_\Gamma(u)$. The last line in $[\![M_1]\!]_p$'s expression 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.

We can observe that in this example, a form of level polymorphism is at work, since we need to assign different levels to y and u, and at the same time we emit both names on f.

Our system is not able to typecheck the whole image of STλ, as the following (new) counterexample shows:

Example 3.4. We first look at the following rather simple π-calculus process:

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

We use a CCS-like notation for channels that are used to transmit the $\star$ value. We can observe that this process belongs to the *localised π-calculus*. It 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 $m \leqslant n$, which leads to an inconsistency.

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

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

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 then perform two communications (on y_1 and y_3), 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 2.11), 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 (while M_2 belongs to STλ).

3.3. *Discussion on expressiveness and related works*

The system of Section 2 is a refinement of the first type system of Deng and Sangiorgi (2006) (called 'the core system' in that work) – let us call the latter system $\mathcal{S}_1$, for the

sake of the present discussion. $\mathcal{S}_1$ serves as a starting point for several extensions in Deng and Sangiorgi (2006), which we discuss here. Another kind of extension has been studied in Demangeon *et al.* (2010), where the 'level-based' and 'semantics-based' approaches are combined: the type systems we study in Section 4 follow that direction.

A first extension of $\mathcal{S}_1$ introduced in Deng and Sangiorgi (2006) focuses on analysing the argument of communications, in the example situation where natural numbers are transmitted (what is important there is that transmitted values belong to a well-founded set). This allows the authors to introduce a type system which is capable to accept as terminating the (encoding in the π-calculus of) recursive functions. In some sense, this extension is orthogonal to the analysis of termination in presence of name-passing. We therefore believe that it should be possible to adapt this analysis to our system without major difficulty.

Deng and Sangiorgi (2006) then moves on to two other extensions of $\mathcal{S}_1$: the analysis of *sequences of input prefixes* (system $\mathcal{S}_2$), and a type system exploiting partial orders on names (system $\mathcal{S}_3$). These enriched type systems address the question of termination in presence of 'recursive outputs.'

Process $!a(x).b(y).\overline{a}\langle y\rangle$ is a very simple example of a process accepted by $\mathcal{S}_2$. This process is not typable using the rules of Figure 1, because of the 'recursive output' on a. However, it is harmless from the point of view of termination, as soon as one realizes that to release the output on a, one has to provide both an output on a *and* an output on b. This is the basic idea behind sequences of input prefixes.

The last system of Deng and Sangiorgi (2006), $\mathcal{S}_3$, enriches further the analysis by introducing a well-founded order between names: a process like $!p(x).a(y).(\overline{p}\langle x\rangle \mid \overline{b}\langle y\rangle)$, in which we furthermore impose that a and b have the same type, is rejected by $\mathcal{S}_2$ – $\overline{b}\langle y\rangle$ acts here as a recursive output. It can be accepted by $\mathcal{S}_3$, as long as a is above b according to the partial order on names (note that we also rely on sequences of input prefixes to typecheck this process).

If we now come back to the processes of Example 3.2, we can notice that P_1 is not typable according to $\mathcal{S}_2$, because the F_2 component forces the level of f_2 to be strictly higher than f_1's.

On the other hand, P_1 can be typed using $\mathcal{S}_3$, by assigning the same level to f_1 and f_2, and by letting f_2 be above f_1 according to the partial order. This would indeed allow one to typecheck F_2, and level polymorphism would not be necessary (since f_1 and f_2 would have the same level) to typecheck two calls to g, with f_1 and f_2, in P_1.

It is however not difficult to modify slightly the definition P_1 so that is not typable using $\mathcal{S}_3$, while still being typable according to the type system of Section 2. For instance, we can define $F_2' = a(f_1).b(f_2).F_2$ and replace F_2 in P_1 with $F_2' \mid \overline{a}\langle f_1\rangle \mid \overline{b}\langle f_2\rangle$. This has the effect of forbidding to compare names f_1 and f_2 using the partial order, thus making $\mathcal{S}_3$ reject this modified process.

Conversely, there are processes that are typable according to $\mathcal{S}_3$, but not using the system of Section 2. An example is given by $!p(a,b).a(x).(\overline{p}\langle a,b\rangle \mid \overline{b}\langle x\rangle) \mid \overline{p}\langle u,v\rangle \mid \overline{p}\langle v,w\rangle$, which, as discussed in Deng and Sangiorgi (2006), comes from the encoding of a concurrent list structure. In $\mathcal{S}_3$, the type assigned to p carries the information that the first component of the received pair dominates the second component according to the partial order, so that

the replicated process is typable (as above, the input on a 'dominates' the output on b). This process is not typable using the rules of Figure 1, even if we consider an extension of these rules with sequences of input prefixes, essentially because a and b must have the same type (this is forced by the $\overline{p}\langle u, v\rangle \mid \overline{p}\langle v, w\rangle$ subterm).

It thus turns out that the type system of Section 2 extends $\mathcal{S}_1$ (Lemma 3.1), but is otherwise incomparable with all other systems of Deng and Sangiorgi (2006).

While $\mathcal{S}_2$ and $\mathcal{S}_3$ are based on a fine analysis of 'recursive outputs,' the examples studied in Sections 3.1 and 3.2 show that subtyping gives more flexibility when transmitting names, making it possible, in particular, to exploit a form of level polymorphism. These two kinds of approaches should bring complementary benefits in analysing termination of processes. It would therefore be interesting to try to combine the type system we propose with systems $\mathcal{S}_2$ (sequences of input prefixes) and $\mathcal{S}_3$ (partial orders).

4. Subtyping and functional names

4.1. *A Church-style type system*

4.1.1. *Definitions and preliminary properties.* In order to handle functional computation as expressed by STλ, we extend the system of Section 2 along the lines of Demangeon *et al.* (2010). The idea is to classify names into *functional* and *imperative* names. Intuitively, functional names arise through the encoding of STλ into the π-calculus. For termination, since we have seen that the type system of Section 2 is not expressive enough, they are dealt with using an appropriate method – the 'semantics-based' approaches (Sangiorgi 2006; Yoshida *et al.* 2004), discussed in Section 1. For the names that are not functional, that we call imperative names, we resort to (an adaptation of) the rules of Section 2.

Because we follow Demangeon *et al.* (2010), our first type system is à la Church – we define a second type system in Section 4.2, which is in some sense closer to the one of Section 2, being à la Curry. In presenting our type system and its soundness proof, we stress the parts where subtyping brings additional difficulties w.r.t. Demangeon *et al.* (2010). In some other places, on the other hand, proofs are essentially the same, and we refer to that work for more details. Overall, the presentation should be self-contained.

We now introduce a π-calculus with functional definitions. Names are divided into two sets. Functional names are ranged over using $f, g, h, \ldots$. Names that are not functional are called *imperative*, and are ranged over using $a, b, c, \ldots$. We use $n, m, \ldots, x, y, \ldots$ to range over all names, functional and imperative. In the system à la Church, functional names are introduced using *functional definitions*, that are special processes of the form $\texttt{def}\ f = (x).P\ \texttt{in}\ Q$. Here f is a functional name, and the only persistent input on f is given by the parametrized process $(x).P$. The grammar for processes is as follows:

$$P \quad ::= \quad \mathbf{0} \mid P_1 | P_2 \mid \overline{n}\langle v\rangle \mid (\boldsymbol{\nu} a)\, P \mid a(x).P \mid\ !a(x).P \mid \texttt{def}\ f = (x).P\ \texttt{in}\ Q,$$

(values are defined like in Section 2.1). In $\texttt{def}\ f = (x).P\ \texttt{in}\ Q$, x is bound in P and f is bound in Q (we actually forbid usages of f in P).

The presentation of the type system follows the one of Demangeon *et al.* (2010). In particular, types are à la Church: there exists a priori a function, noted Γ, that associates

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

$$\frac{\Gamma \vdash a : \mathsf{i}^k T \qquad \Gamma(x) = T \qquad \Gamma \vdash P : w \qquad k > w}{\Gamma \vdash (!)\, a(x).P : 0} \qquad \frac{\Gamma \vdash n : \mathsf{o}^k T \qquad \Gamma \vdash v : T \qquad \mathrm{lvl}_\Gamma(n) = k}{\Gamma \vdash \overline{n}\langle v \rangle : k}$$

$$\frac{\Gamma(f) = \mathsf{o}^k T \qquad \Gamma(x) = T \qquad \Gamma \vdash P : w \qquad k \geq w \qquad f \notin \mathrm{fn}(P) \qquad \Gamma \vdash Q : w'}{\Gamma \vdash \mathtt{def}\ f = (x).P\ \mathtt{in}\ Q : w'}$$

$$\frac{\Gamma \vdash P : w}{\Gamma \vdash (\nu n)\, P : w}\ \Gamma(n) \neq \mathbb{U} \qquad \frac{\Gamma \vdash P_1 : w_1 \qquad \Gamma \vdash P_2 : w_2}{\Gamma \vdash P_1 | P_2 : \max(w_1, w_2)} \qquad \frac{}{\Gamma \vdash \mathbf{0} : 0}$$

Fig. 3. Types à la Church for a calculus with functional and imperative channels.

every name with its type. In order to perform α-conversion, we need to postulate, for every type, the existence of an infinite number of names having that type. We write $\mathrm{lvl}_\Gamma(n) = k$ when $\Gamma(n) = T$ with $T \in \{\mathsf{o}^k U, \mathsf{i}^k U, \sharp^k U\}$ for some U, that is, $\mathrm{lvl}_\Gamma(n)$ is the level of name n according to Γ, without use of subtyping. Note that we use the same notation as in Section 2, as $\mathrm{lvl}_\Gamma(a)$ depends on Γ: in a system à la Curry, Γ is specific to a given typing derivation, while Γ is fixed priori in a Church-style type system.

We furthermore impose that the typing context Γ associates a type of the form $\mathsf{o}^k T$ to name n if and only if n is a functional name. On the other hand, Γ can associate types of the form $\sharp^k T$ or $\mathsf{i}^k T$ to imperative names: while it seems reasonable to use $\sharp^k T$ for names that are created using restriction (in order for the name to be usable both in input and output), it can well be the case that only the input capability on a name is received, either through an input or through a functional definition. In such a situation, the abstracted name has a type of the form $\mathsf{i}^k T$.

As we discuss below (Remark 4.16), it is possible in our calculus to emit only the output capability of an imperative name, say a. As the communication occurs, a replaces a (bound) name, say x, which has a type of the form $\mathsf{o}^k T$, and is hence functional. In some sense, a is transmitted in this case 'as a functional name,' to take the place of x.

The typing rules are presented on Figure 3. The typing judgement is of the form $\Gamma \vdash P : w$, where w is a natural number. As said above, Γ is here given a priori, and does not change along the construction of a typing derivation.

According to the usage of functional names in Sangiorgi (2006), in the typing rule for definitions, f should not occur in P (no recursion is allowed), and can be used only in output in Q. In particular, it can be observed that a functional name offers only the output capability: the input capability is at work only in the construct for definition (which also acts as a restriction, in the sense that it introduces a new name). Accordingly, as explained above, Γ associates types of the form $\mathsf{o}^k T$ to functional names. Note that the condition on levels is relaxed in the typing rule for definitions: we have $k \geqslant w$ instead of $k > w$.

Note that we write a single typing rule for inputs on imperative channels, be them replicated or not – hence the notation $(!)a(x).P$ in the conclusion of the rule, that stands for either $!a(x).P$ or $a(x).P$; this notation will be used throughout this section. Indeed, like

in Demangeon *et al.* (2010), typing non-replicated inputs (on imperative names) involves the same constraints as for replicated inputs: the relaxed control over functional names makes it necessary to be more restrictive on all usages of imperative names. The following divergent process from Demangeon *et al.* (2010):

$$\texttt{def } f = (x).(\overline{a}\langle x\rangle \mid \overline{x}\langle t\rangle) \texttt{ in def } g = (z).a(y).\overline{f}\langle y\rangle \texttt{ in } \overline{f}\langle g\rangle,$$

illustrates why the linear inputs on imperative names (here, a) have to be controlled. In this example, the strict inequality imposed when typing $a(y)\overline{f}\langle y\rangle$ prevents the process from being typable.

Expressiveness of the type system. As in Demangeon *et al.* (2010), it can be shown that the encoding of STλ of Section 3.2 yields processes where only functional names are used. Such processes are typable using the rules of Figure 3, by assigning level 0 to all names. Moreover, as it is mentioned in Section 3, subtyping brings additional expressiveness to our system w.r.t. Demangeon *et al.* (2010) – we return to this point in Remark 4.16.

Operational semantics. The reduction relation is written $\longrightarrow$, and is defined using *evaluation contexts*, ranged over with E, and given by

$$E \quad ::= \quad [\,] \mid E|P \mid (\boldsymbol{\nu} a)\, E \mid \texttt{def } f = (x).P \texttt{ in } E,$$

where $[\,]$ is the *hole* of the context. $E[P]$ is the process obtained by replacing the hole with P in E. We write $\mathrm{capt}(E)$ for the set of names *captured* by E, that is, those names that are bound at the occurrence of the hole in E. $\longrightarrow$ is defined by the following rules:

$$\frac{\mathrm{capt}(E') \cap \mathrm{fn}((x).P) = \varnothing}{E[\texttt{def } f = (x).P \texttt{ in } E'[\overline{f}\langle v\rangle]] \;\longrightarrow\; E[\texttt{def } f = (x).P \texttt{ in } E'[P[v/x]]}$$

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

$$\frac{Q \equiv P \qquad P \longrightarrow P' \qquad P' \equiv Q'}{Q \longrightarrow Q'}.$$

In the last rule, $\equiv$ denotes structural congruence, defined by adding the following axiom to the definition of $\equiv$ in Section 2.1:

$$Q \mid \texttt{def } f = (x).P_1 \texttt{ in } P_2 \quad \equiv \quad \texttt{def } f = (x).P_1 \texttt{ in } P_2|Q \quad \text{if } f \notin \mathrm{fn}(Q).$$

Proposition 4.1 (subject reduction). If $\Gamma \vdash P : w$ and $P \longrightarrow P'$, then $\Gamma \vdash P : w'$ for some $w' \leqslant w$.

Proof (sketch). We reason by induction over the derivation of $P \longrightarrow P'$, and exploit a lemma about substitution which corresponds to Lemma 2.10 in Section 2. □

Definition 4.2 (similarity). Similarity, written $\gtrsim$, is the greatest relation such that whenever $P \gtrsim Q$ and $Q \longrightarrow Q'$, there exists P' such that $P \longrightarrow P'$ and $P' \gtrsim Q'$.

The defining property of similarity will sometimes be written $(\gtrsim\longrightarrow)\subseteq(\longrightarrow\gtrsim)$.

Lemma 4.3. If $P \gtrsim Q$, then for any E, $E[P] \gtrsim E[Q]$.

4.1.2. *Termination.* We now establish the soundness of our type system, i.e. that all typable processes terminate. The overall structure of the proof, which follows the one in Demangeon *et al.* (2010), is as follows. We reason by contradiction, supposing the existence of an infinite sequence of reductions starting from a typable term. The main ingredient in the proof is the definition of a *pruning function* (Definition 4.4), that maps a process into a functional process (a process that uses only functional names). By pruning all terms involved in the original divergence, we show that we obtain a divergent computation involving only functional processes. This is established thanks to a *simulation result* (Proposition 4.13). Since all typable functional processes terminate (Theorem 4.8), we obtain a contradiction. More intuitions about the technical notions involved in the proof are provided below.

For technical reasons, the pruning function is parametric w.r.t. an integer, p, that will be fixed in the proof of Theorem 4.15.

Definition 4.4 (pruning). Given P such that $\Gamma \vdash P : w$, we define the *pruning of P at level p*, written $\mathsf{prun}_p(P)$, by induction over P as follows:

$$\mathsf{prun}_p(\mathtt{def}\ f = (x).P\ \mathtt{in}\ Q) = \begin{cases} \mathtt{def}\ f = (x).\mathsf{prun}_p(P)\ \mathtt{in}\ \mathsf{prun}_p(Q) & \text{if } \Gamma(f) = \mathsf{o}^p T \\ \mathbf{0} & \text{otherwise} \end{cases}$$

$$\mathsf{prun}_p(\overline{n}\langle v\rangle) = \begin{cases} \overline{n}\langle v\rangle \text{ if } \Gamma(n) = \mathsf{o}^p T \\ \mathbf{0} \quad \text{otherwise} \end{cases} \qquad \mathsf{prun}_p((!)a(x).P) = \mathbf{0}$$

$$\mathsf{prun}_p(P_1|P_2) = \mathsf{prun}_p(P_1)\,|\,\mathsf{prun}_p(P_2) \qquad \mathsf{prun}_p((\boldsymbol{\nu} a)\,P) = \mathsf{prun}_p(P) \qquad \mathsf{prun}_p(\mathbf{0}) = \mathbf{0}.$$

Pruning is extended to evaluation contexts in the obvious way ($\mathsf{prun}_p([\,]) = [\,]$).

Intuitively, the intent in the definition of $\mathsf{prun}_p(P)$ is to keep all the parts of P that involve communications on functional channels whose level is p. Therefore, all outputs occurring in $\mathsf{prun}_p(P)$ are at functional names whose level is p.

Lemma 4.5 (pruning and $\equiv$). If $P \equiv Q$, then $\mathsf{prun}_p(P) \equiv \mathsf{prun}_p(Q)$.

Definition 4.6. We say that a process P is *functional* if all occurrences of imperative names in P are as arguments of inputs or outputs.

Lemma 4.7. The pruning of a typable process is a typable functional process.

The intuition in Definition 4.6 is that imperative names are allowed only in 'inert occurrences' in a functional process: they are never used. It is important to realize that when $\mathsf{prun}_p(\mathtt{def}\ f = (x).P\ \mathtt{in}\ Q) \neq \mathbf{0}$, x can only be used in $\mathsf{prun}_p(P)$ in output: indeed, either x is functional, in which case its type guarantees this property, or x is imperative (either $\Gamma(x) = \sharp^k T_0$ or $\Gamma(x) = \mathsf{i}^k T_0$), in which case all inputs and outputs at x are erased when computing $\mathsf{prun}_p(P)$, and x can only be transmitted (on a functional

channel). Hence, in the pruning of a process, imperative names that remain can only be transmitted, and are never used to communicate. Note that we could actually modify the definition of pruning, in order to obtain processes that use only functional names. For this, we should postulate the existence of a function $\mathcal{F}$ that maps every name of type $\sharp^k T$ or $\mathsf{i}^k T$ to a name of type $\mathsf{o}^k T$. We would then apply $\mathcal{F}$ to x (resp. to v) in the defining clause of pruning for definitions (resp. messages), replacing all 'inert imperative names' with functional names. We preferred instead to keep the definition of pruning simpler.

Theorem 4.8. All well-typed functional processes terminate.

Proof. It can be shown (see Demangeon *et al.* (2010)) that a functional process corresponds to a process of the calculus of Sangiorgi (2006), and that whenever P is functional and $P \longrightarrow P'$, there is a reduction between the processes corresponding to P and P' in that calculus. We can therefore rely on the proof in Sangiorgi (2006) to derive termination. □

By Lemma 4.7 and Theorem 4.8, the pruning of a typable process always yields a terminating process.

We now establish the simulation property (Proposition 4.13). Several technical results are necessary for this.

Definition 4.9 (extension relation). $\sqsupset$ is the smallest reflexive and transitive relation between processes that contains all pairs of the form $(\overline{n}\langle v\rangle, \mathbf{0})$ and is a congruence.

Intuitively, $P \sqsupset Q$ holds if Q is obtained by replacing some messages in P with $\mathbf{0}$.

Lemma 4.10. $\sqsupset \subseteq \gtrsim$.

Lemma 4.11 (pruning and substitution). Suppose that we have $\Gamma(x) = T$, $\mathrm{lvl}_\Gamma(x) \leqslant p$ and $\Gamma \vdash v : T$. Then $(\mathsf{prun}_p(P))[v/x] \sqsupset \mathsf{prun}_p(P[v/x])$, and both processes are functional.

Proof (sketch). The interesting case corresponds to the pruning of $\overline{n}\langle t\rangle$, when $n = x$ and $\Gamma(n) = \mathsf{o}^p T'$. Then $(\mathsf{prun}_p(\overline{n}\langle t\rangle))[v/x] = \overline{v}\langle t\rangle$ (note that $t[v/x] = t$ for typing reasons).

We moreover know $\Gamma \vdash v : \mathsf{o}^p T'$. In case v is a functional name, i.e. $\Gamma(v) = \mathsf{o}^k T''$ for some $k \leqslant p$ and T'', we have that $\mathsf{prun}_p(\overline{n}\langle t\rangle[v/x])$ is either $\mathbf{0}$ (if $k < p$) or $\overline{v}\langle t\rangle$ (if $k = p$). When v is not functional (this happens for instance whenever $\Gamma(v) = \sharp^{k'} T''$ for some k', T''), $\mathsf{prun}_p(\overline{n}\langle t\rangle[v/x]) = \mathbf{0}$. Finally, we have indeed $(\mathsf{prun}_p(P))[v/x] \sqsupset \mathsf{prun}_p(P[v/x])$. □

Lemma 4.11 is simpler in Demangeon *et al.* (2010): we have $=$ instead of $\sqsupset$, essentially because in the absence of subtyping, substitutions involve names having *exactly the same type* (hence the same level: a name whose level is p cannot replace a name having a different level).

Lemma 4.12 (pruning – inactivity). We have $\mathbf{0} \gtrsim (\mathsf{prun}_p(P_1))[v/x]$ whenever $\Gamma(x) = T$ and $\Gamma \vdash v : T$ for some T, in each of the two following situations:

1. $\Gamma \vdash \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2 : w$ with $\mathrm{lvl}_\Gamma(f) = n$ and $n < p$;
2. either $\Gamma \vdash !a(x).P_1 : w$ or $\Gamma \vdash a(x).P_1 : w$ with $\mathrm{lvl}_\Gamma(a) = n$ and $n \leqslant p$.

The proof of Lemma 4.12 follows closely the lines of the corresponding result in Demangeon *et al.* (2010). The intuition is that, in each case, no occurrence of outputs on functional names at levels $\geqslant p$ can occur in P_1, so that all outputs are erased by pruning (more precisely, such outputs can occur in the body of definitions that P_1 may contain, but in this case these definitions cannot be triggered, and are thus equivalent to $\mathbf{0}$, since this would require an output on a functional name whose level is $\geqslant p$ in the '`in...`' part of the definition).

We write $\longrightarrow_k$ (resp. $\longrightarrow^k$) when reduction is derived using a communication on a functional (resp. imperative) name n such that $\mathrm{lvl}_\Gamma(n) = k$.

Proposition 4.13 (simulation). Suppose $\Gamma \vdash P : w$. Then:

1. if $P \longrightarrow_p P'$ then $\mathsf{prun}_p(P) \longrightarrow\gtrsim \mathsf{prun}_p(P')$;
2. if either $P \longrightarrow_n P'$ with $n < p$ or $P \longrightarrow^n P'$ with $n \leqslant p$, then $\mathsf{prun}_p(P) \gtrsim \mathsf{prun}_p(P')$.

Proof.

1. By induction on the derivation of $P \longrightarrow_p P'$. The interesting case is when

$$P = \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ E[\overline{f}\langle v\rangle] \qquad \text{and} \qquad P' = \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ E[\,P_1[v/x]\,]$$

(for the sake of simplicity, we omit the common context above P and P'). Then

$$\mathsf{prun}_p(P) = \mathtt{def}\ f = (x).\mathsf{prun}_p(P_1)\ \mathtt{in}\ E'[\overline{f}\langle v\rangle] \quad \text{and}$$
$$\mathsf{prun}_p(P') = \mathtt{def}\ f = (x).\mathsf{prun}_p(P_1)\ \mathtt{in}\ E'[\mathsf{prun}_p(P_1[v/x])], \text{ with } E' = \mathsf{prun}_p(E).$$

We moreover have:

$$\mathsf{prun}_p(P) \longrightarrow \left(\mathtt{def}\ f = (x).\mathsf{prun}_p(P_1)\ \mathtt{in}\ E'[\,(\mathsf{prun}_p(P_1))[v/x]\,]\right) = P_0,$$

and, in the case where $\mathrm{lvl}_\Gamma(x) \leqslant p$, $P_0 \sqsupset \mathsf{prun}_p(P')$ by Lemma 4.11, which yields the expected result by applying Lemmas 4.10 and 4.3.
Suppose now that $\mathrm{lvl}_\Gamma(x) > p$. If x is imperative, then all usages of x (in input or in output) are removed by pruning, and $(\mathsf{prun}_p(P_1))[v/x] = \mathsf{prun}_p(P_1[v/x])$.
If x is functional, x can only be used in output in P_1. Moreover, because P is typable, no output on x can occur in P_1 without occurring under an input prefix or inside a definition. However, since $\mathrm{lvl}_\Gamma(x) > p$, such input prefixed processes or definitions are necessarily removed by pruning, still as a consequence of typability. Therefore $(\mathsf{prun}_p(P_1))[v/x] = \mathsf{prun}_p(P_1[v/x])$, also in this case.
2. We sketch the reasoning for the second property, which is proved by induction over the derivation of $P \longrightarrow^n P'$ with $n \leqslant p$. The interesting case corresponds to $P = !a(x).P_1 \mid \overline{a}\langle v\rangle$, $\mathrm{lvl}_\Gamma(a) = n$ and $P' = !a(x).P_1 \mid P_1[v/x]$. Then $\mathsf{prun}_p(P) = \mathbf{0}$ (since a is an imperative name), and $\mathsf{prun}_p(P') = \mathsf{prun}_p(P_1[v/x])$.
If $\mathrm{lvl}_\Gamma(x) \leqslant p$, we can rely on Lemma 4.11 to deduce $\mathsf{prun}_p(P_1)[v/x] \sqsupset \mathsf{prun}_p(P_1[v/x])$, which then gives, by Lemma 4.10, $\mathsf{prun}_p(P_1)[v/x] \gtrsim \mathsf{prun}_p(P_1[v/x])$, and we can apply Lemma 4.12 to deduce that we have $\mathbf{0} \gtrsim \mathsf{prun}_p(P_1)[v/x] \gtrsim \mathsf{prun}_p(P')$, hence the expected result.
If, on the other hand, $\mathrm{lvl}_\Gamma(x) > p$, we reason as above to show that we can apply Lemma 4.12 and deduce the expected result.

□

Again, the corresponding result in Demangeon *et al.* (2010) is simpler: in absence of subtyping, a reduction $P \longrightarrow_p P'$ implies $\mathsf{prun}_p(P) \longrightarrow \mathsf{prun}_p(P')$. Here $\gtrsim$ appears due to Lemmas 4.11 and 4.10.

We need a last result before proving the main theorem of this section.

Proposition 4.14 (decreasing measure). Whenever $\Gamma \vdash P : w$, we define $\mathcal{C}_p(P)$ as the number of outputs at level p that occur in P without occurring under an input prefix or inside the body of a definition. We have:

— If $P \longrightarrow^p P'$ then $\mathcal{C}_p(P) > \mathcal{C}_p(P')$.
— If $P \longrightarrow_n P'$ or $P \longrightarrow^n P'$ with $n < p$ then $\mathcal{C}_p(P) \geqslant \mathcal{C}_p(P')$.

Above, it makes sense to write $\mathcal{C}_p(P')$ because of Proposition 4.1 (subject reduction).

In the proof of the following result, integer p plays a pivotal rôle to establish that the existence of a divergence is contradictory. p is the (highest) level where the divergence takes place, in the sense that infinitely many communications involve synchronizations on channels whose level is p. $\mathcal{C}_p(P)$ is the quantity of 'available messages' at p. By the above Proposition, this quantity decreases along imperative reductions (as it is the case for the type system of Section 2.1); moreover, it cannot increase along other kinds of reductions, as a consequence of the definition of the type system of this section. This allows us to derive a contradiction.

Theorem 4.15 (soundness). Suppose $\Gamma \vdash P : w$. Then P is terminating.

Proof. We reason by contradiction, and consider a divergent computation arising from $P = P_0 \longrightarrow P_1 \longrightarrow \ldots$. Since P is typed using a finite number of levels, there exists an integer p such that for an infinite number of is, we have either $P_i \longrightarrow_p P_{i+1}$ or $P_i \longrightarrow^p P_{i+1}$. We can moreover suppose that we pick for p the largest integer having this property. This entails that at the cost of 'forgetting about' an initial part of the infinite sequence, we can suppose that there is no reduction $P_i \longrightarrow_n P_{i+1}$ or $P_i \longrightarrow^n P_{i+1}$ for $n > p$. So we consider an infinite sequence of reductions involving communications on names whose level is $\leqslant p$, and such that there are infinitely many reduction steps corresponding to a communication on a channel whose level is p.

We observe that among the communications at level p, there are necessarily infinitely many $\longrightarrow_p$ steps, since otherwise we would get a contradiction, by Proposition 4.14.

We can now apply pruning to all processes involved in this infinite sequence, which yields a sequence $(Q_i)_{i\geqslant 0}$ of functional processes (Lemma 4.7) such that, by the simulation property (Proposition 4.13), for all i, either $Q_i \longrightarrow\gtrsim Q_{i+1}$ or $Q_i \gtrsim Q_{i+1}$, and the former case occurs infinitely often.

By observing that $\gtrsim\gtrsim \subseteq \gtrsim$ and $(\gtrsim\longrightarrow) \subseteq (\longrightarrow\gtrsim)$, this yields a divergence arising from a well-typed (by Lemma 4.7) functional process, which contradicts Theorem 4.8. □

4.2. *A Curry-style type system*

Contrarily to the setting of Section 2, the type system of Section 4.1 is presented à la Church: every name has a type a priori, which means in particular that it is either

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

$$\frac{\Gamma \vdash a : \mathsf{i}^m T \qquad \Gamma, x : T, f : \mathsf{o}^k U \diamond \_ \vdash P : w \qquad m > w}{\Gamma \diamond f : \mathsf{o}^k U \vdash (!)a(x).P : 0}$$

$$\frac{\Gamma, f : \mathsf{o}^k T \diamond \_ \vdash P_1 \qquad \Gamma \diamond f : \mathsf{o}^k T \vdash P_2}{\Gamma \diamond f : \mathsf{o}^k T \vdash P_1 | P_2} \qquad \frac{\Gamma \diamond f : \mathsf{o}^k T \vdash P_1 \qquad \Gamma, f : \mathsf{o}^k T \diamond \_ \vdash P_2}{\Gamma \diamond f : \mathsf{o}^k T \vdash P_1 | P_2}$$

$$\frac{\Gamma, g : \mathsf{o}^k T \diamond f : \mathsf{o}^m U \vdash P : w}{\Gamma \diamond g : \mathsf{o}^k T \vdash (\nu f) P : w} \qquad \frac{\Gamma, a : \sharp^m T \diamond f : \mathsf{o}^k U \vdash P : w}{\Gamma \diamond f : \mathsf{o}^k U \vdash (\nu a) P : w} \qquad \frac{}{\Gamma \diamond f : \mathsf{o}^k T \vdash \mathbf{0} : 0}$$

Fig. 4. Types à la Curry for functional and imperative channels.

functional or imperative, and that it has *a* level (something we rely on for instance in the proof of Theorem 4.15). In this section, we get back to the calculus of Section 2 (without functional definitions), and show how we can build on i/o-capabilities to define a type system à la Curry that handles functional and imperative names. The kind of a name, functional or imperative, is fixed along the construction of a typing derivation: intuitively, some replicated inputs are recognized as being equivalent to definitions.

We first present the typing rules in Section 4.2.1, and then establish termination in Section 4.2.2. This proof relies on termination of the calculus of Section 4.1, and exploits a calculus of *expanded processes*, that intuitively lies in between the Church-style and the Curry-style calculi.

4.2.1. *Handling functional names using i/o-types.* We now manipulate typing environments of the form $\Gamma \diamond f : \mathsf{o}^k T$, where Γ is like in Section 2 – the intuition here is that we isolate a particular name, f. f is the name which can be used to build replicated inputs at top-level, and the intent is to treat f as a functional name. The typing rules are given on Figure 4.

The typing rules of Figure 4 rely on i/o-capabilities and the isolated name to enforce the usage of functional names as imposed by functional definitions in Section 4.1. Let us analyse how our system implements this usage.

As in Section 4.1, functional (resp. imperative) names have a type of the form $\mathsf{o}^k T$ (resp. $\sharp^k T$ or $\mathsf{i}^k T$). There are two rules to type restriction, 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 about the restricted name is added to the Γ part of the typing environment).

In the rule for input on an imperative name (replicated or not), the typing environment is of the form $\Gamma \diamond \_$ in the premise where we typecheck the continuation process: this has to be understood as $\Gamma \diamond d : \mathsf{o}^k T$, 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 in the continuation of the input on the imperative name. 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 replicated inputs on functional names under input prefixes is necessary because of diverging terms like the following one, from Demangeon *et al.* (2010) (a is imperative, f is functional):

$$(\boldsymbol{\nu} f)\big(\, a(x).!f(y).\overline{x}\langle y\rangle \;\mid \overline{a}\langle f\rangle \mid \overline{f}\langle v\rangle \,\big).$$

Note that in Section 4.1, (the counterpart of) this term cannot be written, since the `def` construct forces the replication on f to occur right under the restriction on f (which is not the case in this process: the input $a(x)$ comes in between).

The notation $\Gamma \diamond \_$ is also used in the rule to type a replicated input on a functional name. In this case, f cannot be used at all in the premise, to avoid recursion.

There are two typing rules for parallel composition, to remain close to the calculus of Section 4.1 (and for Lemma 4.33 below to hold): we impose that there is at most one replicated input on a functional name. Without major difficulty, these rules could be replaced with a single rule, where both premises have $f : \mathsf{o}^k T$ in their isolated part. This would yield a more expressive type system, where several definitions can be given for the same functional name.

As appears from the explanations we just gave, the isolated name is introduced to somehow mimic the behaviour of the `def` $f = (x).P_1$ `in` P_2 construct, which is not present in the system à la Curry. The latter construct insures that a functional name is used only once as a replicated input, and can be used as an output only in P_2. These properties are ensured by the typing rules of Figure 4.

Remark 4.16 (expressiveness). Our system provides more expressiveness than the one introduced in Demangeon *et al.* (2010). Indeed, we are able to typecheck, for instance, process

$$!u(x).\overline{x}\langle w\rangle \mid !v(z).\overline{u}\langle t\rangle \mid \overline{u}\langle v\rangle \mid \overline{u}\langle a\rangle \mid a(y).\overline{b}\langle y\rangle.$$

Here, name a is necessarily imperative (it is used in a non-replicated input) while name v must be functional (v can be unified with x, hence its level must be smaller than u's level; at the same time, subterm $!v(z).\overline{u}\langle t\rangle$ imposes that u's level is smaller than v's), and both are emitted on u. This is impossible in Demangeon *et al.* (2010), where every channel carries either a functional or an imperative name. In our setting, only the output capability on a is transmitted along u, so in a sense a is transmitted 'as a functional name' in this example.

4.2.2. *A calculus of expanded processes.* Since the typing rules of Figure 4 apply to the processes we have introduced in Section 2, our type system *recognizes* that some names are used in a functional way in a (plain) π-calculus process. On the other hand, the system of Section 4.1 needs the guidance of `def` constructs for that.

However, reasoning about this type system is rather difficult. Indeed, because of the particular handling of restrictions on functional names, typability is not preserved by $\equiv$. As an example, consider $P = (\boldsymbol{\nu} f)\,(!f(x).P_1 \mid (\boldsymbol{\nu} g)\,(!g(y).P_2 \mid P_3))$, where f and g have to be treated as functional names for P to be typable. Then $P \equiv (\boldsymbol{\nu} f)(\boldsymbol{\nu} g)\,(!f(x).P_1 \mid !g(y).P_2 \mid P_3)$, a term that cannot be typed, because we would need to have two isolated names to typecheck the inputs on f and g.

Accordingly, the analogue of Lemma 2.9 does not hold for this type system. In order to prove soundness, we establish a correspondence with the Church-style type system of Section 4.1. This is achieved by studying a calculus where structural congruence is controlled in a stricter way.

We now introduce *expanded processes*, that define an intermediate calculus between the calculus of Section 4.1, where functional definitions impose a rather rigid syntactic structure, and the calculus of Section 4.2.1, where no construct specific to functional names is provided. The intuition is that expanded processes correspond to a way to write Curry-style processes where inputs on functional names are grouped together – we call *Curry-style processes* the processes of Section 4.2, that is, the processes defined by the grammar of Section 2.

Because of space constraints, we do not provide full details for all proofs. We nevertheless enlighten the most important parts of the reasoning.

Definition 4.17 (expanded processes). The grammar of *expanded processes*, ranged over using A, B, is given by:

$$A, B \quad ::= \quad (\nu \widetilde{c})(\nu \widetilde{f}) \left(\Pi_i !f_i(x_i).A_i \mid \Pi_j (!)a_j(y_j).B_j \mid \Pi_k \overline{n}_k \langle v_k \rangle \right)$$

with the following constraints:

— $\Pi_i !f_i(x_i).A_i$ stands for $!f_1(x_1).A_1 \mid \ldots \mid !f_p(x_p).A_p$ for some $p \geqslant 0$, and $\widetilde{f} = (f_1, \ldots, f_p)$. If $p = 0$, then $\widetilde{f}$ is empty and $\Pi_i !f_i(x_i).A_i = \mathbf{0}$;

— the names in $\widetilde{f}$ furthermore satisfy:

  – $\forall i, n. (n \geqslant i) \Rightarrow f_n \notin \mathrm{fn}(A_i)$;

  – except for its unique occurrence in replicated input, every f_i is used only in output (in particular, the f_is are pairwise distinct, and different from all the a_js);

— the notation Π is also used for the two other components (inputs on the a_j and outputs on the n_k), without further constraints on the usage of these names.

Example 4.18. $A_0 = (\nu b)(\nu f_1, f_2) (!f_1(x).a(y).\overline{x}\langle y \rangle \mid !f_2(z).(\overline{f}_1\langle z \rangle | \overline{f}_1\langle z \rangle) \mid \overline{f}_2\langle b \rangle)$ is an example of an expanded process. We shall come back to it below, to illustrate the definitions and results we present about expanded processes.

The set of expanded processes is included in the set of Curry-style processes. Note that while all inputs on the f_is occur under a restriction, this is not necessarily the case for the outputs and inputs on the other names.

Reduction. We now introduce reduction between expanded processes, written $\longmapsto$.

Consider $A = (\nu \widetilde{c})(\nu \widetilde{f}) (\Pi_i !f_i(x_i).A_i \mid \Pi_j (!)a_j(y_j).B_j \mid \Pi_k \overline{n}_k \langle v_k \rangle)$, and suppose that $n_{k_0} = f_{i_0}$, for some $i_0 \leqslant p$ and $k_0 \leqslant k$. Let $(\nu \widetilde{d})(\nu \widetilde{g}) (\Pi_r !g_r(z_r).A'_r \mid \Pi_s (!)b_s(u_s).B'_s \mid \Pi_t \overline{m}_t \langle w_t \rangle)$ be *a copy* of $A_{i_0}[v_{k_0}/x_{i_0}]$ in which all bound names have been renamed using fresh names, in order to abide the Barendregt convention (all bound names are pairwise distinct and different from the free names). We then have $A \longmapsto A'$, with

$$A' = (\nu \widetilde{c})(\nu \widetilde{d})(\nu \widetilde{f})(\nu \widetilde{g}) \left(\Pi_i !f_i(x_i).A_i \mid \Pi_r !g_r(z_r).A'_r \right.$$
$$\left. \mid \Pi_j (!)a_j(y_j).B_j \mid \Pi_s (!)b_s(u_s).B'_s \mid \Pi_{k \neq k_0} \overline{n}_k \langle v_k \rangle \mid \Pi_t \overline{m}_t \langle w_t \rangle \right).$$

It can be checked that this definition makes sense because A' obeys the constraints of Definition 4.17 (because A does).

$\longmapsto$ steps involving imperative inputs are defined similarly (α-conversion is not necessary to handle the case of non-replicated inputs).

Remark 4.19. Reusing the above notations, if we write

$$\underline{A_{i_0}[v_{k_0}/x_{i_0}]} = (\nu \widetilde{d})(\nu \widetilde{g})\left(\Pi_r\, !g_r(z_r).A'_r \mid \Pi_s(!)b_s(u_s).B'_s \mid \Pi_t \overline{m}_t\langle w_t\rangle\right), \text{ we have}$$

$$A' \equiv (\nu \widetilde{c})(\nu \widetilde{f})\left(\Pi_i\, !f_i(x_i).A_i \mid \mid \Pi_j(!)a_j(y_j).B_j \mid \mid \Pi_{k\neq k_0}\overline{n}_k\langle v_k\rangle \mid \underline{A_{i_0}[v_k/x_{i_0}]}\right)$$

(thus in particular $A \longrightarrow A'$, if we view A and A' as 'plain' processes).

Example 4.20. Coming back to the expanded process introduced in Example 4.18, we have

$$A_0 \longmapsto (\nu b)(\nu f_1, f_2)\left(\, !f_1(x).a(y).\overline{x}\langle y\rangle \mid !f_2(z).(\overline{f}_1\langle z\rangle|\overline{f}_1\langle z\rangle) \mid \overline{f}_1\langle b\rangle \mid \overline{f}_1\langle b\rangle\,\right).$$

Typing. We refer to the notation used in Definition 4.17 to define, for an expanded process A,

$$\lfloor A\rfloor \stackrel{def}{=} (\nu \widetilde{a})\,(\nu f_1)\left(\, !f_1(x_1).A_1 \mid \cdots \mid (\nu f_p)\,(!f_p(x_p).A_p \mid \Pi_j(!)a_j(y_j).B_j \mid \Pi_k \overline{n}_k\langle v_k\rangle\ )\ldots\right).$$

We say that an expanded process A is *typable* if $\Gamma \diamond \_ \vdash \lfloor A\rfloor : w$ can be inferred using the rules of Figure 4 for some Γ and w, in such a way that the f_is (resp. a_js) appear as functional (resp. imperative) names in the typing derivation – we say that a name *appears as functional* if it appears at some point in the isolated part of the typing context; otherwise, we say that the name *appears as imperative*.

Example 4.21. If we consider the (typable) expanded process A_0 from Example 4.18, we have

$$\lfloor A_0\rfloor = (\nu b)(\nu f_1)\,(!f_1(x).a(y).\overline{x}\langle y\rangle \ \mid\ (\nu f_2)\,(!f_2(z).(\overline{f}_1\langle z\rangle|\overline{f}_1\langle z\rangle) \ \mid\ \overline{f}_2\langle b\rangle)\,).$$

Proposition 4.22 (subject reduction). If A is typable and $A \longmapsto A'$, then A' is typable.

Proof (sketch). This property is proved along the lines of Theorem 2.11: we rely on a substitution lemma, and reason by cases on the kind of communication involved in $A \longmapsto A'$. □

From Curry-style processes to expanded processes. As announced above, expanded processes are somehow 'in between' Curry-style processes and processes with functional definitions. We first explain how we can relate Curry-style processes and expanded processes. In order to do this, it is suitable to disallow α-conversion in the definition of $\equiv$: this gives rise to relation $\equiv^{\circ}$, in Definition 4.24.

Definition 4.23 (static context). *Static contexts* are given by the following grammar:

$$C \quad ::= \quad [\,] \mid C|P \mid (\nu n)\,C.$$

Given a static context C and a process P, we write $C[P]$ to denote the process obtained by inserting process P instead of the hole in C. We implicitly assume, when using this notation, that $C[P]$ abides the Barendregt convention.

Definition 4.24. We introduce the following relations:

— $=_\alpha$ stands for α-conversion.
— $\equiv^\circ$ is defined as $\equiv$, but without including α-conversion; $\equiv^\circ$ is only defined between processes that abide the Barendregt convention.
— $Q \overset{\circ}{\to} Q'$ holds whenever $Q \equiv^\circ C[Q_1]$ and $Q' \equiv^\circ C[Q_1']$ for some static context C, with
  – either $Q_1 = n(x).P_1 \mid \overline{n}\langle v\rangle$ and $Q_1' = P_1[v/x]$;
  – or $Q_1 = !n(x).P_1 \mid \overline{n}\langle v\rangle$ and $Q_1' = !n(x).P_1 \mid \underline{P_1[v/x]}$, where $\underline{P_1[v/x]}$ denotes a copy of $P_1[v/x]$ that has been adequately renamed in order for Q_1' to satisfy the Barendregt convention. In the case where $P_1 = \mathbf{0}$, we have $Q_1' = !n(x).P_1$.

Lemma 4.25. $P \longrightarrow P'$ iff $P \overset{\circ}{\to} =_\alpha P'$.

In the remainder of this section, we shall manipulate reduction relations and typing judgements involving processes that belong to different calculi: Curry-style processes and expanded processes are typed using $\Gamma \diamond f : \mathsf{o}^k T \vdash \cdot$, while Church-style processes are typed using $\Gamma \vdash \cdot$. Reduction between Curry-style processes is written $\longrightarrow$, but we reason mostly about $\overset{\circ}{\to}$ (by Lemma 4.25); reduction between expanded processes is written $\longmapsto$, and notation $\longrightarrow$ is also used for reduction between Church-style processes.

Lemma 4.26. Suppose that $C[P] \equiv^\circ C'[P]$, where C, C' are static contexts. Then for any Q, $C[Q] \equiv^\circ C'[Q]$.

Lemma 4.27 (reduction). If $P \overset{\circ}{\to} P'$ and $P \equiv^\circ A$, then there exists A' such that $A \longmapsto A'$ and $P' \equiv^\circ A'$.

Proof. Suppose $P \equiv^\circ A$, A being of the form given in Definition 4.17.
Suppose also $P \overset{\circ}{\to} P'$. Then, according to Definition 4.24, we have

$$P \equiv^\circ C[!n(x).P_1 \mid \overline{n}\langle v\rangle] \quad \text{and} \quad C[!n(x).P_1 \mid \underline{P_1[v/x]}] \equiv^\circ P'$$

(the case of a communication involving a non-replicated input being treated similarly). Because A is written like in Definition 4.17, and since $P \equiv^\circ A$, n is either one of the f_is or one of the a_js, and we have $A \equiv^\circ C'[!n(x).Q_1 \mid \overline{n}\langle v\rangle]$, where C' is of the form $(\nu\widetilde{c})(\nu\widetilde{f})([\,] \mid Q)$ for some Q, $Q_1 \equiv^\circ P_1$, and moreover, writing A in this form involves only applications of associativity and commutativity of $\mid$ for the top-level parallel components of A. We thus have $P \equiv^\circ C[!n(x).P_1 \mid \overline{n}\langle v\rangle] \equiv^\circ C'[!n(x).P_1 \mid \overline{n}\langle v\rangle] \equiv^\circ A$.

We moreover have $A \longmapsto A'$, following the definition of $\longmapsto$ given above. Like in Remark 4.19 (which actually holds also with $\equiv^\circ$ instead of $\equiv$), we can show that $A' \equiv^\circ C'[!n(x).P_1 \mid \underline{P_1[v/x]}]$. This allows us to apply Lemma 4.26 to deduce $A' \equiv^\circ C[!n(x).P_1 \mid \underline{P_1[v/x]}]$, and hence $A' \equiv^\circ P'$. □

The result above holds in an untyped setting. Typability can also be transferred to expanded processes:

1 $P_1 \mid (\nu b)\, P_2 \rightsquigarrow (\nu b)\,(P_1 \mid P_2)$

2 $((\nu b)\, P_1) \mid P_2 \rightsquigarrow (\nu b)\,(P_1 \mid P_2)$

3 $(\nu f)(\nu b)\, P_1 \rightsquigarrow (\nu b)(\nu f)\, P_1$

4 $P_1 \mid !f(x).P_2 \rightsquigarrow !f(x).P_2 \mid P_1$

5 $M \mid (\nu f)P_2 \rightsquigarrow ((\nu f)P_2) \mid M$

6 $\overline{n}\langle v\rangle \mid (!)b(x).P_1 \rightsquigarrow (!)b(x).P_1 \mid \overline{n}\langle v\rangle$

7 $M \mid (\nu f)\,(!f(x).P_2 \mid P_3) \rightsquigarrow (\nu f)\,(!f(x).P_2 \mid M \mid P_3)$

8 $(\nu f)(!f(x).P_1 \mid P_2) \mid (\nu f')(!f'(x').P_3 \mid P_4) \rightsquigarrow (\nu f, f')(!f(x).P_1 \mid !f'(x').P_3 \mid P_2 \mid P_4)$

· f, f' range over $f_1, \ldots, f_p$
· b ranges over $b_1, \ldots, b_q$
· n, v range over $f_1, \ldots, f_p, b_1, \ldots, b_q$
· M ranges over processes of the form $(!)b(x).P_1$ or $\overline{n}\langle v\rangle$

Fig. 5. Rules to rewrite a process into an expanded process.

Lemma 4.28 (typability). Suppose $\Gamma \diamond \_ \vdash P : w$. Then there is A s.t. $P \equiv^{\circ} A$ and A is typable.

Proof. Consider $\mathcal{D} : \Gamma \diamond \_ \vdash P : w$, a typing derivation for P.

We suppose that P contains no subterm of the form $Q|\mathbf{0}$ or $\mathbf{0}|Q$ (otherwise it is rather simple to derive typability for a process where such occurrences of $\mathbf{0}$ have been erased).

We moreover suppose w.l.o.g. that names that appear as functional (resp. imperative) names in $\mathcal{D}$ are called $f_1, \ldots, f_p$ (resp. $b_1, \ldots, b_q$), and, as usual, that P obeys the Barendregt convention. So the set of names occurring in P (free or bound) is $f_1, \ldots, f_p, b_1, \ldots, b_q$.

We transform P by applying the rewriting rules of Figure 5 modulo associativity of parallel composition (nota: commutativity is handled, in a controlled way, using the rules themselves). Rules 1–3 are *extrusion rules*, to pull up restrictions on imperative names. Rules 4–6 are *commutativity rules*, that rearrange processes following the structure of expanded processes. Finally, rules 7 and 8 are *intrusion rules*, which are used to push processes inside (what plays the rôle of) functional definitions. Note that by hypothesis, no α-conversion is necessary to apply the extrusion and intrusion rules.

We have the following properties:

1. $\rightsquigarrow \subseteq \equiv^{\circ}$ (all rewrite rules preserve $\equiv^{\circ}$).
2. Rewriting terminates.
   This can be proved by defining an appropriate measure on processes. Intuitively, rules pull restrictions upwards (and restrictions on imperative names above restrictions on functional names), move replicated inputs on functional names to the left, and messages to the right.
3. Rewriting yields an expanded process.
   Note that rewriting is not deterministic: for instance the first two rules yield a critical pair (from $(\nu b_1)P_1 \mid (\nu b_2)P_2$) that is not joinable.
4. Rewriting preserves typability. This is established by examining the rules of Figure 5. The first two rules, as well as the commutation rules, are handled easily. Rule 3 has the effect of 'anticipating' a change in the isolated name. When rewriting using rule 4, we switch from one rule to type parallel composition to the other. Finally, intrusion rules 'transport' a whole typing derivation. In doing so, the isolated part of the typing context at top-level does not change: in rule 7, M does not make use of it because it is not an input on a functional name; in rule 8, a whole functional definition (of f') gets transported inside the definition of f. □

Example 4.29. Consider $P_0 = (\boldsymbol{\nu} f_1)(\boldsymbol{\nu} b)(\boldsymbol{\nu} f_2)\,(!f_1(x).a(y).\overline{x}\langle y\rangle \mid \overline{f}_2\langle b\rangle) \mid !f_2(z).(\overline{f}_1\langle z\rangle|\overline{f}_1\langle z\rangle))$. Then $P \rightsquigarrow A_0$, where A_0 is the expanded process from Example 4.18.

Corollary 4.30. If A is typable, $P \equiv^{\circ} A$ and $P \xrightarrow{\circ} P'$, then there is a typable A' s.t. $P' \equiv^{\circ} A'$ and $A \longmapsto A'$.

Proof. Follows from Lemma 4.27 and Proposition 4.22. □

From expanded processes to church-style processes. We now explain how we can transform expanded processes into Church-style processes, by introducing functional definitions, in such a way as to preserve reductions.

Definition 4.31. Let A be an expanded process s.t. $\mathcal{D} : \Gamma \diamond \_ \vdash \lfloor A \rfloor : w$ for some $\mathcal{D}, \Gamma, w$. We suppose w.l.o.g. that A abides the Barendregt convention, so that every name that appears in A (and thus in $\mathcal{D}$) can be associated to a type, given by $\mathcal{D}$. We write $\Gamma_{\mathcal{D}}(n)$ for the type associated to n this way. $\Gamma_{\mathcal{D}}$ allows us to define the Church-style process

$$\lceil A \rceil \stackrel{def}{=} (\boldsymbol{\nu}\widetilde{a})\,\mathtt{def}\ f_1 = (x_1).A_1\ \mathtt{in}\ \ldots \mathtt{def}\ f_p = (x_p).A_p\ \mathtt{in}\ \left(\Pi_j(!)a_j(y_j).B_j \mid \Pi_k \overline{n}_k\langle v_k\rangle\right),$$

which belongs to the grammar of Section 4.1.

Example 4.32. Referring again to Example 4.18, we have

$$\lceil A_0 \rceil = (\boldsymbol{\nu} b)\,\mathtt{def}\ f_1 = (x).a(y).\overline{x}\langle y\rangle\ \mathtt{in}\ \mathtt{def}\ f_2 = (z).(\overline{f}_1\langle z\rangle|\overline{f}_1\langle z\rangle)\ \mathtt{in}\ \overline{f}_2\langle b\rangle.$$

As another example, we can consider the process $P_1 = (\boldsymbol{\nu} u)\,(!u(x).\overline{x} \mid (\boldsymbol{\nu} v)\,(!v.\overline{u}\langle t\rangle \mid \overline{u}\langle v\rangle))$, mentioned in Example 3.4 above. We observe $P_1 \rightsquigarrow A_1 = (\boldsymbol{\nu} u, v)\,((!u(x).\overline{x} \mid !v.\overline{u}\langle t\rangle \mid \overline{u}\langle v\rangle)$, and A_1 can be typed by treating u and v as functional names. We then have $\lfloor A_1 \rfloor = P_1$ and $\lceil A_1 \rceil = \mathtt{def}\ u = (x).\overline{x}\ \mathtt{in}\ \mathtt{def}\ v = ().\overline{u}\langle t\rangle\ \mathtt{in}\ \overline{u}\langle v\rangle$.

Lemma 4.33 (typability). If A is typable, then $\Gamma \vdash \lceil A \rceil : w$ for some Γ, w.

Proof (sketch). We have by hypothesis a derivation $\mathcal{D} : \Gamma \diamond \_ \vdash \lfloor A \rfloor : w$. We reason by induction on $\mathcal{D}$, and construct a typing derivation for $\lceil A \rceil$. The essential step consists in transforming a sequence of deductions to type a restriction and an input on a functional name into an application of the rule for functional definitions from Section 4.1. □

Lemma 4.34 (reduction). If $A \longmapsto A'$, then $\lceil A \rceil \longrightarrow \lceil A' \rceil$.

Proof (sketch). First remark that by Proposition 4.22, A' is typable, which allows us to write $\lceil A' \rceil$. To prove $\lceil A \rceil \longrightarrow \lceil A' \rceil$, we reason by cases over the rule that is used to derive the $\longmapsto$ transition, and construct the corresponding evaluation context for $\longrightarrow$ in the calculus of Section 4.1. After $\lceil A \rceil$'s reduction, we obtain a process that is structurally congruent to $\lceil A' \rceil$, along the lines of the situation described in Remark 4.19. □

We can now establish termination for the type system of Figure 4.

Theorem 4.35 (termination). If $\Gamma \diamond \_ \vdash P : w$, then P terminates.

Proof. We reason by contradiction, and suppose that there exists an infinite sequence of reductions starting from $P = P_0 \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \ldots$.

Applying Lemma 4.25 to $P_0 \longrightarrow P_1$ gives $P_0 \overset{\circ}{\rightarrow} P_1' =_\alpha P_1 \longrightarrow P_2$ for some P_1', thus $P_1' \longrightarrow P_2$, and, again by Lemma 4.25, we get $P_0 \overset{\circ}{\rightarrow} P_1' \overset{\circ}{\rightarrow} P_2' =_\alpha P_2$: we obtain like this an infinite sequence of $\overset{\circ}{\rightarrow}$-reductions starting from $P_0 = P$.

By Lemma 4.28, there is A s.t. A is typable and $P \equiv^\circ A$. We then rely on Corollary 4.30 to deduce an infinite sequence of $\longmapsto$-reductions starting from A, and involving typable expanded processes. Lemma 4.34 then allows us to deduce an infinite sequence of reductions starting from $\lceil A \rceil$, which contradicts Theorem 4.15. □

Notice that the hypothesis $\Gamma \diamond \_ \vdash P : w$ in the statement of Theorem 4.35 is not restrictive: if $\Gamma \diamond f : \mathsf{o}^k T \vdash P : w$, then $\Gamma \diamond \_ \vdash (\boldsymbol{\nu} f)P : w$, thus $(\boldsymbol{\nu} f)P$ terminates, and so does P.

Beyond soundness of the type system of Figure 4, the results we have established show a rather close correspondence between Curry-style and Church-style processes. Expanded processes play a pivotal rôle in this correspondence: if the $\equiv$-equivalence class of a Curry-style process contains a typable expanded process, then the latter has a counterpart in the Church-style calculus, and the reductions of these processes match each other in the two calculi. From this point of view, the calculus of Section 4.2 provides a framework for type inference for the calculus of Section 4.1.

5. Type inference

We now study type inference for the system of Section 2, 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 2.4). Type inference for level-based systems has been studied in Demangeon *et al.* (2007), 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.

5.1. *Type inference for termination in the localised π-calculus*

In this section, we study a subsystem of the one we have defined in Section 2. 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.

Technically, we introduce Lπ by restricting to a *subset* of the i/o-types only, while keeping the same typing rules. We adopt the following restricted syntax for types (that we call 'S-types'):

$$S \quad ::= \quad \mathsf{o}^k S \mid \mathbb{U},$$

and typing environments contain hypotheses of the form $a : S$ or $a : \sharp^k S$ only. Hypotheses of the latter form can only be added using the typing rule for restriction. This entails that in typing derivations, we can only use types of the form $\sharp^k S$ (in the rule for restriction), $\mathsf{i}^k S$ (in the rules for input prefixes, thanks to subtyping), and S. We write $\Gamma \vdash^{L\pi} P : w$ for

the resulting typing judgement for Lπ processes. This means in particular that to build $\vdash^{L\pi}$ derivations, subtyping can be used only to exploit flexibility on levels, but not, e.g. to deduce $\Gamma \vdash a : \mathsf{o}^k\sharp^n S$ from $\Gamma \vdash a : \mathsf{o}^k\mathsf{o}^n S$, as this would mean violating the restriction on types introduced above.

Note in passing 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 3.2, and the encoding of the λ-calculus – belong to Lπ. Obviously, typability for $\vdash^{L\pi}$ entails typability for $\vdash$, hence termination.

From the point of view of implementations, the restriction to Lπ makes sense. For instance, the language JoCaml Mandel and Maranget (2012) 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 Ericsson Computer Science Laboratory (2012) can also be viewed as obeying 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 (like in the `def` construct of Section 4.1). Additionally, in cryptographic implementations of the π-calculus, disallowing the emission of the read capability fixes the problem of preservation of forward secrecy (Abadi 1999).

We now describe a type inference procedure for $\vdash^{L\pi}$.

We first check typability when levels are not taken into account. For this, we rely on a type inference algorithm for simple types (Vasconcelos and Honda 1993), together with a simple syntactical check to verify that no received name is used in input. If this first step fails, the process is rejected. Otherwise, we replace $\sharp T$ types with $\mathsf{o}S$ 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 $a(x).\overline{b}\langle x\rangle$). As an example, consider the processes $P_1 = a(x).\overline{x}\langle t\rangle$ and $P_2 = a(x).x(y).\mathbf{0}$. The inference procedure for simple types deduces $a : \sharp\sharp\alpha$, $x : \sharp\alpha$, in both cases, and, for P_1, $t : \alpha$, where α is a type variable. However, P_2 is deemed not typable in Lπ, since x is used as an input. Lastly, for P_1 we modify the types to obtain $a : \sharp\mathsf{o}T$ and $x : \mathsf{o}T$.

What remains to be done is to find out whether types can be decorated with levels in order to ensure termination.

We first introduce an auxiliary typing judgement for processes, noted $\Gamma \vdash^{L\pi}_{\mathrm{TI}} P : w$. The rules for $\vdash^{L\pi}_{\mathrm{TI}}$ are the same as for $\vdash$, except for the rules involving prefixes, which are the following:

$$\frac{\Gamma(a) = \sharp^k S \qquad \Gamma, x : S \vdash^{L\pi}_{\mathrm{TI}} P : w}{\Gamma \vdash^{L\pi}_{\mathrm{TI}} a(x).P : w} \qquad \frac{\Gamma(a) = \sharp^k S \qquad \Gamma, x : S \vdash^{L\pi}_{\mathrm{TI}} P : w \qquad k > w}{\Gamma \vdash^{L\pi}_{\mathrm{TI}} !a(x).P : 0}$$

$$\frac{\Gamma(a) = \sharp^k S \text{ or } \Gamma(a) = \mathsf{o}^k S \qquad \Gamma \vdash v : S}{\Gamma \vdash^{L\pi}_{\mathrm{TI}} \overline{a}\langle v\rangle : k}$$

(the definition of the typing judgement $\Gamma \vdash a : S$ is left unchanged). Notice that in these rules, we *read* the type of the subject (a) in the typing context, thus disallowing the use of the subsumption rule.

Lemma 5.1. For any Γ, P and w, $\Gamma \vdash^{L\pi} P : w$ iff $\Gamma \vdash^{L\pi}_{\mathrm{TI}} P : w$.

Proof. The implication from right to left is rather easy, since $\Gamma(a) = \sharp^k S$ implies $\Gamma \vdash a : \mathsf{i}^k S$.

To establish the converse, we reason by induction on the height of the derivation of the judgement $\Gamma \vdash^{L\pi} P : w$. We discuss the case of replicated input, that is, $P = !a(x).P_1$. Because P is typable, we have $\Gamma \vdash a : \mathsf{i}^n S$ for some n, S. Since we are in Lπ, necessarily $\Gamma(a) = \sharp^k S'$ for some k, S', and $\sharp^k S' \leqslant \mathsf{i}^n S$. We also have by hypothesis $\Gamma, x : S \vdash^{L\pi} P_1 : w$ with $w < n$. Since $\sharp^k S' \leqslant \mathsf{i}^n S$, we have $S' \leqslant S$, and by narrowing (Proposition 2.8), we obtain $\Gamma, x : S' \vdash^{L\pi} P_1 : w'$ with $w' \leqslant w$, which by induction gives $\Gamma, x : S' \vdash^{L\pi}_{\mathrm{TI}} P_1 : w'$. Moreover, $\sharp^k S' \leqslant \mathsf{i}^n S$ also yields $n \leqslant k$, thus $w' < k$. We can thus finally derive $\Gamma \vdash^{L\pi}_{\mathrm{TI}} !a(x).P_1 : 0$. □

In view of this lemma, we look for an assignment of levels according to $\Gamma \vdash^{L\pi}_{\mathrm{TI}} \cdot$, which is less 'liberal' than $\Gamma \vdash \cdot$. As mentioned above, we suppose w.l.o.g. that we have a term P abiding the Barendregt convention (all bound names are pairwise distinct, and distinct from all free names). We define the following sets of names:

— names(P) stands for the set of all names, free and bound, of P;
— bn(P) is the set of names that appear bound (either by restriction or by input) in P;
— 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);
— 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 $(\nu a)\, Q$ for some Q);
— chan(P) is the set of names that are used as communication channels ($a \in \mathrm{chan}(P)$ iff P has a subterm of the form $\overline{a}\langle v \rangle$ for some v, or of the form $(!)a(x).Q$ for some x, 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 either the prefix $a(x)$ or the prefix $!a(x)$: we write in this case $a = \mathrm{father}(x)$ ($a \in \mathrm{fn}(P) \cup \mathrm{res}(P)$, that is, $a \notin \mathrm{rcv}(P)$, because we are in Lπ).

We build a graph as follows:

— For every name in names(P), create a node labelled by $\{n\}$.
— For every name $n \in \mathrm{chan}(P)$, create a node labelled by $\{\mathrm{son}(n)\}$. Intuitively, if n has type $\sharp^k S$ or $\mathsf{o}^k S$, then son(n) has type S.
— For every $x \in \mathrm{rcv}(P)$, let $a = \mathrm{father}(x)$, merge the nodes whose labels contain x and son(a), that is, add x to the label of the node whose label contains son(a) (note that several names in rcv(P) can have the same father, but every such name has a unique father).

son(n) is used in the type inference procedure to represent the fact that the type of names received on n must match the type obtained by removing the topmost capability from n's type. In other words, if n has type $\sharp^k S$ and the process contains a subterm of the form $n(x).Q$, then x should have type S, which we represent by associating son(n) and x to the same node in the graph.

Example 5.2. 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 9 nodes with their labels:

$$\{a\}, \{\mathrm{son}(a), x, y\}, \{b\}, \{c\}, \{\mathrm{son}(c)\}, \{d\}, \{\mathrm{son}(d), z\}, \{\mathrm{son}(x)\}, \{\mathrm{son}(y)\}.$$

The next step is to insert edges in our graph, to represent the constraints between levels. This is motivated by the following observations about $\vdash^{L\pi}_{\mathrm{TI}}$.

Lemma 5.3. If $\Gamma \vdash^{L\pi}_{\mathrm{TI}} \overline{a}\langle v\rangle$, then either $v = \star$, or there exist k, k', n, S, S' such that $\Gamma(a) = \sharp^n \mathsf{o}^k S$ or $\Gamma(a) = \mathsf{o}^n \mathsf{o}^k S$, and moreover $\Gamma(v) = \mathsf{o}^{k'} S'$, $S \leqslant S'$ and $k' \leqslant k$.

Lemma 5.4. Suppose we have a typing derivation for $\Gamma \vdash^{L\pi}_{\mathrm{TI}} !a(x).Q : 0$. Then for any subterm of Q of the form $\overline{n}\langle m\rangle$ that does not occur under a replication in Q, if $\Gamma(a) = \sharp^k S$ and n has type $\mathsf{o}^{k'} S'$ or $\sharp^{k'} S'$ for some k', S' in the typing derivation, we have $k' < k$.

This leads us to add the following (directed) edges to our graph:

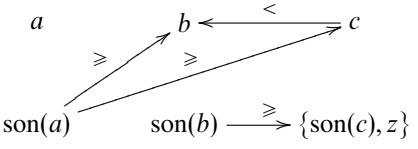
— For every output of the form $\overline{n}\langle m\rangle$, we insert an edge labelled with '$\geqslant$' from $\mathrm{son}(n)$ to m.
— For every subterm of P of the form $!a(x).Q$, and for every output of the form $\overline{n}\langle m\rangle$ that occurs in Q without occurring under a replication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

The first (resp. second) kind of edges is justified by Lemma 5.3 (resp. Lemma 5.4). As mentioned above, the edges represent constraints between levels. Consider then an output $\overline{n}\langle m\rangle$: Lemma 5.3 highlights the dependency between the level of the type carried by m and the level of n, hence the edge from $\mathrm{son}(n)$ to m. Similarly, for the process $!a(x).P$, Lemma 5.4 justifies the $\xrightarrow{>}$ edge between a and an unguarded output in its continuation P.

Example 5.5. 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 to the nodes of the graph. The assignment should respect the constraints expressed by the labels of the edges. This is possible as long as there are no cycles involving at least one $\xrightarrow{>}$ edge.

At the beginning, all nodes of the graph are unlabelled. We label them using natural numbers according to the following procedure:

1. We go through all nodes of the graph, and collect those nodes that have no outgoing edge leading to an unlabelled node in a set $\mathcal{S}$.

2. If $\mathcal{S}$ is not empty, we label every node n in $\mathcal{S}$ as follows: we start by setting n's label to 0. We then examine all outgoing edges of n. For every $n \overset{\geqslant}{\longrightarrow} m$, we replace n's label, say k, with $\max(k, k')$, where k' is m's label, and similarly for $n \overset{>}{\longrightarrow} m$ edges, with $\max(k, k'+1)$. We then empty $\mathcal{S}$, and start again at step 1.
3. If $\mathcal{S} = \varnothing$, 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 $\overset{>}{\longrightarrow}$ edge, the procedure stops and reports failure. Otherwise, the cycle involves only $\overset{\geqslant}{\longrightarrow}$ 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 5.6. On the graph of Example 5.5, the procedure first assigns level 0 to nodes a, b and $\{\mathrm{son}(c), z\}$. In the second iteration, $\mathcal{S} = \{\mathrm{son}(b), c\}$; 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 $b : \mathsf{o}^0\mathsf{o}^0 T, c : \sharp^1\mathsf{o}^0 T, a : \mathsf{o}^0\mathsf{o}^1\mathsf{o}^0 T$ for the process of Example 5.5.

Example 5.7. We illustrate the whole inference procedure on the process

$$P = !a(x).!b(t).\overline{x} \mid !d.(\overline{b}\langle c\rangle \mid \overline{c}) \mid \overline{a}\langle c\rangle).$$

We have the following steps:

— the inference procedure for simple types yields the following types of the names in P:

$$a, b : \sharp\sharp\mathbb{U} \qquad\qquad c, d, x, t : \sharp\mathbb{U}$$

with the sets: $\mathrm{names}(P) = \{a, b, c, d, x, t\}$, $\mathrm{bn}(P) = \mathrm{rcv}(P) = \{x, t\}$, $\mathrm{fn}(P) = \{a, b, c, d\}$ and $\mathrm{chan}(P) = \{a, b, c, d, x\}$;

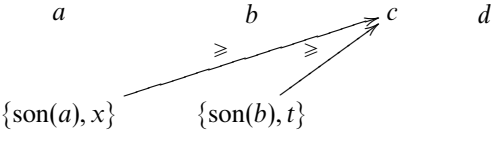
— we check that none of the names in the set $\mathrm{rcv}(P)$ is used in input, and adjust the types to Lπ:

$$a, b : \sharp\mathsf{o}\mathbb{U} \qquad\qquad c, x, t : \mathsf{o}\mathbb{U} \qquad\qquad d : \sharp\mathbb{U}$$

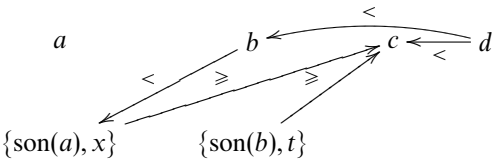
— the next step consists in building the graph. For each name m we create a node $\{m\}$ and for each $n \in \mathrm{chan}(P)$ we introduce the node $\{\mathrm{son}(n)\}$. Then the nodes $\{\mathrm{son}(n)\}$ and $\{m\}$ are merged if m is received on channel n. This yields the following nodes:

$$\{a\}, \{\mathrm{son}(a), x\}, \{b\}, \{\mathrm{son}(b), t\}, \{c\}, \{\mathrm{son}(c)\}, \{d\}, \{\mathrm{son}(d)\}, \{\mathrm{son}(x)\}.$$

In the sequel, for simplicity we ignore the label $\mathrm{son}(m)$ if m carries the unit type. This means that we do not mention $\mathrm{son}(c)$, $\mathrm{son}(x)$ and $\mathrm{son}(d)$ (note that $t \notin \mathrm{chan}(P)$). Next we add the edges of our graph. For every output $\overline{n}\langle m\rangle$ there is an edge between $\mathrm{son}(n)$ and m. For P this gives:



An edge is also added whenever we have an unguarded output under an replicated input. In P this occurs in the subprocesses $!b(t).\overline{x}$ and $!d.(\overline{b}\langle c\rangle \mid \overline{c})$. The construction of the graph is now complete:



— in the last step we verify that the graph contains no oriented cycle. When this is the case, we assign levels to our types so that the constraints on the edges are satisfied. We can finally deduce the following types:

$$a : \sharp^1\mathsf{o}^1\mathbb{U} \qquad b : \sharp^2\mathsf{o}^1\mathbb{U} \qquad c, x, t : \mathsf{o}^1\mathbb{U} \qquad d : \sharp^3\mathbb{U}$$

We let the *size* of a process P be defined as the number of input prefixes plus the number of output messages in P.

Proposition 5.8. The procedure described above terminates in time polynomial in the size of P.

Proof. In the first step, a graph is built in polynomial time in terms of the size of the process. The size of the graph is linear in the size of names(P). The second step, that computes the level assignment, is polynomial in the size of the graph. □

5.2. *Discussion: inferring i/o-types*

If we consider type inference for the whole system of Section 2, the situation becomes more complex than in Section 5.1. 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 #' are preferable because they are more precise). $\mathsf{io}T$ is moreover a supertype of all other types mentioned above. So in a sense, this type for a could be used to describe all possible types that allow one to typecheck $a(x).\overline{x}\langle t\rangle$. It turns out that 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 $\mathsf{i}T \leqslant \uparrow T$ and $\mathsf{o}T \leqslant \uparrow T$).

Igarashi and Kobayashi (2000) presents a type inference algorithm for (an enrichment of) i/o-types, where such a capability $\uparrow$ is added to the system of Pierce and Sangiorgi (1996) (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:

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

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 $a : \mathsf{ii}\,T, b : \mathsf{io}\,T$. 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' (one can observe that this does not prevent from comparing names, which may be useful to study behavioural equivalences Hennessy and Rathke (2004)): 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 (or compare it with another name). 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 $a : \mathsf{i}\sharp T, b : \mathsf{io}\,T, c : \mathsf{oo}\,T$. 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 Igarashi and Kobayashi (2000). It is 'minimal,' in the terminology of Igarashi and Kobayashi (2000). 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 $\mathsf{i}^k T \leqslant \uparrow T$ and $\mathsf{o}^k T \leqslant \uparrow T$. 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.

6. 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 the type systems for termination of processes presented in Deng and Sangiorgi (2006). As we discuss in Section 3.3, 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 can be studied further. It would be interesting to analyse how the procedure of Section 5.1 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 4.2 is a challenging question, essentially because making the distinction between functional and imperative names would be part of the inference process (contrarily to the setting of Section 4.1, where the syntax of processes provides this information).

Another point worth studying further, with possible links with the implementation of the type systems of Section 4 and type inference, is the approach we used to establish a correspondence between the Curry-style and the Church-style type systems in Section 4.2.2. It might be interesting to generalize this technique, and also to see whether the way we handle structural congruence in the proof (see, e.g. Definition 4.24 and Figure 5) can give insights for the representation of processes on machine.

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,' and ANR-2010-BLANC-0305-02 'PiCoq.'

References

Abadi, M. (1999) Protection in programming-language translations. In: Secure Internet Programming, Security Issues for Mobile and Distributed Objects. *Springer Lecture Notes in Computer Science* **1603** 19–34.

Cristescu, I. and Hirschkoff, D. (2011) Termination in a pi-calculus with subtyping. In: Luttik, B. and Valencia, F. (eds.) Proceedings of EXPRESS'11. *Electronic Proceedings in Theoretical Computer Science* **64** 44–58.

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

Demangeon, R., Hirschkoff, D., Kobayashi, N. and Sangiorgi, D. (2007) On the complexity of termination inference for processes. In: Proceedings of TGC'07. *Springer Lecture Notes in Computer Science* **4912** 140–155.

Demangeon, R., Hirschkoff, D. and Sangiorgi, D. (2009) Mobile processes and termination. In: Semantics and Algebraic Specification. *Springer Lecture Notes in Computer Science* **5700** 250–273.

Demangeon, R., Hirschkoff, D. and Sangiorgi, D. (2010) Termination in impure concurrent languages. In: Proceedings of CONCUR'10. *Springer Lecture Notes in Computer Science* **6269** 328–342.

Deng, Y. and Sangiorgi, D. (2006) Ensuring termination by typability. *Information and Computation* **204** (7) 1045–1082.

Ericsson Computer Science Laboratory. (2012) Erlang programming language website. Available at `http://www.erlang.org`.

Hennessy, M. and Rathke, J. (2004) Typed behavioural equivalences for processes in the presence of subtyping. *Mathematical Structures in Computer Science* **14** (5) 651–684.

Igarashi, A. and Kobayashi, N. (2000) Type reconstruction for linear π-calculus with i/o subtyping. *Information and Computation* **161** (1) 1–44.

Kobayashi, N. and Sangiorgi, D. (2010) A hybrid type system for lock-freedom of mobile processes. *ACM Transactions on Programming Languages and Systems* **32** (5) 1–49.

Mandel, L. and Maranget, L. (2012) The JoCaml programming language. Available at `http://jocaml.inria.fr`.

Pierce, B. C. and Sangiorgi, D. (1996) Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science* **6** (5) 409–453.

Sangiorgi, D. (2006) Termination of processes. *Mathematical Structures in Computer Science* **16** (1) 1–39.

Sangiorgi, D. and Walker, D. (2001) *The π-Calculus: A Theory of Mobile Processes*, Cambridge: Cambridge University Press.

Vasconcelos, V. T. and Honda, K. (1993) Principal typing schemes in a polyadic pi-calculus. In: Proceedings of CONCUR'93. *Springer Lecture Notes in Computer Science* **715** 524–538.

Yoshida, N., Berger, M. and Honda, K. (2004) Strong normalisation in the Pi-calculus. *Information and Computation* **191** (2) 145–202.