Termination in impure concurrent languages

Romain Demangeon[1], Daniel Hirschkoff[1], and Davide Sangiorgi[2]

[1] ENS Lyon, Université de Lyon, CNRS, INRIA, France
[2] INRIA/Università di Bologna, Italy

Abstract. An impure language is one that combines functional and imperative constructs. We propose a method for ensuring termination of impure concurrent languages that makes it possible to combine term rewriting measure-based techniques with traditional approaches for termination in functional languages such as logical relations. The method can be made parametric on the termination technique employed on the functional part; it can also be iterated.
We illustrate the method in the case of a π-calculus with both functional and imperative names, and show that, with respect to previous approaches to termination, it allows us to extend considerably the set of processes that can be handled.
The method can also be applied to sequential languages, e.g., λ-calculi with references.

1 Introduction

In this paper, an impure language is one that combines functional and imperative constructs; for instance, a λ-calculus with references, or a π-calculus with functional names.
The π-calculus is naturally imperative. A π-calculus name may however be considered functional if it offers a service immutable over time and always available. Special laws, such as copying and parallelisation laws, are only valid for functional names. As a consequence, the distinction between functional and imperative names is often made in applications. Examples are intermediate target languages of compilers of programming languages based on π-calculus, such as Pict [**?**], in which the peculiarities of functional names are exploited in code optimisations. In the π-calculus language in this paper, the distinction between functional and imperative names is made directly in the syntax. The functional names are introduced with a dedicated construct akin to the 'letrec' of functional languages. The other names — the 'ordinary' π-calculus names — are called imperative. Correspondingly, a process is functional or imperative if it uses only functional or only imperative names.
The subject of this paper is termination in impure concurrent languages. There are powerful techniques that ensure termination in purely functional languages, notably realisability and logical relations, that exploit an assignment of types to terms. Attempts at transporting logical relations onto concurrent languages have had limited success. In the π-calculus, the sets of processes which have been proved to be terminating

with logical relations [**?**,**?**] are mainly "functional": they include the standard encodings of the λ-calculus into the π-calculus, but fail to capture common patterns of usage of imperative names.
The other kind of techniques for termination proposed for concurrent languages such as π-calculus uses ideas from term-rewriting systems, with a well-founded measure that decreases as the computation proceeds [**?**]. Such measure-techniques are suitable to handling imperative languages and correspondingly, in the π-calculus, imperative processes. In contrast, the techniques can handle only a limited class of functions, or functional processes. In languages richer than (the analogue of) simply-typed λ-calculi, for instance, measure-based proofs may be impossible (e.g., no measure argument representable in 2nd-order arithmetic could prove termination of the polymorphic λ-calculus, as this would amount to proving consistency of the arithmetic; termination can however be proved with logical relations). In this paper we propose a method for the termination of concurrent languages that combines the measure-based techniques of imperative languages and the logical relation techniques of functional languages. The method can be made parametric on the functional sublanguage and its termination proof.
We illustrate the combination of the two kinds of techniques from their simplest instance, in which the types used in logical relations are those of a simply-typed calculus, and the measure needed in the term-rewriting technique strictly decreases at every imperative computation step.
We explain the basic idea of the method. We first extend the imperative measure to the functional constructs, but without the ambition of ensuring termination: the control on the measure performed on functional constructs is looser than the one performed on the imperative ones. Therefore, while the overall measure of a term decreases along imperative reductions (i.e., steps stemming from imperative constructs), the measure may increase along the functional ones. Indeed the measure assignment also captures divergent terms. However, termination holds if the purely functional sublanguage is terminating. The termination of this sublanguage can be established separately. The soundness proof of the method proceeds by contradiction. The crux is to prune a divergence of an impure term into a divergence of a purely functional term.
The termination condition for the functional subcalculus (the simply-typed system) can be combined with the measure assignment. The result is a type system that refines the simply-typed one by attaching a measure to each type. The final termination result can therefore be stated as a well-typedness condition in this type system.
The method subsumes both component techniques, in the sense that the resulting terminating terms include those that can be proved terminating with logical relations or measures alone. For instance, a simply-typed purely functional term is accepted because no measure-based constraint is generated.
In the soundness proof of the method we never have to manipulate logical relations (that are used to establish termination for the functional sublanguage). We indeed take the functional sublanguage and its termination proof as a black box that we insert into our soundness proof. We can therefore generalise the method and make it parametric with respect

to the functional core and the termination technique employed on it. The method could also be iterated, that is, applied more than once (in this case all iterations but the last one are applied on a purely functional language, a subset of which is treated using measures, another subset with other means).
Further, we show that the method can be enhanced by building it on top of more sophisticated measure-based techniques.
The method can also be applied to impure *sequential* languages, e.g., λ-calculi with references. While the schema of the method is the same, the technical details, in particular the key step of pruning and the definition of stratification, are quite different, and will be presented elsewhere (the interested reader is referred to [**?**]).
We mostly give only proof sketches of important results, for lack of space. Details and additional material can be found in [**?**].

2 An impure π-calculus

The π-calculus, in its standard presentation, is imperative: the service offered by a name (its input end) may change over time; it may even disappear. There may be however names whose input end occurs only once, is replicated, and is immediately available. This guarantees that all messages sent along the name will be consumed, and that the continuation applied to each such message is always the same. In the literature these names are called *functional*, and identified either by the syntax, or by a type system. The remaining names are called *imperative*.
In the π-calculus we use, functional names are introduced using a `def` construct, akin to a "letrec" (as said above, we could as well have distinguished them using types). We use a, b for imperative names, f, g for functional names, x, y, c to range over both categories, and v, w for values; a value can be a name or $\star$ (unit). In examples, later, the grammar for values may be richer – integers, tuples, etc.; such additions are straightforward but would make the notations heavier.

$$\begin{array}{rrl} \text{(Processes)} & P ::= & P_1|P_2 \mid \mathbf{0} \mid \overline{c}\langle v\rangle.P \mid \texttt{def}\ f = (x).P_1\ \texttt{in}\ P_2 \\ & & \mid (\boldsymbol{\nu} a)\, P \mid c(x).P \mid\ !c(x).P \\ \text{(Values)} & v ::= & a \mid f \mid \star \end{array}$$

We sometimes use the CCS notation ($a.P$, $\overline{c}.P$, $\texttt{def}\ f = P_1\ \texttt{in}\ P_2$, ...) for inputs and outputs that carry the unit value $\star$; we omit trailing occurrences of $\mathbf{0}$ under a prefix. We write $\mathrm{fn}(P)$ for the set of free names of P (the binding constructs are restriction, input and `def`, the latter binding both the functional channel – f in the grammar above – and the received name – x).
An *input-unguarded output* in P is an output prefix that does occur (i) neither under an input in P (ii), nor, for any subterm of P of the form $\texttt{def}\ f = (x).P_1\ \texttt{in}\ P_2$, in the P_1 subterm.
We call $\pi_{\mathtt{ST}}$ the simply-typed version of this calculus (the discipline of simple types in π is standard – it is intrinsically contained in the type

$$(\textbf{call})\ \frac{\mathrm{capt}(\mathbf{E}_2) \cap \mathrm{fn}((x).P) = \emptyset}{\mathbf{E}_1[\mathtt{def}\ f = (x).P\ \mathtt{in}\ \mathbf{E}_2[\overline{f}\langle v\rangle.Q]] \longrightarrow \mathbf{E}_1[\mathtt{def}\ f = (x).P\ \mathtt{in}\ \mathbf{E}_2[P\{v/x\} \mid Q]]}$$

$$(\textbf{trig})\ \frac{}{\mathbf{E}[\overline{a}\langle v\rangle.Q \mid\ !a(x).P] \longrightarrow \mathbf{E}[Q \mid P\{v/x\} \mid\ !a(x).P]}$$

$$(\textbf{comm})\ \frac{}{\mathbf{E}[\overline{a}\langle v\rangle.Q \mid\ a(x).P] \longrightarrow \mathbf{E}[Q \mid P\{v/x\}]}$$

$$(\textbf{cong})\ \frac{Q \equiv P \qquad P \longrightarrow P' \qquad P' \equiv Q'}{Q \longrightarrow Q'}$$

Fig. 1. Reduction for $\pi_{\mathtt{ST}}$

system we present in Section 3). Typing in particular ensures that the subject c of inputs $c(x).P$ and $!c(x).P$ is always an imperative name.
Evaluation contexts are given by the following grammar:

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

We write $\mathbf{E}[P]$ for the process obtained by replacing the hole $[\,]$ in $\mathbf{E}$ with P, and $\mathrm{capt}(\mathbf{E})$ stands for the *captured names* of $\mathbf{E}$, that is, the names which are bound at the occurrence of the hole in $\mathbf{E}$.
Structural congruence between processes, written $\equiv$, is defined as the least congruence that is an equivalence relation, includes α-equivalence, satisfies the laws of an abelian monoid for $|$ (with $\mathbf{0}$ as neutral element), and satisfies the following two extrusion laws:

$$P \,|\, (\boldsymbol{\nu} a)\, Q \ \equiv\ (\boldsymbol{\nu} a)\,(P|Q) \quad \text{ if } a \notin \mathrm{fn}(P)$$

$$P \,|\, \mathtt{def}\ f = (x).Q_1\ \mathtt{in}\ Q_2 \ \equiv\ \mathtt{def}\ f = (x).Q_1\ \mathtt{in}\ P|Q_2 \quad \text{ if } f \notin \mathrm{fn}(P).$$

(Rules for replication or for swapping consecutive restrictions or definitions are not needed in $\equiv$.) We extend $\equiv$ to evaluation contexts.
Figure 1 presents the rules defining the reduction relation on $\pi_{\mathtt{ST}}$ processes, written $\longrightarrow$. $P\{v/x\}$ stands for the result of the application of the (capture avoiding) substitution of x with v in P. In rule (**call**), we impose that the intrusion of $P\{v/x\}$ in the context $\mathbf{E}_2$ avoids name capture. A reduction is *imperative* (resp. *functional*) when it is derived using a communication along an imperative (resp. functional) name.
A process P_0 *diverges* (or *is divergent*) if there is an infinite sequence of processes $\{P_i\}_{i\geq 0}$, such that, for all $i \geq 0$, $P_i \longrightarrow P_{i+1}$. Then P *terminates* (or *is terminating*) if P is not divergent; similarly, a set of processes is terminating if all its elements are.

Subcalculi. $\pi_{\mathtt{def}}$ is the subcalculus with only functional names (i.e., without the productions in the second line of the grammar for processes: restriction, linear and replicated inputs), and $\pi_{\mathtt{imp}}$ is the purely imperative one (i.e., without the `def` construct).

Termination constraints and logical relation proofs for λ-calculi can be adapted to $\pi_{\mathtt{def}}$. In the simply-typed case, the only additional constraint is that definitions $\mathtt{def}\ f = P\ \mathtt{in}\ Q$ cannot be recursive (f is not used in the body P). We call $\pi^1_{\mathtt{def}}$ the language obtained in this way. It is rather easy to show that $\pi^1_{\mathtt{def}}$ is included in the languages studied in [**?**,**?**], which gives:

Theorem 1 *$\pi^1_{\mathtt{def}}$ is terminating.*

Various terminating sublanguages of $\pi_{\mathtt{imp}}$ have been defined in the literature. One of the simplest ones, in [**?**], ensures termination by a straightforward measure-decreasing argument. Roughly, an integer number, called *level*, is attached to each name; then one requires that in any input $a(x).P$, the level of each input-unguarded output in P is smaller than the level of a; in other words, the output at a consumed to activate P must be heavier than any output which is released. Thus the overall measure of a process (computed on the multiset of all its input-unguarded outputs) decreases at every reduction. The measure assignment is formalised as a type system to ensure invariance under reduction. We call $\pi^1_{\mathtt{imp}}$ this terminating language, derived by adapting to $\pi_{\mathtt{imp}}$ the simplest type system from [**?**] (this type system is hardwired into the system we present in Section 3).

3 Types for termination in $\pi_{\mathtt{ST}}$

In this section we define a type system for termination that combines the constraints of the imperative $\pi^1_{\mathtt{imp}}$ with those of the functional $\pi^1_{\mathtt{def}}$. First note that a straightforward merge of the two languages can break termination. This is illustrated in

$$\mathtt{def}\ f = \overline{a}\ \mathtt{in}\ (!a.\overline{f}\,|\,\overline{a})\ ,$$

where a recursion on the name a is fed via a recursion through the functional definition of f. The process is divergent, yet the constraints in $\pi^1_{\mathtt{def}}$ and $\pi^1_{\mathtt{imp}}$ are respected (the process is simply-typed, has no recursive definition, the imperative input $!a.\overline{f}$ is well-typed as the level of a can be greater than the level of f).
Termination is guaranteed if the measure constraint imposed on imperative names is extended to the functional ones, viewing a `def` as an input. This extension would however be too naive: it would indeed affect the class of functional processes accepted, which would be rather limited (not containing for instance the calculus $\pi^1_{\mathtt{def}}$ and the process images of the simply-typed λ-calculus).
In the solution we propose, we do impose measures onto the functional names, but the constraints on them are more relaxed. This way, $\pi^1_{\mathtt{def}}$ is captured. The drawback is that the measure alone does not anymore guarantee termination. However, it does if the purely functional sublanguage (in our case $\pi^1_{\mathtt{def}}$) is terminating.

$$(\textbf{Res})\ \frac{\vdash_\Gamma P : l}{\vdash_\Gamma (\boldsymbol{\nu} a)\, P : l} \qquad (\textbf{Par})\frac{\vdash_\Gamma P_1 : l_1 \qquad \vdash_\Gamma P_2 : l_2}{\vdash_\Gamma P_1 \mid P_2 : \max(l_1, l_2)}$$

$$(\textbf{Nil})\frac{}{\vdash_\Gamma \mathbf{0} : 0} \qquad (\textbf{Out})\ \frac{\vdash_\Gamma P : l \qquad \vdash_\Gamma c : \sharp^k_\bullet\, T \qquad \vdash_\Gamma w : T}{\vdash_\Gamma \overline{c}\langle w\rangle.P : \max(k, l)}$$

$$(\textbf{In})\ \frac{\vdash_\Gamma P : l \qquad \vdash_\Gamma a : \sharp^k_{\mathtt{I}}\, T \quad \vdash_\Gamma x : T \qquad k > l}{\vdash_\Gamma a(x).P : 0} \qquad (\textbf{Rep})\ \frac{\vdash_\Gamma P : l \qquad \vdash_\Gamma a : \sharp^k_{\mathtt{I}}\, T \quad \vdash_\Gamma x : T \qquad k > l}{\vdash_\Gamma\, !a(x).P : 0}$$

$$(\textbf{Def})\ \frac{\vdash_\Gamma P_1 : l \qquad \vdash_\Gamma P_2 : l' \qquad \vdash_\Gamma f : \sharp^k_{\mathtt{F}}\, T \quad k \geq l \qquad f \notin \mathrm{fn}(P_1)}{\vdash_\Gamma \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2 : l'}$$

Fig. 2. Typing Rules for $\pi^1_{\mathtt{ST}}$

The whole constraints are formalised as a refinement of the simply-typed system in which a level is attached to types. To ease the proofs, the system is presented *à la Church*: every name has a predefined type. Thus a typing Γ is a total function from names to types, with the proviso that every type is inhabited by an infinite number of names. We write $\Gamma(x) = T$ to mean that x has type T in Γ. Types are given by:

$$T \ ::= \ \sharp^k_{\mathtt{F}}\, T \mid \sharp^k_{\mathtt{I}}\, T \mid \mathtt{unit}\ .$$

Type $\sharp^k_{\mathtt{F}}\, T$ is assigned to functional names that carry values of type T; similarly for $\sharp^k_{\mathtt{I}}\, T$ and imperative names. In both cases, k is a natural number called the *level*. We write $\sharp^k_\bullet\, T$ when the functional/imperative tag is not important.

The typing judgement for processes is of the form $\vdash_\Gamma P : l$, where l is the *level* (or weight) *of* P. It is defined by the rules of Figure 2; on values, $\vdash_\Gamma v : T$ holds if either $\Gamma(v) = T$, or $v = \star$ and $T = \mathtt{unit}$.

We comment on the definition of the type system. Informally, the level of a process P indicates the maximum level of an input-unguarded output in P. Condition $k > l$ in rules (**In**) and (**Rep**) implements the measure constraint for $\pi^1_{\mathtt{imp}}$ discussed in Section 2. In rule (**Def**) the constraint is looser: the outputs released can be as heavy as the one being consumed (condition $k \geq l$). In other words, we just check that functional reductions do not cause violations of the stratification imposed on the imperative inputs (which would happen if a functional output underneath an imperative input a could cause the release of imperative outputs heavier than a, as in the example at the beginning of this section). In the same rule (**Def**), the constraint $f \notin \mathrm{fn}(P_1)$ is inherited from $\pi^1_{\mathtt{def}}$, and forbids recursive calls in functional definitions.

We call $\pi^1_{\mathtt{ST}}$ the set of well-typed processes. Our type system subsumes the existing ones, and satisfies the subject reduction property:

Lemma 2 *We have* $\pi^1_{\mathtt{def}} \subseteq \pi^1_{\mathtt{ST}}$ *and* $\pi^1_{\mathtt{imp}} \subseteq \pi^1_{\mathtt{ST}}$.

Proposition 3 *If* $\vdash_\Gamma P : l$ *and* $P \longrightarrow P'$ *then* $\vdash_\Gamma P' : l'$ *for some* l'.

The termination proof for $\pi^1_{\mathtt{imp}}$ uses the property that at every reduction the *multiset measure* of a process (the multiset of the levels of input-unguarded outputs in the process) decreases. In $\pi^1_{\mathtt{ST}}$, this only holds for imperative reductions: along functional reductions the overall weight may actually increase. It would probably be hard to establish termination of $\pi^1_{\mathtt{ST}}$ by relying only on measure arguments.

Inferring termination. In view of the results in [**?**], we believe that a type inference procedure running in polynomial time can be defined for the type system of Figure 2. This is possible as long as typability (and hence termination) can be inferred in polynomial time for the core calculus (here, $\pi^1_{\mathtt{def}}$). The situation is less clear if we work in an impure π-calculus without syntactical distinction (via the `def` construct) between imperative and functional names, replicated inputs being used both for functional definitions and imperative inputs.

4 Termination proof

In the proof, we take a well-typed $\pi^1_{\mathtt{ST}}$ process, assume that it is divergent, and then derive a contradiction. Using a pruning function (Definition 4) and a related simulation result (Lemma 7), we transform the given divergence into one for a functional process in $\pi^1_{\mathtt{def}}$. This yields contradiction, as $\pi^1_{\mathtt{def}}$ is terminating (Theorem 1).
Thus the definition of pruning is central in our proof. Intuitively, pruning computes the *functional backbone* of a process P, by getting rid of imperative outputs and replacing imperative inputs (replicated or not) in P with the simplest possible functional term, namely $\mathbf{0}$. However, to establish simulation, we would need reduction to commute with pruning (possibly replacing reduction with a form of "semantic equality" after commutation). This however does not hold, at least using a simple definition of pruning: take for instance $P_0 \stackrel{\text{def}}{=} !a.\overline{f} \mid \overline{a} \longrightarrow P_1 \stackrel{\text{def}}{=} !a.\overline{f} \mid \overline{f}$; the pruning of P_0 would be $\mathbf{0}$ whereas that of P_1 would be $\overline{f}$, and the latter processes cannot be related in a natural way.
We therefore have to be more precise, and make pruning parametric on a level p that occurs infinitely often in the reductions of the given divergent computation (cf. Lemma 8 – the level of a reduction is the level of the name along which the reduction occurs). Further, we define p as the maximal level that occurs infinitely often in the divergent computation (Lemma 8). Thus, at the cost of possibly removing an initial segment of the computation, we can assume the absence of reductions at levels greater than p. Indeed the actual pruning computes the *functional backbone at level p* of a process: we remove all imperative constructs, *and* the functional constructs (`def` and outputs) acting on functional names

$$\mathsf{pr}^p_\Gamma(a(x).P) = \mathsf{pr}^p_\Gamma(!a(x).P) = \mathsf{pr}^p_\Gamma(\mathbf{0}) \stackrel{\text{def}}{=} \mathbf{0}$$

$$\mathsf{pr}^p_\Gamma(P_1 \mid P_2) \stackrel{\text{def}}{=} \mathsf{pr}^p_\Gamma(P_1) \mid \mathsf{pr}^p_\Gamma(P_2)$$

$$\mathsf{pr}^p_\Gamma((\boldsymbol{\nu} a)\, P) \stackrel{\text{def}}{=} (\boldsymbol{\nu} a)\, \mathsf{pr}^p_\Gamma(P) \qquad \mathsf{pr}^p_\Gamma(\overline{a}\langle v\rangle.P) \stackrel{\text{def}}{=} \mathsf{pr}^p_\Gamma(P)$$

$$\mathsf{pr}^p_\Gamma(\mathtt{def}\ f^n = (x).P_1\ \mathtt{in}\ P_2) \stackrel{\text{def}}{=}$$

$$\begin{cases} \mathtt{def}\ f = (x).\mathsf{pr}^p_\Gamma(P_1)\ \mathtt{in}\ \mathsf{pr}^p_\Gamma(P_2) & \text{if } n = p \\ \mathsf{pr}^p_\Gamma(P_2) & \text{otherwise} \end{cases}$$

$$\mathsf{pr}^p_\Gamma(\overline{f}^n\langle v\rangle.P) \stackrel{\text{def}}{=} \begin{cases} \overline{f}\langle v\rangle.\mathsf{pr}^p_\Gamma(P) & \text{if } n = p \\ \mathsf{pr}^p_\Gamma(P) & \text{otherwise} \end{cases}$$

Fig. 3. Pruning in $\pi^1_{\mathtt{ST}}$

whose level is smaller than p. Typing ensures us that, in doing so, no functional constructs at level p that participate in the infinite computation are removed. Therefore, the functional reductions at level p in the original divergent computation are preserved by the pruning, while the other kinds of reductions have no semantic consequence on pruning (in the sense that processes before and after the reduction are pruned onto semantically equal terms). We thus derive an infinite computation in $\pi^1_{\mathtt{def}}$, which is impossible as $\pi^1_{\mathtt{def}}$ is terminating.
We can remark in passing that in the example given above, P_0 is pruned to $\mathbf{0}$, and so is P_1: indeed, the level of a is less or equal than p (otherwise we would not examine a reduction of P_0), so that f's level is strictly smaller than p, which entails, by Definition 4 below, that the output on f disappears when pruning P_1: in this case, the prunings of the terms involved in the reduction are equal.

Pruning and its properties. The definition and properties of the pruning function are parametric upon a level p (which will be fixed in Lemma 8 below) and a typing Γ.

Definition 4 (Pruning) *The pruning of P w.r.t. p and Γ, written $\mathsf{pr}^p_\Gamma(P)$, is defined by induction on P as in Figure 3, where a (resp. f^n) is a name whose type in Γ is imperative (resp. functional with level n).*

We write $\mathrm{PR}(\Gamma)$ for the typing context of the simply-typed π-calculus obtained from Γ by removing all level information from the types, and $\vdash^\pi_\Gamma Q$ for the typing judgements in the simply-typed π-calculus. We rely on these notations to state the following lemma:

Lemma 5 *Suppose $\vdash_\Gamma P : l$. Then, for any p, $\vdash^\pi_{\mathrm{PR}(\Gamma)} \mathsf{pr}^p_\Gamma(P)$.*

We write $\simeq$ for *strong barbed congruence*, defined in the usual way [**?**]. $\simeq$ is needed in the statement of the two next lemmas — we essentially use that $\simeq$ preserves divergences and is closed by static contexts.

The following technical lemma establishes that in certain cases, the pruning of the continuation of an input is inactive, up to $\simeq$, and can thus be removed. This property is useful to derive the key simulation result involving pruning (Lemma 7).

Lemma 6 (Pruning – inactivity)
- *If* $\vdash_\Gamma \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2 : n$, *with* $\Gamma(f) = \sharp^n_{\mathrm{F}}\ T$ *and* $n < p$, *then for any* v *s.t.* $\Gamma(v) = T$, $\mathsf{pr}^p_\Gamma(P_1)\{v/x\} \simeq \mathbf{0}$.
- *If* $\vdash_\Gamma !a(x).P_1 : n$, *with* $\Gamma(a) = \sharp^n_{\mathrm{F}}\ T$ *and* $n \leq p$, *then for any* v *s.t.* $\Gamma(v) = T$, $\mathsf{pr}^p_\Gamma(P_1)\{v/x\} \simeq \mathbf{0}$.
- *If* $\vdash_\Gamma a(x).P_1 : n$, *with* $\Gamma(a) = \sharp^n_{\mathrm{F}}\ T$ *and* $n \leq p$, *then for any* v *s.t.* $\Gamma(v) = T$, $\mathsf{pr}^p_\Gamma(P_1)\{v/x\} \simeq \mathbf{0}$.

Note that these properties are not immediate: for instance, in $a(x).P$, the continuation P may contain top-level definitions at level p, that are not removed by pruning. We write $P \longrightarrow^n_{\mathrm{I}} P'$ (resp. $P \longrightarrow^n_{\mathrm{F}} P'$) if P has a reduction to P' in which the interacting name is imperative (resp. functional) and its level is n.

Lemma 7 (Simulation) *Suppose* $\vdash_\Gamma P : l$.
1. *If* $P \longrightarrow^p_{\mathrm{F}} P'$, *then* $\mathsf{pr}^p_\Gamma(P) \longrightarrow \mathsf{pr}^p_\Gamma(P')$;
2. *If* $P \longrightarrow^n_{\mathrm{F}} P'$ *with* $n < p$, *then* $\mathsf{pr}^p_\Gamma(P) \simeq \mathsf{pr}^p_\Gamma(P')$;
3. *If* $P \longrightarrow^n_{\mathrm{I}} P'$ *with* $n \leq p$, *then* $\mathsf{pr}^p_\Gamma(P) \simeq \mathsf{pr}^p_\Gamma(P')$.

A delicate aspect of the proof of Lemma 7 is that if, for instance, $!a(x).P$ is involved in a reduction, an instantiated copy of P is unleashed at top-level; we rely on Lemma 6 to get rid of it (modulo $\simeq$).

The impossibility of an infinite computation. We can now present the termination proof, which exploits pruning to derive a contradiction from the existence of an infinite computation starting from a typable process.

Lemma 8 *If* $\vdash_\Gamma P_0 : l$, *and if there is an infinite computation starting from* P_0, *given by* $\{P_i\}_{i\geq 0}$, *then there exists a maximum level* p *such that there are infinitely many reductions at level* p *in the sequence. Moreover, there are infinitely many functional reductions at level* p.

Proof (sketch). *The first part holds because a process is typed using a finite number of levels.*
For the second part, we exploit the fact that the multiset of input-unguarded outputs at level p *in a process decreases strictly along imperative reductions at level* p, *and decreases or remains equal along (imperative or functional) reductions at levels strictly smaller than* p.

Theorem 9 (Soundness) *All processes in* π^1_{ST} *are terminating.*

Proof (sketch). *By absurd, let $\{P_i\}_{i\geq 0}$ be an infinite computation starting from a $\pi^1_{\mathtt{ST}}$ process P_0 such that $\vdash_\Gamma P_0 : l$ for some Γ and l. By Proposition 3, all the P_i's are well-typed. By Lemma 8 there is a maximal p s.t. for infinitely many i, $P_i \longrightarrow^p_{\mathrm{F}} P_{i+1}$. Moreover, still by Lemma 8, we can suppose w.l.o.g. that no reduction occurs at a level greater than p. Whenever $P_i \longrightarrow^p_{\mathrm{F}} P_{i+1}$, we obtain by Lemma 7 $\mathsf{pr}^p_\Gamma(P_i) \longrightarrow \mathsf{pr}^p_\Gamma(P_{i+1})$. Lemma 7 also tells us that when P_i reduces to P_{i+1} via a reduction that is not a functional reduction at level p, we have $\mathsf{pr}^p_\Gamma(P_i) \simeq \mathsf{pr}^p_\Gamma(P_{i+1})$. This allows us to construct an infinite computation in $\pi^1_{\mathtt{def}}$ (by Lemma 5), which is impossible (Theorem 1).*

5 Parametrisation on the core calculus

In this and the following sections we discuss how the exposed method can be enhanced and enriched.

In Section 4, we assume a termination proof for (a subset of) the core functional $\pi_{\mathtt{def}}$ from which the impure terms are built, but we never look into the details of such proof. As a consequence, the method can be made parametric. Let F be the chosen terminating functional core, and L the impure π-calculus language that we want to prove terminating. There are three requirements on L:

1. processes in L are well-typed in the type system of Figure 2, but without the constraint $f \notin \mathrm{fn}(P_1)$ of rule (**Def**) (that represented a specific constraint coming from $\pi^1_{\mathtt{def}}$);
2. L is reduction closed (i.e., $P \in L$ and $P \longrightarrow P'$ imply $P' \in L$);
3. the pruning of a process in L (as given by Definition 4) returns a process in F.

Theorem 10 *If F and L satisfy (1-3) above, then L is terminating.*

It may be possible to formulate constraints for condition (3) as typing constraints to be added to the type system used in condition (1), as we actually did in Section 3, where F is $\pi^1_{\mathtt{def}}$ and L is $\pi^1_{\mathtt{ST}}$.

An example of a functional core that could be chosen in place of $\pi^1_{\mathtt{def}}$ is $\pi^{\mathrm{pr}}_{\mathtt{def}}$, the π-calculus image of the primitive-recursive functions. Its termination is enforced by a control of recursion; the constraint on a construct $\mathtt{def}\ f = (x).P\ \mathtt{in}\ Q$ is that for any input-unguarded output $\overline{c}\langle v\rangle$ in P: (i) either the level of f is higher than that of c, or (ii) the two names have the same level and carry a first-order argument (e.g., a natural number) that decreases its value ($v < x$).

Even when the termination of the functional core can be proved using measures, as is actually the case for $\pi^{\mathrm{pr}}_{\mathtt{def}}$, the parametrisation on this core offered by the method could be useful, because the measures employed in the proofs for the functional core and for the whole language can be different; see example $Sys_{2'}$ discussed at the end of Section 6.

Iterating the method. Our method could also be applied to a purely functional calculus in which names are partitioned into two sets, and the measure technique guarantees termination only for the reductions along names that belong to one of the sets. An example is $\pi^{1+\mathrm{pr}}_{\mathtt{def}}$, the calculus combining $\pi^{1}_{\mathtt{def}}$ and $\pi^{\mathrm{pr}}_{\mathtt{def}}$; thus, in any construct $\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2$, either f is not used in P_1, or all outputs in P_1 at the same level as f emit values provably smaller than x. (We think that $\pi^{1+\mathrm{pr}}_{\mathtt{def}}$ is weaker than the π-calculus image of System T [**?**].)
The method can be iteratively applied. For instance, having used it to establish termination for the functional calculus $\pi^{1+\mathrm{pr}}_{\mathtt{def}}$, as suggested above, we can then take $\pi^{1+\mathrm{pr}}_{\mathtt{def}}$ as the functional core of a new application of the method. This iteration could for instance add imperative names. In [**?**], we use an iteration of the method to prove the termination of an example that combines the two examples presented in Section 6.

6 Examples

This section shows examples of the kind of systems whose termination can be proved with the method exposed in the earlier sections. These systems make use of both imperative and functional names. The systems cannot be handled by previous measure-based techniques, which are too weak on functional names, or by logical relation techniques, which are too weak on imperative names. Our method offers the capability of combining techniques for both kinds of names, which is essential.
The examples use polyadicity, and first-order values such as integers and related constructs (including arithmetic constructs, and if-then-else statements). These extensions are straightforward to accommodate in the theory presented earlier. A longer example is given in [**?**].

An encryption server. In this example, several clients c_i are organised into a chain. Clients are willing to communicate, however direct communications between them are considered unsafe. Instead, each client must use a secured channel s to contact a server that is in charge of encrypting and sending the information to the desired client. Hence the messages travel through the c_i's in order to be finally emitted on d. A client c_i, receiving a message, has to perform some local imperative atomic computations. For readability, we condense this part into the acquire and release actions of a local lock named $lock_i$.
Several messages can travel along the chain concurrently, and may overtake each other; the example is stated with n initial messages (they are all sent to c_1 but could have been directed to other clients).

$$
\begin{array}{l}
Sys_1 \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu} lock_1, .., lock_k) \\
\quad \Big(\ lock_1 \mid .. \mid lock_k \mid \\
\quad\quad \mathtt{def}\ s = (c, x).\overline{c}\langle \mathbf{enc}[c, x]\rangle\ \mathtt{in} \\
\quad\quad\quad \mathtt{def}\ c_1 = C_1\ \mathtt{in}\ \ \ldots\ \mathtt{def}\ c_{k-1} = C_{k-1}\ \mathtt{in}\ \mathtt{def}\ c_k = C_k\ \mathtt{in} \\
\quad\quad\quad\quad (\overline{s}\langle c_1, \mathrm{msg}_1\rangle \mid \ \ldots \mid \overline{s}\langle c_1, \mathrm{msg}_n\rangle) \ \Big)
\end{array}
$$

where C_i $(1 \leq i < k)$ and C_k are:

$$C_i \stackrel{\text{def}}{=} (y_i).\overline{lock_i}.(lock_i \mid \overline{s}\langle c_{i+1}, \mathbf{dec}_i[y_i]\rangle)$$
$$C_k \stackrel{\text{def}}{=} (y_k).\overline{lock_k}.(lock_k \mid \overline{d}\langle \mathbf{dec}_k[y_k]\rangle)$$

and **enc**, **dec** are operators for encryption and decryption, with equalities $\mathbf{dec}_i[\mathbf{enc}[c_i, m]] = m$ for all i.
The way the c_i's are organised (here a chain, but any well-founded structure could have been used) ensures that the interactions taking place among the server and the clients do not lead to divergent behaviours: this is guaranteed by showing that the system is terminating.
In the typing all the c_i's must have the same type, because they all appear as the first component of messages on s. We can type-check Sys_1 using the system of Section 3 (where the core functional language is $\pi^1_{\mathtt{def}}$); if $\mathtt{b}$ is the type for the encrypted/decrypted data, then we assign $\sharp^1_{\mathtt{F}}\,\mathtt{b}$ for the c_i's, $\sharp^1_{\mathtt{F}}\,(\sharp^1_{\mathtt{F}}\,\mathtt{b} \times \mathtt{b})$ for s, and $\sharp^1_{\mathtt{I}}\,\mathtt{unit}$ for the $lock_i$'s.
The loose assignment of levels to functional names (the possibility $k = l$ in rule (**Def**) of Section 3) is essential for the typing: an output on c_i can follow an input on s on the server's side, and conversely on the clients' side: c_i and s must therefore have the same level.

A movie-on-demand server. In this second example, the server s is a movie database, and handles requests to watch a given movie (in streaming) a given number of times. By sending the triple $\langle 15, r', \mathbf{tintin}\rangle$ on s, the client pays for the possibility to watch the movie **tintin** 15 times; r' (r in the server's definition) is the return channel, carrying the URL (h) where the movie will be made available once.
Two loops are running along the execution of the protocol. On the server's side, a recursive call is generated with $n - 1$, after consumption of one access right. On the client's side, process $!c.r'(z).(\overline{c} \,|\, z)$ keeps interrogating the server: the client tries to watch the movie as many times as possible.

$$\begin{array}{rcl} Sys_2 & \stackrel{\text{def}}{=} & \mathtt{def}\ s = (n, r, f). \\ & & \quad \big(\ \mathtt{if}\ f = \mathbf{tintin}\ \mathtt{then}\ (\nu h)\ (\overline{r}\langle h\rangle.\overline{h}) \\ & & \quad\ \mid \mathtt{if}\ f = \mathbf{asterix}\ \mathtt{then}\ \ldots \\ & & \quad\ \ldots \\ & & \quad\ \mid \mathtt{if}\ n > 0\ \mathtt{then}\ \overline{s}\langle n-1, r, f\rangle\ \big) \\ & & \mathtt{in}\ \ (\boldsymbol{\nu} r')\ \big(\ \overline{s}\langle 15, r', \mathbf{tintin}\rangle \mid (\boldsymbol{\nu} c)\,(\overline{c} \mid !c.r'(z).(\overline{c} \,|\, z))\ \big) \end{array}$$

Here again, we rely on typing to guarantee that this system, where a server is able to call itself recursively, does not exhibit divergences.
Sys_2 uses both functional names (e.g., s) and imperative names (e.g., r). Its termination is proved by appealing to the primitive recursive language $\pi^{\mathrm{pr}}_{\mathtt{def}}$ as core functional calculus. In the typing, channel c is given level 1, and s, r level 2 (this allows us to type-check the recursive output on c).

To understand the typing of functional names, it may also be useful to consider the variant $Sys_{2'}$ in which the following definition of a server s' is inserted between the definition of s and the body $(\boldsymbol{\nu} r')\ldots$:

$$\mathtt{def}\ s' = (n, r, f).\overline{s}\langle n, r, f\rangle\ \mathtt{in}\ ..$$

Here, s' models an old address where the server, now at s, used to run. Some of the clients might still use s' instead of s, and s' hosts a forwarder that redirects their requests to s.
We can type $Sys_{2'}$ thanks to the looser level constraints on functional names which allow s and s' to have the same type; the functional core is still $\pi^{\mathrm{pr}}_{\mathtt{def}}$. Note that a termination proof of the core calculus $\pi^{\mathrm{pr}}_{\mathtt{def}}$ by a simple measure argument is not in contradiction with the observation that similar measures are too weak to type $Sys_{2'}$: the levels used in the typing of $Sys_{2'}$ need not be the same as those used in the termination proof of its functional core (the pruning of $Sys_{2'}$). Indeed in this functional core s' can be given a level higher than that of s (which is possible because the imperative clients have been pruned, hence, *after the pruning*, s and s' need not have the same type).

7 Refinements

Non-replicated imperative inputs. In the type system of Figure 2, non-replicated inputs introduce the same constraint $k > l$ as replicated inputs (rules (**In**) and (**Rep**)). This can be annoying, as the inputs that are dangerous for termination are the replicated ones. Simply removing the constraint $k > l$ from rule (**In**) would however lead to an unsafe system. For instance, we could type the divergent process

$$\mathtt{def}\ f = (x).(\overline{a}\langle x\rangle \mid \overline{x})\ \mathtt{in}\ \ \mathtt{def}\ g = a(y).\overline{f}\langle y\rangle\ \mathtt{in}\ \overline{f}\langle g\rangle\ \ .$$

This example shows some of the subtle interferences between functional and imperative constructs that may cause divergences.
We can improve rule (**In**) so to impose the constraint $k > l$ only when the value communicated in the input is functional. Rule (**In**) is replaced by the following rule, where "T functional" holds if T is of the form $\sharp^{k'}_{\mathtt{F}}\ T'$, for some k', T':

$$(\mathbf{In}')\ \frac{\vdash_\Gamma P : l \qquad \vdash_\Gamma a : \sharp^{k}_{\mathtt{I}}\ T \qquad \vdash_\Gamma x : T \qquad \text{if } T \text{ functional then } k > l \text{ and } l' = 0, \text{ otherwise } l' = l}{\vdash_\Gamma a(x).P : l'}$$

Corresponding modifications have to be made in the definition of pruning and related results. The rule could be refined even further, by being more selective on the occurrences of x in P when x is functional. An example of this is discussed in [**?**], to be able to treat name c in the process Sys_2, from Section 6, as functional. (It is also possible to avoid communications of functional names along imperative names, by simple program transformations whereby communication of a functional name is replaced by communication of an imperative name that acts as a forwarder towards the functional name.)
A benefit of these refinements is to be able to accept processes $a(x).P$ where a is imperative and appears in input-unguarded outputs of P. This typically arises in the modelling of mutable variables in the asynchronous π-calculus (references can also be modelled as services that accept read and write requests; in this case the above refinements of the input rule are not needed).
We discuss another refinement of the measure-based analysis in [**?**].

Polymorphism. The method can be extended to handle polymorphism [**?**]. A level on each type should however be maintained, and type abstraction therefore cannot hide levels. A type variable X is hence annotated with a level k, and it should only be instantiated with types whose level is less than or equal to k (assuming that first-order types like integers have level 0).

8 Conclusions

We have described an analysis to guarantee termination of impure languages. Our technique exploits pruning in order to lift a termination proof based on logical relations (for a functional language) to a termination proof for a richer language.

Despite recent progresses in impure simply-typed λ-calculi, adapting the realisability/logical relation technique to non-trivial impure concurrent languages such as π-calculus remains hard. All our attempts failed, intuitively because, on the one hand, typing naturally arises in π-calculus on names and less naturally on terms; on the other hand, even adopting forms of behavioural types for π-calculus terms, the main operator for composing processes is parallel composition, and this operator does not isolate computation (in contrast with λ-calculus application, which forces two terms to interact with each other). Using restricted forms of parallel composition in π-calculus often forbids useful imperative patterns.

The extensions of realisability recently studied by Boudol [**?**] and Amadio [**?**] for impure λ-calculi can also handle extra features such as threads. However the threads in these works do not interfere with termination. Most importantly, the underlying language must be the λ-calculus (the technique relies on the standard realisability, and the crucial type construct in proofs is the functional type). It does not seem possible to transport these proofs outside the λ-calculus, e.g., in the π-calculus.

In another extension of the realisability technique, Blanqui [**?**] is able to lift a termination proof based on realisability for simply-typed λ-calculus into a termination proof for a λ-calculus with higher-order pattern-matching operators, provided a well-founded order exists on the constructors used for pattern-matching. Again, as refinement of the realisability approach for a λ-calculus, the method proposed, and the details of the proof, are entirely different from ours.

In the paper, we have proposed a method to treat the problem in which a standard measure-based technique for imperative languages is made parametric with respect to a terminating functional core language; the termination of the core language is established separately. The method can be further enhanced by refining the initial measure technique. The core language parameter of the method is functional. We do not know how to relax the property, needed in the definition of pruning.

Acknowledgements. We wish to thank G. Boudol for fruitful discussions in the initial stages of this work. Support from the french ANR projects "CHoCo" and ANR-08-BLANC-0211-01 "Complice", and by the European Project "HATS" (contract number 231620) is also acknowledged.

A Examples of typings

A.1 Typing the examples of Section 6

To type-check the example Sys_1, we use the following type assignment:

$$lock_i : \sharp^0_{\mathrm{I}}\,\mathtt{unit} \qquad c_i : \sharp^1_{\mathrm{F}}\,\mathtt{b} \qquad s : \sharp^1_{\mathrm{F}}\,(\sharp^1_{\mathrm{F}}\,\mathtt{b} \times \mathtt{b}) \qquad \mathrm{msg}_i : \mathtt{b} \qquad d : \sharp^1_{\mathrm{I}}\,\mathtt{b}$$

Thus, the functions $\mathbf{dec}_i$ have type $\mathtt{b} \to \mathtt{b}$ and the function **enc** has type $(\sharp^1_{\mathrm{F}}\,\mathtt{b} \times \mathtt{b}) \to \mathtt{b}$. Definition on s (of level 1) is typed as the sole output is on c (of level 1) and $1 \geq 1$. Definitions on c_i (of level 1) are typed as the outputs are on $lock_i$ (of level 0), s (of level 1) and d (of level 1), thus we have $1 \geq 0$.
Moreover, the definitions respects the $\pi^1_{\mathtt{def}}$ conditions: s does not appear in the definition of s, and c_i does not appear in C_i.
To type-check the example Sys_2, we use the following type-assignment:

$$f, \mathbf{tintin}, \mathbf{asterix} : \mathtt{b} \qquad h, z : \mathtt{b} \qquad r, r' : \sharp^1_{\mathrm{F}}\,\mathtt{b} \qquad s : \sharp^2_{\mathrm{F}}\,(\mathtt{nat} \times \sharp^1_{\mathrm{F}}\,\mathtt{b} \times \mathtt{b})$$

$$c : \sharp^0_{\mathrm{I}}\,\mathtt{b}$$

We assume that the weight of an `if then else` construct is the maximum weight of each branch. Definition of s (of level 1) is type-checked as it contains only outputs on smaller level (r, h) and an output on s but with a smaller integer argument $(n - 1 < n)$. The imperative replicated input on c is typed as the recursive output on c (of level 0 is found under an input on r' (of level 1).

A.2 A larger example

The example presented here refines the two in Section 6 and puts them together. We describe a distributed server that broadcasts movies and a bank server managing the account of the clients. The former server is implemented by a chain of specialised servers, $\mathrm{Server_i}$, the latter is constituted by a bunch of replications in parallel.
The main interest in giving this example is in how termination is proved, which we discuss below. We first provide some explanations about the code given in Figure 4:

- $\mathrm{Server_i}$ is the ith server in the chain; it is able to broadcast film mov_i; if that film was not requested, the request is passed to the next server.
- $\mathrm{Server_i}$ receives: a request about an account information acc (encrypted), a number of runs for the movie n, a return channel r and a movie name f.
- The servers do not know how to encrypt data, they can only decrypt it using their own key. Hence when $\mathrm{Server_i}$ wants to transmit the request to $\mathrm{Server_{i+1}}$, it goes through the centralised server.
- To handle a request, $\mathrm{Server_i}$ acquires its local lock $lock_i$, performs some internal computation (e.g. logging), releases the lock, interacts with the bank, generates a new channel for the streaming (h), and finally sends the movie.

$$\begin{array}{l}
\text{Bank} \stackrel{\text{def}}{=} \quad !create(acc).\overline{acc}\langle 0\rangle \;\mid\; !add(acc,n).acc(x).\overline{acc}\langle n+x\rangle \\
\qquad\quad \mid\; !get(acc,n,s).acc(x).\texttt{if } n \geq x \;\; \texttt{then } (\overline{acc}\langle n-x\rangle \mid \overline{s}\langle \mathbf{accept}\rangle) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else } (\overline{acc}\langle x\rangle \mid \overline{s}\langle \mathbf{reject}\rangle)
\end{array}$$

$$\begin{array}{l}
\text{Server}_{\text{i}} \stackrel{\text{def}}{=} \quad (n,r,f,acc).\texttt{if } f = \text{mov}_i \\
\qquad\qquad \texttt{then } \big(\texttt{if } n>0 \texttt{ then } \overline{s_i}\langle y_i, n-1, r, f\rangle \texttt{ else } \mathbf{0} \\
\qquad\qquad\quad \mid \overline{lock_i}.(\boldsymbol{\nu} ch)\; \overline{get}\langle acc, 1, ch\rangle.ch(ans).lock_i. \\
\qquad\qquad\qquad \texttt{if } ans = \mathbf{accept} \texttt{ then } (\boldsymbol{\nu} h)\,\overline{r}\langle h\rangle.\overline{h} \texttt{ else } \mathbf{0}\,\big) \\
\qquad\qquad \texttt{else } \overline{s}\langle s_{i+1}, n, r, f, \mathbf{dec}_i[acc]\rangle
\end{array}$$

$$\begin{array}{l}
Sys \stackrel{\text{def}}{=} \quad (\boldsymbol{\nu} lock_1, .., lock_k) \\
\qquad \texttt{def } s = (c,n,r,f,x).\overline{c}\langle n,r,f,\mathbf{enc}[c,x]\rangle \texttt{ in} \\
\qquad \texttt{def } s_1 = \text{Server}_1 \texttt{ in} \\
\qquad\quad \ldots \\
\qquad \texttt{def } s_{k-1} = \text{Server}_{\text{k}-1} \texttt{ in} \\
\qquad \texttt{def } s_k = (n,r,f,acc).\mathbf{0}) \quad \texttt{in} \\
\qquad \Big(\quad lock_1 \mid .. \mid lock_k \\
\qquad\quad \mid \text{ Bank} \\
\qquad\quad \mid \; (\boldsymbol{\nu} r', id)\,\big(\overline{create}\langle id\rangle \mid \overline{add}\langle id, 14\rangle \\
\qquad\qquad \mid \overline{s}\langle s_1, 15, r', \mathbf{asterix}, id\rangle \mid (\boldsymbol{\nu} c)\,(\overline{c} \mid !c.r'(z).(z \mid \overline{c})) \;\mid\; \overline{t}\langle r'\rangle\,\big)\,\Big)
\end{array}$$

Fig. 4. A distributed movie server interrogating a bank

– The initial process opens an account id, adds some money to id, and sends a request for 15 visions of the movie **asterix**, the return channel being r'. When a request for a movie is sent, the server interrogates the bank to find out whether enough money is available.
– While trying to see the movie as often as possible, the client also sends r' to a friend, on channel t, thus giving her the possibility to watch the movie as well: the input capability on r' can thus be transmitted.
– The Bank server offers three methods: *create* to create a new account, *add* and *get* to put and retrieve money

In the example, the imperative aspects are given by the interactions with process Bank to manipulate the bank account, and by the manipulations of the locks which are local to each Server_i.
In order to type-check this example, as discussed at the end of Section 5, we view it as belonging in a calculus that consists in the superposition of three calculi. Name s is a functional name in $\pi^{1}_{\mathtt{def}}$, the servers s_i are functional names in $\pi^{\text{pr}}_{\mathtt{def}}$, and the bank methods $create, add, get$ are imperative names. We first prove the termination of $\pi^{1+\text{pr}}_{\mathtt{def}}$, using our method, and then, considering it as a terminating functional calculus, we prove the termination of the whole calculus. To handle the type-checking of this example, the extension we mentioned in Section 7 is required: the imperative inputs on *acc* in the Bank subprocess have to be treated with rule (**In**$'$) for non-replicated inputs.

Name c is imperative. The recursive output at c is not dangerous because "covered" by the intermediate input at r. We could also have chosen to implement the replicated input at c using a functional definition, modulo a simple modification of the typing rule (**Def**), discussed in Remark 12.

B Further enrichments of the analysis: input sequences

We give an example of how our termination method can be further enhanced by enriching the measure-based system. The weights are given by *multisets* of levels rather than single levels, adapting ideas from [**?**].
The refinement discussed here makes it possible to accept recursive replications, that is, processes $!a(x).P$ where P contains input-unguarded outputs at a or at names of the same level as a. For this, intuitively, the typing follows the structure of the process under the replication, recording the levels of the nested inputs so encountered; the multiset of these levels is compared against the multiset of the levels of the remaining input-unguarded outputs. We discuss the simplest such analysis, where the sequence of inputs is at the top. This is a common pattern in applications, and does not involve heavy technicalities.
We thus assume that the syntax of the calculus allows replicated input sequences, i.e., a construct $!a_1(x_1).\ldots.a_n(x_n).P$. As we need to compare multisets, we represent them as vectors $(n_1, \ldots n_m)$ where n_j is the number of occurrences of level j in the multiset. In the rule below, N is the vector of the levels of the input-unguarded outputs in P, $\uplus_j k_j$ is

the vector of the k_i's, $<_{\mathrm{mul}}$ is vector comparison. Finally, $\mathrm{Mlf}(P)$ is the maximum level of an input-unguarded functional output in P.

$$(\mathbf{Rep}') \quad \frac{\vdash_\Gamma P : N \qquad \forall j, \Gamma(a_j) = \sharp_{\mathtt{I}}^{k_j} T_j \text{ and } \Gamma(x_j) = T_j \qquad \uplus_j k_j <_{\mathrm{mul}} N \qquad \max_j(k_j) > \mathrm{Mlf}(P)}{\vdash_\Gamma !a_1(x_1).\ldots.a_p(x_p).P : \emptyset}$$

The need for the final constraint on $\mathrm{Mlf}(P)$ is shown with the divergent process $\mathtt{def}\ f = (\overline{a} \mid \overline{a})\ \mathtt{in}\ !a.a.\overline{f} \mid \overline{f}$, that would otherwise be typed with the assignments $a : \sharp_{\mathtt{I}}^1\ \mathtt{unit}$ and $f : \sharp_{\mathtt{F}}^1\ \mathtt{unit}$.
As a simple example, if a and b are imperative and of levels 2 and 3, and f is functional and of level 1, with the above rule we can accept $!b.a.(\overline{a} \mid \overline{f} \mid \overline{f})$, as the multiset sum $\{3, 2\}$ of the levels of the inputs is strictly greater than the the multiset sum $\{2, 1, 1\}$ of the levels of the input-unguarded outputs in the continuation.

Remark 11 *Notice that this extension gives us another way to typecheck imperative references in the π-calculus (as the extension for non-replicated inputs of Section 7 did). In order to typecheck the process $!add(acc, n).acc(x).\overline{acc}\langle x + n\rangle$, from Appendix A, we treat the prefixes $add(acc, n).acc(x)$ as a single input sequence, which is heavier than the weight of the output $\overline{acc}\langle x + n\rangle$.*

Remark 12 (A refinement of rule (Def)) *The language $\pi^1_{\mathtt{ST}}$ was defined in Section 3 by combining the imperative $\pi^1_{\mathtt{imp}}$ and the functional $\pi^1_{\mathtt{def}}$. If however we follow more closely the requirements in the parametrisation Theorem 10, then rule* (**Def**) *can be refined as follows. The requirement that the defined name is not used in the body of the definition, can be replaced by the weaker requirement that such name is used in the body only in input-guarded positions. This refinement is justified by the fact that, in the soundness proof of the method, when pruning the body of the definition all inputs disappear (and with them also all input-guarded outputs). The refinement could be used to turn the name c of the example in Appendix A.2 into a functional name, as hinted at the end of that appendix.*