Typing and Subtyping for the Termination of Mobile Processes

Ioana Cristescu

Training period at LIP, ENS Lyon, march-june 2011

Abstract. We propose a method for ensuring termination in concurrent programming languages. In our work we rely on π-calculus, a theoretical model useful for proving the method's correctness. Our solution is based on the description of a type system which combines previously used techniques for termination and an ordering on types, often employed in π-calculus, i/o subtyping. Our approach seems well-suited for the analysis of Erlang, a programming language emulating behaviour of mobile processes. We enrich the expressiveness of our type system by adapting it so that it contains an encoding of the λ-calculus. A brief description of the implementation of our work in Erlang is also available.

1 Introduction

During this internship, I worked in the Plume research team part of the LIP computer science laboratories, under the supervision of Daniel Hirschkoff. We worked on the termination property guaranteed by some type systems describing the behaviour of concurrent processes and our work was a continuation of the PhD thesis of Romain Demangeon [Dem10]. A more technical description of our results is available in [CH11].

Distributed programming and termination properties
The emergence of distributed systems as cost effective and generally applicable solutions for an increasing number of computational problems led to the need of finding concurrent algorithms and programming languages to run on such systems. Concurrent computing and a corresponding formalism that would assert its correctness are gaining in interest. Within an interactive system there are many forms of interprocess communication. The most used is probably message passing, which emulates the behaviour of independent systems connected through a network. Such a model of communication is exemplified by the **Erlang** programming language, [Lab11] which offers powerful primitives for managing processes and their asynchronous communication. As an example consider the Erlang program consisting of two processes sending the messages *ping* and *pong* to each other. We use the function `spawn` to create processes, the expression `Pid ! message` to send a message to a process with identifier `Pid` and the structure `receive message` $\rightarrow$ `handle_message` to receive a message. In our work, this is the kind of programs we focused on while reasoning over processes communication.

```
% the main function creating the two processes
start() -> PongPid = spawn(ex, pong, []), spawn(ex, ping, [PongPid]).

% process 1 : first sends (ping) and then receives (pong) a message from process 2
ping(PongPid) -> PongPid ! {ping, self()},
    receive pong -> print("Ping received pong")
    end.

% process 2 : after receiving a message, it sends one to process 1
pong() -> receive {ping, PingPid} -> print("Pong received ping"), PingPid ! pong
          end.
```

An important property in concurrent systems is termination. As shown in [KS10] termination guarantees lock-freedom, that is it guarantees the absence of concurrent races. At first sight, terminating processes are processes that eventually end their interactions with the environment. However, servers and other such concurrent systems that are expected to offer interaction continuously also need to be modelled. In this case termination refers to when those systems are engaging in an interaction with a process, in which case their exchange will eventually end. For instance, when a server offers access to a resource, we have to guarantee that after usage, the resource is again available. In other words, we want to avoid infinite internal activity within a process.

π-calculus as a model of concurrent and mobile computation
With the growth of the interactive systems' usage also appeared the necessity of building mathematical tools that would lead to precise and correct properties about these systems. This is the purpose of π-calculus [SW01], to be a model for mobile computing, the way λ-calculus is for sequential computational systems. In π-calculus the interaction between processes is viewed as processes communicating with each other through *channels*. The basic entity used is the *name* of a channel. Two processes can interact if they share a name and their interaction consists in exchanging names. They can either exchange a single name, in which case we have a *monadic* π-calculus, or tuples of names making the π-calculus *polyadic*. Among the actions a process can perform we will only consider two: either a process can send or it can receive a name through a channel. We then say that a process has the input or the output capability on a name. Consider again the above **Erlang** program. The two processes P_1 and P_2 can be expressed in π-calculus as $P_1 = \overline{a}\langle ping\rangle.b(message)$ and $P_2 = a(message).\overline{b}\langle pong\rangle$. A concurrent computation can change the process, in which case we say that the process performed a reduction. It can occur after an interprocess communication or after internal actions within a process. For instance, P_1 changes its configuration after sending *ping* to $P_1 = b(message)$.

Termination in π-calculus
In π-calculus we can define resources running for indefinite time, like servers, using replication on a name. The process $!a(x).P$ means that we can run P as many times as we want, by sending a message on channel a each time. For example, $!a(x).P \mid \overline{a}\langle v\rangle$ reduced to $!a(x).P \mid P[v/x]$, where we replaced in $P[v/x]$ all the occurrences of x with v. Finding type systems which ensure termination is a challenging task. One method is level based termination [DS06], in which we assign an integer value to each name, called the name's level, and a weight to each process, representing the maximum level of an output channel inside that process. The weight can then be used as a well-founded decreasing measure on a process, therefore ensuring its termination.

For instance in $!a(x).P$ we require that the weight of P to be smaller than the level of a. The process $!a(x).\overline{a}\langle u\rangle$ is liable to perform an infinite reduction, since $!a(x).\overline{a}\langle u\rangle|\overline{a}\langle u\rangle \longrightarrow !a(x).\overline{a}\langle u\rangle|\overline{a}\langle u\rangle$, and we cannot type it either as it would require $lvl(a) > lvl(a)$. Another example of an infinite reduction is the process $!a(x).\overline{b}\langle x\rangle \mid !b(y).\overline{a}\langle y\rangle$. The goal is to decrease the weight of the overall process even for a reduction which releases the process P under the replicated input. We can imagine these levels to be the cost of using a channel. Once we paid for a, the weight of P is not enough for another usage of a, and therefore it cannot trigger another reduction. In this manner we can avoid infinite loops as the one above. When computing the weight of a process, we are only interested in the output channels as only those can participate in an infinite reduction with the replicated input.

Subtyping
In a typed π-calculus we assign *types* to names, which represent constraints imposed on the name and its capabilities. Along with types the calculus also introduces *typing rules*. An important property, subject reduction, asserts that a reduction keeps processes well-typed. Most common such type system is the simply typed one [DS06], a correspondence of simply typed λ-calculus in π-calculus. For the process $a(x).x(v).\overline{v}\langle t\rangle \mid \overline{a}\langle u\rangle$ we assign the following types:$\Gamma(v) = \sharp T; \Gamma(x) = \sharp(\sharp T); \Gamma(u) = \sharp(\sharp T); \Gamma(a) = \sharp(\sharp(\sharp T))$. For example, the type of a certifies the fact that a can carry names of type $\sharp(\sharp T)$, while $\sharp(\sharp T)$ represents a channel for the names of type $\sharp T$.

But once one introduces types, one might want to assign an order on these types. An example of such a type system is i/o subtyping presented in [PS96]. The idea behind the i/o type system is that names carry in their type their capability (input or output), with one special type representing both capabilities. Considering the same example as above, now the types become: $\Gamma(v) = oT; \Gamma(x) = i(oT); \Gamma(u) = i(oT); \Gamma(a) = \sharp(i(oT))$.

Again, we can take a closer look at the type of a, which testifies that a is used as a channel with both capabilities for the name x. Likewise, x is an input channel for v, which has only the output capability.

Type inference
Both typing and termination play a major role in using a π-calculus type system for describing a concurrent system. The advantages typing brings to both a concurrent or a sequential programming language are considerable: detecting programming errors, inferring information about the behaviour of a program and so on. In a concurrent environment it can help detect deadlocks and infer correctness of a program. Therefore we need a type system which guarantees termination and in which, if possible, the types of the processes can be reconstructed efficiently. The reconstruction procedure is called *type inference.* We will refer to an inference procedure as polynomial if type inference is executed in polynomial time with respect to the size of the process.

The contributions of this training period
One can assign types to processes in π-calculus in order to guarantee termination. A common approach is to represent, relying on the type of a process, a measure, that is a numerical value which decreases along with the reductions that a process performs.

Our work consists of:

– introducing a type system which combines this kind of level based termination with an order on types and weights. Ordering the levels is straightforward, while for the types we used the i/o subtyping;
– proving termination for it;
– evaluating the expressiveness of the type system using various examples;
– showing a possible extension of our type system using a variation of i/o subtyping on which we can also type functional names;
– providing a polynomial time type inference for a version of our type system, more suitable for typing concurrent programming languages, such as Erlang;
– implementing a possible translation of the processes belonging to the type system in Erlang.

Structure of this document
Section 2 introduces the basic notions and notations of π-calculus necessary to understand the notations and conventions used all along this document. Section 3 takes a deeper look at termination and subtyping, and presents the type systems studied before introducing our type system. In Section 4.1 we present our type system for which we first prove termination. Then we show its limits in terms of expressiveness, by using an encoding of λ-calculus into π-calculus. We also present a polynomial time inference procedure for a more restrictive form of our type system, and hint at a possible inference procedure for the whole type system. Points about the implementation of this type system in Caml and Erlang are presented in Section 5. We conclude the document with a discussion in section 6.

2 Preliminary Notions on Processes

This section introduces the basic notations in π-calculus necessary for the further description of our type systems.

Suppose we have an infinite set of names. We use a, b, c, x, y to designate them.

Definition 1 *Processes, denoted by $P, Q, R, \ldots$, and values, v, are defined by the grammar:*

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

Here, $\mathbf{0}$ stands for inactivity and we will omit it when writing a process. Moreover, we often employ the notations a and $\overline{a}$ instead of $a(x)$ and $\overline{a}\langle\star\rangle$, respectively. In this case we assume that a carries names of type $\mathbb{U}$.

Name *binding* in π-calculus has a similar meaning to the one it has in λ-calculus. In $(\boldsymbol{\nu} z)\, P$ and $a(z).P$ the name z is bound within P. Names that are not bound, are called free names. We denote $\mathrm{fn}(P)$ the set of free names of P. The general convention is that all bound names are pairwise distinct and are different from the free names. Similar, $P[b/x]$ is the *substitution* of x by b in P, where x is a free name. We require the substitution to be capture avoiding, which implies that if there is a name collision within the process during the substitution, we use renaming.

Within a process we can rewrite the terms while keeping the resulting process equivalent to the original one. This is formalised using a relation called structural congruence.

Definition 2 *Structural congruence, denoted by $\equiv$, is the smallest equivalence relation that is a congruence, contains α-conversion, and which satisfies the following axioms:*

$$P|(Q|R) \equiv (P|Q)|R \qquad P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad (\boldsymbol{\nu} a)(\boldsymbol{\nu} b)\, P \equiv (\boldsymbol{\nu} b)(\boldsymbol{\nu} a)\, P$$

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

We denote by $P \longrightarrow P'$ a reduction on processes, where P can evolve to P' after an internal action.

Definition 3 *A reduction relation is defined by the following inference rules:*

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

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

Example 4

$$\boldsymbol{\nu} a.(\boldsymbol{\nu} b.\overline{a}\langle b\rangle) \mid a(x).\overline{x}\langle v\rangle \mid b(y) \longrightarrow \boldsymbol{\nu} b.\overline{b}\langle v\rangle \mid b(y) \longrightarrow 0$$

$$\overline{a}\langle b\rangle \mid \overline{a}\langle c\rangle \mid !a(x).\overline{x}\langle v\rangle \longrightarrow \overline{b}\langle v\rangle \mid \overline{a}\langle c\rangle \mid !a(x).\overline{x}\langle v\rangle \longrightarrow \overline{b}\langle v\rangle \mid \overline{c}\langle v\rangle \mid !a(x).\overline{x}\langle v\rangle$$

One can define several sub-calculi to the π-calculus. So far, we made the assumption that the calculus is *synchronous*, that is that a communication between processes require their joint participation. A common subcalulus is the *asynchronous* one, which is more appropriate for most of the concurrent systems. This is implemented in π-calculus by having the process $\mathbf{0}$ after an output. For example $\overline{a}\langle u\rangle$ is asynchronous but $\overline{a}\langle u\rangle.b(v)$ is not. Another, *localised* π-calculus, denoted by $\mathrm{L}\pi$, imposes that the names received on a channel have only the output capability. The process $a(x).b(y)$ and $a(x).b(y).\overline{x}$ satisfy the constraints but $a(x).x$ does not. Both sub-calculi are described in details in [SW01]. The type systems we will describe in the following sections are defined to either one or both of these sub-calculi.

3 Subtyping and Type Systems for Termination

In this section we present type systems in which types are associated to names.

When typing a process we can either use typing “a la Church” or “a la Curry”. When using “Church” typing, we assume that the typing context is aware of all the associations between a name and an unique type. It is therefore like an oracle which returns the type of a name. In contrast, when typing “a la Curry”, the associations *(name, type)* are added to the context using the inference rules. As the goal of this section is to present notions necessary further on, we adopt the presentation of [Dem10]. The notations introduced are easier to understand in a typing “a la Church”, especially the ones used for the termination proof.

3.1 Simple Types

We define types as:

$$T \quad ::= \quad \sharp T \mid \mathbb{U} \ ,$$

where $\mathbb{U}$ stands for the `unit` type able to carry only the $\star$ value.

The typing rules are:

$$\frac{}{\vdash \mathbf{0}} \qquad \frac{\vdash a : \sharp T \qquad \vdash b : T \qquad \vdash P}{\vdash \overline{a}\langle b\rangle.P} \qquad \frac{\vdash a : \sharp T \qquad \vdash x : T \qquad \vdash P}{\vdash a(x).P}$$

$$\frac{\vdash a : \sharp T \qquad \vdash x : T \qquad \vdash P}{\vdash !a(x).P} \qquad \frac{\vdash a : T \qquad \vdash P}{\vdash (\boldsymbol{\nu} a)\, P} \qquad \frac{\vdash P_1 \qquad \vdash P_2}{\vdash P_1 | P_2}$$

Example 5 *1. Let $P = !a(x).\overline{x}\langle u\rangle \mid \overline{a}v$. The process is typable with the following type assignments to the names:*

$$\vdash a : \sharp\sharp T \qquad \vdash v : \sharp T \qquad \vdash x : \sharp T \qquad \vdash u : T$$

2. If $P = a(x).a(y).y(x)$, then P is not typable as $\vdash x : T$, $\vdash y : \sharp T$ and a has to transmit both names.

Definition 6 (Termination) *A process P* diverges *if there exists an infinite sequence of processes $(P_i)_{i\geq 0}$ such that $P = P_0$ and for any i, $P_i \longrightarrow P_{i+1}$.*

P terminates *(or P is terminating) if P does not diverge.*

The problem of termination is a consequence of the usage of replicated input. A reduction consumes from the prefixes of a process, therefore a grammar without replication would force all process to have terminating reductions. However, introducing replicated inputs can lead to loops, as we have seen in the example $!a(x).\overline{a}\langle u\rangle$.

3.2 The level based approach of [DS06]

The following type system is able to type a considerable number of processes for which it guarantees termination, however it cannot type all terminating processes. The type system and its proof for termination are presented in details in [Dem10], we will only summarise the key points here.

The main idea is to annotate types with an integer, representing a *level*. Accordingly the grammar of types is given by:

$$T \quad ::= \quad \sharp^k T \mid \mathbb{U}.$$

The typing rules are:

$$\frac{}{\vdash \mathbf{0} : 0} \qquad \frac{\vdash a : \sharp^k T \qquad \vdash b : T \qquad \vdash P : w}{\vdash \overline{a}\langle b\rangle.P : \max(k, w)} \qquad \frac{\vdash a : \sharp^k T \qquad \vdash x : T \qquad \vdash P : w}{\vdash a(x).P : w}$$

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

We remark the following:

– the type of a process is its weight while the one of a name contains its level;

- only the names used as output channels can participate in the type of a process, as in the rules for input and replicated input the level of the names used as input channel do not interfere with the weight of the process;
- the main difference with the type system of simple types is given by the rule on the replicated input, where we enforce that the cost of the input to be strictly greater than the weight of the freed process. For example in $!a(x).\bar{b}\langle u\rangle$ we have the condition $lvl(a) > lvl(b)$. It is the one ensuring termination, as it is the only one that uses the relation $>$ between levels.

Example 7 *Let $P = !a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid \bar{c}) \mid \bar{a} \mid \bar{c}$. The process is typable with the following type assignments to the names:*

$$\vdash a : \sharp^3 T_a \qquad\qquad \vdash b : \sharp^2 T_b \qquad\qquad \vdash c : \sharp^1 T_c$$

The weight of a process is unfortunately not enough to prove termination. Consider the example above, where a possible reduction sequence is:

$$!a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid \bar{c}) \mid \bar{a} \mid \bar{c} \longrightarrow !a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid \bar{c}) \mid \bar{c} \mid \bar{b} \mid \bar{b} \mid \bar{c} \longrightarrow$$

$$!a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid \bar{c}) \mid \bar{c} \mid \bar{b} \mid \bar{c} \mid \bar{c} \mid \bar{c} \longrightarrow !a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid \bar{c}) \mid \bar{c} \mid \bar{c} \mid \bar{c} \mid \bar{c} \mid \bar{c} \mid \bar{c}.$$

The weight of P is originally 3, after the first reduction is 2, but decreases at 1 only after two more reductions. Therefore the weight is not necessarily decreasing at each reduction.

Nevertheless there is something that does decrease with each reduction, which is the *multiset of available outputs.*

Definition 8 *An available output is a name used as an output channel and that contributes to the weight of a process. Consequently, the output channels situated under a replication are not considered.*

The multiset of available outputs of P, noted $\mathrm{Os}(P)$, is defined by:

$$\mathrm{Os}(0) = \emptyset \qquad \mathrm{Os}(P_1|P_2) = \mathrm{Os}(P_1) \uplus \mathrm{Os}(P_2) \qquad \mathrm{Os}(\bar{a}\langle b\rangle.P) = \{k\} \uplus \mathrm{Os}(P), if\, \Gamma(a) = \sharp^k T$$

$$\mathrm{Os}((\boldsymbol{\nu} a)\, P) = \mathrm{Os}(P) \qquad \mathrm{Os}(a(x).P) = \mathrm{Os}(P) \qquad \mathrm{Os}(!a(x).P) = \emptyset$$

In the example above, we can rewrite the reduction sequence, by replacing the processes with their corresponding multiset:

$$\{3,1\} \longrightarrow \{2,2,1,1\} \longrightarrow \{2,1,1,1,1\} \longrightarrow \{1,1,1,1,1,1\}.$$

Therefore we can use multiset ordering as a decreasing measure in order to prove termination.

Definition 9 *The multiset ordering denoted as $>_{mul}$ is defined as $M_1 >_{mul} M_2$ if N being the maximal multiset such that $M_1 = N \uplus N_1$, $M_2 = N \uplus N_2$ for all $e_2 \in N_2$ there is $e_1 \in N_1$, $e_1 > e_2$. The relation $>_{mul}$ is well founded.*

Example 10 *We have $\{3,1\} >_{mul} \{2,2,1,1\}$ as $\{3,1\} = \{1\} \uplus \{3\}$, $\{2,2,1,1\} = \{1\} \uplus \{2,2,1\}$ and $3 > 2$, $3 > 1$.*

The following theorem permits one to use $\mathrm{Os}(P)$ to prove termination, as it shows that the set of available outputs decreases with each reduction.

Theorem 11 ([Dem10]) *If $\Gamma \vdash P : w$ and $P \longrightarrow P'$ then $\mathrm{Os}(P) >_{mul} \mathrm{Os}(P')$.*

4 Our Type Systems for Termination of Concurrent Processes

In the following we present the main result of our work which consists of a type system more expressive than the one of section 3.2. This section is divided into 5 smaller sections. First, in subsection 4.1, we define the type system, present alternatives and motivate our choices. In the second subsection 4.2, we focus on its properties, especially on the termination one. Subsection 4.3 presents examples of processes that are interesting tests for the expressiveness of our type system. As expected, not all terminating processes are typable in our system, and in subsection 4.4 we look at a way of increasing its expressiveness. Finally, in subsection 4.5, we discuss the question of type inference.

Our work started as a quest to find a type system for programming languages such as Erlang, in which the interprocess communication is asynchronous. Therefore we will continue our discussion using asynchronous π-calculus.

The previous type system introduces levels in order to find a decreasing measure over a process' reduction. The key point of our work is that we allow subtyping on levels. Consider the following example $P = !a(x).\overline{x}\langle t\rangle.P \mid \overline{a}\langle y\rangle \mid Q$. The constraints imposed are $lvl(a) > lvl(x)$, $\Gamma(a) = \sharp^{lvl(a)}\sharp^{lvl(x)}T$ and $\Gamma(x) = \sharp^{lvl(x)}T$. The process is therefore typable only if x and y have the same type. We can send on the channel a a name with a smaller level than x but, necessarily, of the same type. In our type system we allow y to have a different type as long as it smaller than the type of x. For ordering the types we use i/o subtyping such that the types of a and x now become: $\Gamma(a) = \sharp^{lvl(a)}o^{lvl(x)}T$ and $\Gamma(x) = o^{lvl(x)}T$. We can infer for y any type smaller than $o^{lvl(x)}T$, and allow therefore more terminating processes to be typable.

We change the presentation from typing "a la Church" to typing "a la Curry". We motivate the change in Remark 28.

4.1 Definition of the type system

In i/o subtyping, one uses three types for names: input, output and the connection type $\sharp$. The capability of using a channel as both input and output is a subtype of using it either as an input or as an output, as stated by the rules SUBT-#I and SUBT-#O below. If T' is a subtype of T, written as $T' \leq T$, then a name of type T' can always replace a name of type T. In order to better understand, consider the following analogy, where Int is a subtype of $Real$, written $Int \leq Real$. Any operation defined on $Real$ is also defined on Int, while the reverse is not true. For example the successor of a $Real$ is not defined while for an Int it is trivial.

Types are defined by the grammar:

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

Example 12 *1. In $a(x).\overline{x}\langle u\rangle$, the type of a is $i^n o^m T$, and accordingly, a can only receive a name x which has at most the output capability.*

2. In $a(x).\overline{x}\langle u\rangle \mid \overline{a}\langle v\rangle$, with $\Gamma(a) = \sharp^n o^m T$, a can both send and receive a name with the output capability.

3. For $a(x).a(y).x(z).\overline{y}\langle u\rangle$ the type of a is $i^n \sharp^m T$. Consequently a can only receive names that have both capabilities.

The subtyping relation is a preorder relation similar to the one of i/o subtyping of [PS96] and defined by the following inference rules:

$$\textsc{Refl}\ \frac{}{T \leq T} \qquad \textsc{Trans}\ \frac{T_1 \leq T_2 \qquad T_2 \leq T_3}{T_1 \leq T_3} \qquad \textsc{Subt-\#I}\ \frac{}{\sharp^k T \leq i^k T} \qquad \textsc{Subt-\#O}\ \frac{}{\sharp^k T \leq o^k T}$$

$$\textsc{Subt-II}\ \frac{T \leq S \qquad k_2 \leq k_1}{i^{k_1} T \leq i^{k_2} S} \qquad \textsc{Subt-OO}\ \frac{T \leq S \qquad k_1 \leq k_2}{o^{k_1} S \leq o^{k_2} T}$$

We say that the input is covariant, as its argument preserves the direction of subtyping. Accordingly, the output is contravariant, and $\sharp$ is invariant, as it forces type equality. We can, for example, send an Int when a $Real$ is expected, and receive a $Real$ instead of an Int. However the variance changes for the levels. In order to ensure termination, we can have a smaller level on an output name and a bigger one on an input.

Example 13 *1. In $!a(x).!x(y).\overline{b}\langle z\rangle$ we have the condition $lvl(x) > lvl(b)$. We can send on the channel a names that have the input capability and the level greater or equal to $lvl(x)$.*

2. For $!a(x).\overline{x}\langle t\rangle$, with $lvl(a) > lvl(x)$, we can send on the channel a names with the output capability and a level smaller or equal to $lvl(x)$.

Remark 14 *We can rewrite the rules* Subt-II *and* Subt-OO *in another form, in order to stress their variance:*

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

The rules for typing the names are:

$$\frac{\Gamma(a) = T}{\Gamma \vdash a : T} \qquad \overset{\textsc{Subsum}}{\frac{\Gamma \vdash a : T \qquad T \leq U}{\Gamma \vdash a : U}}$$

The second rule of name typing is the only one that uses the subtyping relation. To understand what it states consider the process $a(x).\overline{x}\langle u\rangle \mid \overline{a}\langle b\rangle$. Since a is used as both an input and an output channel, its type is $\sharp^k T$. When applying the typing rules on $a(x).\overline{x}\langle u\rangle$ we will need only the input capability of a, therefore we use the subtyping relation to infer $\Gamma \vdash a : i^k T$.

The inference rules for typing processes are:

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

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

The rules of Section 3.2 in an asynchronous calculus can be derived from the new ones, by replacing i and o with $\sharp$. As it can be noted the main difference between the two type systems is that in ours we check that names have the right capabilities. We make the following observations on the rules above:

- When typing $\overline{a}\langle v\rangle$, we are allowed to change the type of both the channel, a, and of the name carried, v, using rule SubSum. In contrast, in $a(x)$ we cannot change the type of x. Therefore a valid type assignment for a, when a is used as both input and output channel, will keep the type of the names carried on an input channel and try to adapt the type of the ones carried on an output channel.
- Rule Out-S is the one responsible for the asynchronous communication.
- When processes are in a parallel composition they will share the same typing context instead of splitting it between them. The choice of either sharing the context or not is a consequence of typing "a la Curry" and it is not a particularity of these typing rules.

Example 15 *Consider the process $!a(x).!x(y).P \mid \overline{a}\langle v\rangle$. We can type it as follows:*

$$\Gamma \vdash a : \sharp^{k_a}(\sharp^{k_x} T), \Gamma \vdash x : \sharp^{k_x} T, \Gamma \vdash v : \sharp^{k_v} T \qquad\qquad \Gamma \vdash P : w, w \geq 0.$$

According to the typing rules we have that $w < k_x$ but not $w < k_v$, which is necessary in order to make the process $!a(x).!x(y).P \mid !v(y).P[v/x]$ typable. In our type system we have that $\sharp^{k_x} T = \sharp^{k_y} T$ and $w < k_y$.

The equality is a result of the fact that we typed x as $\sharp^{k_x} T$. We can also write $\Gamma \vdash x : i^{k_x} T$ and $\Gamma \vdash y : i^{k_y} T$, with $k_x \leq k_y$. Using the rules Subsum *and* Subt-II *we get $\Gamma \vdash y : i^{k_x} T$. The process $!v(y).P[v/x]$ is still typable as $k_y \geq k_x > w$.*

In the following examples, we discuss how the variance of $\sharp$ on levels, can lead to inconsistency in our type system, and thus show our motivation for keeping it invariant. Note that its invariance on types is a property of the i/o subtyping of [PS96]. A covariant or contravariant $\sharp$ on types can lead to run-time errors.

Example 16 *We will exemplify on processes which, after a reduction, are no longer typable for obvious reasons.*

1. *Covariance. In the process $P = !a(x).!x(y).\bar{t} \mid \bar{a}\langle u\rangle$ the types can be $\Gamma(x) = i^1T$, $\Gamma(u) = i^0T$ and $\Gamma(a) = \sharp^0 i^1 T$. After a reduction the process becomes $P' = !u(y).\bar{t}$, where $lvl(u) = lvl(t)$, making the resulting process not typable. We cannot type P either since in the rule* Out-S *for $\bar{a}\langle u\rangle$, we cannot have $\Gamma \vdash a : o^0 i^0 T$ nor $\Gamma \vdash u : i^1 T$. We can however write the type of a in a more general form as $\Gamma(a) = \sharp^0\sharp^1 T$, keeping the process typable.*
   Consider now that $\sharp$ is covariant. Then we have that $\sharp^0\sharp^1T \leq o^0\sharp^1T \leq o^0\sharp^0T \leq o^0 i^0 T$ and we can therefore type P', which leads to inconsistencies in our type system.
2. *Contravariance. We have that $P = a(x).!b.\bar{x} \mid \bar{a}\langle u\rangle \mid !u.\bar{b}$ reduces to $P' = !b.\bar{u} \mid !u.\bar{b}$. With the types inferred for the names in P, $\Gamma(x) = o^0T$, $\Gamma(u) = \sharp^2 T$ and $\Gamma(a) = \sharp^0 o^0 T$, we cannot type the process. We can change the type of a to $\sharp^0\sharp^0T$.*
   Let $\sharp$ be contravariant. We then can deduce the following inequalities: $\sharp^0\sharp^0T \leq o^0\sharp^0T \leq o^0\sharp^2T$, making P typable.

We remark that our type system is more expressive than the one of Section 3.2, as shown by the following example.

Example 17 *Consider the terminating process $!a(x).\bar{x}\langle t\rangle \mid \bar{a}\langle p\rangle \mid \bar{a}\langle q\rangle \mid !p(z).\bar{q}\langle z\rangle$. Since the types of p and q are different ($lvl(p) > lvl(q)$) the process is not typable in the type system of Section 3.2. In our type system we have:*

$$\Gamma(a) = \sharp^k o^1 T \qquad \Gamma(x) = o^1 T \qquad \Gamma(p) = \sharp^1 T \qquad \Gamma(q) = o^0 T$$

For $\bar{a}\langle p\rangle$ we have $\sharp^k o^1 T \leq o^k o^1 T$ and using subtyping on p, $\Gamma \vdash p : o^1 T$. Typing $\bar{a}\langle q\rangle$ is straightforward as $\sharp^k o^1 T \leq o^k o^1 T \leq o^k o^0 T$.

4.2 Properties of the Type System

In order to ease the presentation, the properties of the type system are only stated and rarely proved. The complete proofs are available in [CH11].

The following Lemma is similar to the rule SubSum for processes.

Lemma 18 *If $\Gamma \vdash P : w$ and $w \neq 0$ then, for any $w' \geq w$, $\Gamma \vdash P : w'$.*

Proof. We reason by induction on the typing derivation. We only consider the case $P = \bar{a}\langle b\rangle$. We have that $\Gamma \vdash a : o^k T$ and $\Gamma \vdash b : T$. Using rule SubSum we can build the following derivation:

$$\dfrac{\dfrac{\Gamma \vdash a : o^k T \qquad \dfrac{T \leq T \qquad k \leq k'}{o^k T \leq o^{k'} T}}{\Gamma \vdash a : o^{k'} T} \qquad \Gamma \vdash b : T}{\Gamma \vdash \bar{a}\langle b\rangle : k'}$$

Proposition 19 (Narrowing) *If $\Gamma, x : T \vdash P : w$ and $T' \leq T$, then $\Gamma, x : T' \vdash P : w'$, $w' \leq w$.*

The subsequent three properties show that congruence, name substitution and reduction preserve process typability.

Lemma 20 *If $P \equiv Q$, then $\Gamma \vdash P : w$ iff $\Gamma \vdash Q : w$.*

Lemma 21 *If $\Gamma, x : T \vdash P : w$ and $\Gamma \vdash b : T$ then $\Gamma \vdash P[b/x] : w'$, for some $w' \leq w$.*

Theorem 22 (Subject reduction) *If $\Gamma \vdash P : w$ and $P \longrightarrow P'$, then $\Gamma \vdash P' : w'$ for some $w' \leq w$.*

Theorem 22 states that the weight of a process is not enough for proving termination, as it does not decrease with each reduction. Therefore we borrow the definitions of the set of available outputs and the multiset ordering from [Dem10], presented in Section 3.2. We however encounter a difficulty as we are working in a typing "a la Curry". That entails that we do not have a typing context in which names and their types are known, and which can return the level of a name, upon request. Instead we have to build the multiset along with the typing derivation. Two processes P and P' might share the same context Γ, and have common types for all their free names, but infer different types for the bound names. Therefore we have to consider all possible derivations. In order to find a decreasing measure, and thus prove type soundness, we search for the minimum type assignment among all the derivations that can be inferred using a given typing context.

We define the multiset by induction over the derivation of a typing judgement. 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 $\Gamma \vdash P : w$.

Definition 23 *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:*

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

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

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

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

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

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

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

The following three Lemmas show how $\mathcal{M}_\Gamma(P)$ behaves within process congruence, name substitution and reduction. Only the reduction strictly decreases the measure, as it is the only property out of the three, that is required in the termination proof.

Lemma 24 *Let $\Gamma \vdash P : w$, $\Gamma(x) = T$ and $\Gamma(v) = T'$. If $T' \leq T$ then $\mathcal{M}_\Gamma(P) \geq_{mul} \mathcal{M}_\Gamma(P[v/x])$.*

Lemma 25 *If $\Gamma \vdash P : w$, $\Gamma \vdash Q : w'$ for some w, w', and $P \equiv Q$ then $\mathcal{M}_\Gamma(P) = \mathcal{M}_\Gamma(Q)$.*

Lemma 26 *If $\Gamma \vdash P : w$ and $P \longrightarrow P'$, then $\mathcal{M}_\Gamma(P) >_{mul} \mathcal{M}_\Gamma(P')$.*

Theorem 27 (Soundness) *If $\Gamma \vdash P : w$, then P terminates.*

Proof. Suppose that P diverges. Therefore there is an infinite sequence$(P_i)_{i \in N}$, where $P_i \longrightarrow P_{i+1}$, $P = P_0$.

According to Theorem 22 every P_i is typable. Using Lemma 26 we can write that $\mathcal{M}_\Gamma(P_i) >_{mul} \mathcal{M}_\Gamma(P_{i+1})$ for all i, which yields to a contradiction.

Remark 28 *Type systems for termination, on which we based our work use typing "a la Church", as we have seen in Section 3.2, while i/o type systems use typing "a la Curry". Having an oracle as a typing context, requires additional conditions on the names that are rather subtle, but that are easier to understand when using typing "a la Curry". Consider the rule* Inp-S *of $a(x).P$ on which typing "à la Curry" forces one to derive the type of x from the type of a. Such a constraint is not necessarily implied when typing "à la Church", and one can be tempted to use subtyping on x, which is not permitted.*

4.3 Encodings of the λ-calculus

In the following we will exhibit a possible interpretation of simply typed λ-calculus terms in π-calculus. Translating functions of λ-calculus into processes is of great interest in the study of process calculi, as it creates a bridge between a sequential model for computation and a parallel one. It allows us to reason about functions in a concurrent environment, we can apply techniques from one calculus to another, or it can help us in modelling programming languages that offer constructs for both functions and concurrency. In particular, we are interested in representing simply typed λ-terms as processes typable by the types system presented so far, as this is a testimony of their expressiveness. Ideally, a type system should be able to infer types for all simply typed λ-terms, but as we will show, this is not always the case. However, before looking at such examples, we have to present a brief introduction to the interpretation of λ-terms as π-calculus processes. A more detailed description can be found in [SW01].

We will work in the simply typed λ-calculus, defined by:

$$M \quad ::= \quad x \mid \lambda x.M \mid M\ N.$$

In π-calculus, we can model abstractions as two communicating processes, out of which one is the function and the second is its argument. Consequently, a β-reduction is the interaction between them. A term becomes available through the name of a channel. An important difference between an abstraction in λ-calculus and its corresponding processes, is that the latter ones are placed in an concurrent environment, where they can engage in communications with other processes. We prevent this by creating new, private channels used solely for their communication. In this manner we ensure that the only exchange possible is through that channel only, and that we keep the term unaltered by the environment.

There are several possible strategies to interpret a λ-term corresponding to the λ-calculus reduction strategies: call-by-value, call-by-name and call-by-need. We chose the first one, but we think any of them would have given similar results.

In the call-by-value strategy we use the following rules for reduction:

$$V \quad ::= \quad \lambda x.M \mid x \qquad\qquad \frac{}{(\lambda x.M)V \longrightarrow M\{V/x\}} \qquad\qquad \frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N} \qquad\qquad \frac{N \longrightarrow N'}{V\ N \longrightarrow V\ N'}$$

We use $[\![M]\!]_p$ for the encoding of a λ-term M parametric on the name p. As we mentioned above, the process corresponding to M becomes available through interaction with the channel p. For the encoding we have the following rules:

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

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

We encode a value, $[\![x]\!]_p$, by simply transmitting it on the channel p. The term $\lambda x.M$ returns a function. In its encoding, instead of transmitting a function, which is not allowed in π-calculus, we transmit the function's location, acting as a pointer. To perform a computation one has to pass on the argument of the function and retrieve the result on the channel given as pointer. Thus a function is interpreted as a resource which can be accessed as many times as we want by passing to it its argument.

For the term $M\ N$ we chose the encoding of *parallel* call-by-value, in which the terms M and N are reduced in parallel, without making M wait for N to finish. In this manner we simplify the obtained process. In our approach, the differences between the two encodings are insignificant.

However, the parallel and sequential strategies yield equivalent results. Once both terms finished execution the result returned by N is fed as an argument to M.

The type system of [DS06], on which we based our approach is not able to type all the terms of simply typed λ-calculus. Consider the following example, belonging to [DHS09]:

Example 29 (From [DHS09]) *Consider the λ-term $M_1 \overset{def}{=} f\ (\lambda x.(f\ u\ (u\ v))$ which can be typed in simply typed λ-calculus as $f : (\sigma \longrightarrow \tau) \longrightarrow \tau \longrightarrow \tau, v : \sigma, u : \sigma \longrightarrow \tau$.*

Computing the encoding of this term according to the above definition yields the process:

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

We make the following observations :

- $\overline{q}\langle f\rangle \mid ... \mid \overline{r}\langle y\rangle \mid q(f').r(z).\overline{f}'\langle z,p\rangle$ *implies that f and f' have the same type $T_f = f : o^k(T_y, T_p)$.*
- *from $\overline{q}_2\langle f\rangle \mid \overline{r}_2\langle u\rangle \mid q_2(f_2).r_2(z_2).\overline{f}_2\langle z_2, q_2\rangle$ we have that $T_f = f : o^k(T_u, T_{q_1})$.*
- $k_y > max(k_f, k_u)$

In the type systems of [DS06,Dem10], this necessarily entails $T_u = T_y$, which is in conflict with $k_y > k_u$.

The problematic subterm can be typed in the type system of Section 4.1 as follows:

$$
\dfrac{\varGamma \vdash f : o^k(\sharp^{k_y} T, \_) \qquad \dfrac{\varGamma \vdash u : o^{k_u} T \qquad \dfrac{k_u \le k_y \qquad T_y \le T_u}{o^{k_u} T \le o^{k_y} T}}{\varGamma \vdash u : o^{k_y} T_y}}{\varGamma \vdash \overline{f}\langle u, \_\rangle : k}
$$

where we can take $T_y = T_u$ for the sake of simplicity.

The example above is typable in our type system. We can however provide an example of process not typable in neither type systems.

Example 30 *Consider the term*

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

We want to prove that its translation in π-calculus is not typable. We can use Theorem 22 to state that for two processes P and P', where $P \longrightarrow P'$, if P' is not typable then P is not typable either. Therefore, in order to ease presentation, instead of considering the encoding of M_2, we look at a smaller process, obtained after some 'administrative reductions' have been performed:

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

On this process we can perform reductions corresponding to β-reductions, which reduce not only the π-calculus processes, but also their image in λ-calculus. We refer to the other reductions, as the administrative ones. We obtain a process which contains

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

as a subterm. By further performing β-reductions, we have the cleaner process:

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

which is the one we will try to type.

This process is not typable in type system of Section 3.2 because we need to send to u of $\Gamma \vdash u : \sharp^{k_u} o^{k_x} T$ the name v of type $\sharp^{k_v} T$.

In our type system, using subtyping we can rewrite $\Gamma \vdash v : o^{k_v} T$. But the rule Subt-OO *keeps the inequality between levels and therefore we cannot send a heavier process than expected. We conclude that this example is not typable in our type system, either.*

However, by looking at the sequence of reductions, we can remark that the process above is terminating: $(\boldsymbol{\nu} u)\left((\boldsymbol{\nu} v)\left(\overline{u}\langle v\rangle \mid !v.\overline{u}\langle t\rangle\right) \mid !u(x).\overline{x} \longrightarrow \overline{v} \mid !v.\overline{u}\langle t\rangle \mid !u(x).\overline{x} \longrightarrow \overline{u}\langle t\rangle \mid !v.\overline{u}\langle t\rangle \mid !u(x).\overline{x} \longrightarrow \overline{t} \mid !v.\overline{u}\langle t\rangle \mid !u(x).\overline{x}\right.$.

4.4 Subtyping and functional names

As we have seen in the previous examples, our type system is not expressive enough to accommodate all the processes of simply typed λ-calculus. In the following we present a possible adjustment to the typing rules that would permit typability for all such λ-terms. Our approach is based on [DHS10]. Once more a brief introduction on functional π-calculus is necessary.

Functional π-calculus
We consider the names as belonging to two disjoint subsets, imperative and functional. The subset of functional names is the result of the encoding of strong normalising simply typed λ-calculus into π-calculus. These are names that offer a service that is always available and is the same every time, but impose some constraints on the typing rules. When using both sets of names we say that we work in an impure language.

In the work of [DHS10] the imperative names are used in typing rules similar to the ones of Section 3.2. The functional names, however are employed only in the following two rules:

$$\textsc{Out-F}\quad \frac{\Gamma \vdash a : \sharp^k T \qquad \Gamma \vdash v : T}{\Gamma \vdash \overline{a}\langle b\rangle : k} \qquad\qquad \textsc{Def}\quad \frac{\Gamma, x : T \vdash P_1 : l \qquad \Gamma \vdash P_2 : l' \qquad \Gamma \vdash f : \sharp^k T \qquad k \geq l \qquad f \notin \mathrm{fn}(P_1)}{\mathrm{def}\ f = (x).P_1\ \mathrm{in}\ P_2 : l'}$$

The construction def $f = (x).P_1$ in P_2 is the equivalent of $\boldsymbol{\nu} a.(!a(x).P_1 \mid P_2)$ for imperative names. Using such a construction helps us in imposing the necessary constraints of functional names in a more natural way. Its meaning is similar to the keyword *let* in Caml. Rule Def states that:

- We do not allow recursion on a functional name. This is ensured by the condition $f \notin \mathrm{fn}(P_1)$.
- Functional names do not appear in an input that is not replicated.
- The nested functional names are ordered. In the process $!f_1(x_1).P_1|!f_2(x_2).P_2|!f_3(x_3).P_3$, where f_1, f_2, f_3 are functional names, we have that in P_3 can appear f_1, f_2, in P_2 only f_1 and in P_1 none of them.

Those are constraints required for a functional name. Besides ensuring them, rule Def also requires that the functional names are in Lπ-1. This means that when receiving a functional name we cannot use it in input. Moreover, there is exactly one replicated input on that name. This is a consequence of creating a fresh name for each f in *def* $f = \ldots$.

Integrating functional π-calculus in our type system
In an impure language, one has to distinguish between the functional and imperative names in order to treat them accordingly. In [DHS10], this is done by assigning tags and thanks to the `def` construct. I/o subtyping allows us to treat names differently, based on their capabilities. Therefore we can impose the constraints of functional names using the input and output capabilities instead of differentiating names based on the set they belong to.

We present below some of the more interesting typing rules and we follow with their explanation:

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

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

The simplest condition we have to ensure on functional names, is not to allow recursion. In $!f(x).P$, we remove f from the environment when typing P. The second condition, that a functional name cannot appear in a normal input, is a natural one to impose in a i/o type system. After using f as a replication, we keep it in the environment with only its output capability. However, since the typing rules, as presented above, do not differentiate between a replicated and a normal input, this alone does not guarantee that f behaves like a functional name. Therefore we ensure $\Gamma(f) = o^k T$, throughout its usage. An additional typing rule is added allowing functional names, which only have the output capability, to perform replications.

This method of ensuring the second condition, makes our type system more expressive than the one of [DHS10], as functional names are no longer in Lπ-1. We can perform a replicated input on f as long as it is in the typing context, without creating a fresh name each time. For example we can type processes such as $(\boldsymbol{\nu} f)\,(!f(x).P \mid !f(y).Q \mid R)$. This is not possible using the `def` construct.

The procedure to type a process in an impure language requires first to decide which names are functional and which are not. The imperative names are created with both the input and output capabilities, while the functional ones only with the output capability. In this manner, we avoid using tags to differentiate between the names, but we still need to decide beforehand whether a name is functional or not. In either cases, there are several possible typings for the same process in which names are treated differently.

The last condition is the least natural one for our approach. We can write the construct of a functional name in the form $\boldsymbol{\nu} f.!f(x)$, which guarantees that each time a functional name is used, it is the most recent one created. We ensure that by working on a typing environment of the form $\Gamma \bullet f : o^k T$, in which the name f is isolated from the context Γ. In the typing rules, we always keep a single functional name separated, representing the most recent one added to the context. However, this form of environment can break the condition on recursion. A process $!f(x).P$, where $P = E[!f(y)]$ is not allowed, and therefore when typing P, the typing context becomes $\Gamma \bullet -$. Another situation in which we employ $\Gamma \bullet -$ is when an imperative name is used in input. Consider the process $c(x).!f(y).\overline{x}\langle y\rangle \mid \overline{c}\langle f\rangle \mid \overline{f}\langle v\rangle$, which reduces to $!f(y).\overline{f}\langle y\rangle \mid \overline{f}\langle v\rangle$ and breaks the conditions on functional names. Therefore we do not allow any usage of functional names under an input.

Theorem 31 (Soundness) *If $\Gamma \bullet f : o^k T \vdash P : w$, then P terminates.*

Proof (sketch). *The proof has the same structure as the corresponding proof in [DHS10]. An important aspect of that proof is that we exploit the termination property for the calculus where all names are functional without looking into it. To handle the imperative part, we must adapt the proof along the lines of the termination argument for Theorem 27.*

The complete resulting type system is available in [CH11].

4.5 Type inference

Along with the expressiveness of our type system, we are also interested in finding an efficient procedure for type reconstruction. In our type system such a procedure is nondeterministic. However, by using a more restrictive version of the same type system, we have a polynomial time inference procedure.

Type inference for Lπ

We work on a subsystem of the one presented in Section 4.1, which belongs to Lπ. This restriction makes sense when considering the communication in Erlang, for instance. An input channel corresponds to an Erlang process, which can receive the identifier of another process, but cannot create it dynamically. Therefore, it cannot emulate the behaviour of a received name with the input capability. A more detailed description is available in section 5. The encoding of λ-calculus of section 4.3 also belongs to Lπ.

The types become:

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

We modify the typing rules such that any name a has either the type $\sharp^k S$ or $o^k S$. We can still use the rules with $i^k S$, by applying subtyping.

Lπ limits the usage of subtyping. We can only apply it for output channels, like $\overline{a}\langle u\rangle$, with $\Gamma(a) = o^k S$ where a can carry names of type S or smaller. Consequently in the typing rules for the input, $a(x).P$ or $!a(x).P$, we read the type of a from the context and disallow the usage of the rule Subsum on it.

We still have a greater expressivity than the type system of section 3.2, as it is shown in the example 17. Therefore working in Lπ is a manner of simplifying i/o subtyping, while keeping some aspects of the flexibility brought by our system.

The inference procedure for simple types of [Dem10] is polynomial w.r.t. the size of the process, measured in number of prefixes. In our type inference we take the results of the aforementioned procedure and add a minimal level assignment.

We only present the inference procedure and some examples. Further discussion and the proofs for its correction are available in [CH11].

In order to infer the types of a process P we build a graph depicting the relations between the names appearing in P. We first create the nodes, as follows:

- for every name n, such that $n \in \mathrm{fn}(P)$ or $P = E[\boldsymbol{\nu} n.Q]$ we create one node, labelled with n, and a second one, labelled with $\mathrm{son}(n)$. If n has type $\sharp^k S$, $\mathrm{son}(n)$ has type S.
- for every name x, such that $P = E[a(x).Q]$, let $a = \mathrm{father}(x)$, add x as a label to $\mathrm{son}(a)$.

Example 32 *We associate to the process $P = a(x).(\boldsymbol{\nu} b)\,\overline{x}\langle b\rangle \mid !a(y).(\overline{c}\langle y\rangle \mid d(z).\overline{y}\langle z\rangle)$ the following set of 8 nodes with their labels: $\{a\}, \{\mathrm{son}(a), x, y\}, \{b\}, \{\mathrm{son}(b)\}, \{c\}, \{\mathrm{son}(c), y\}, \{d\}, \{\mathrm{son}(d), z\}$.*

We then add the edges, as follows:

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

Example 33 *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:

$$\begin{tikzcd}
a & b & c \arrow[l, "<"'] \\
\mathrm{son}(a) \arrow[ru, "\geq"] \arrow[rru, "\geq"'] & \mathrm{son}(b) \arrow[r, "\geq"] & \{\mathrm{son}(c), z\}
\end{tikzcd}$$

A process is not typable if the graph contains a cycle involving at least one $\xrightarrow{\geq}$ edge. Otherwise, we can assign a level to each node, by starting with the leaves and go up. Each time we encounter a $\xrightarrow{\geq}$ edge we increment the level.

Example 34 *On the graph of Example 33, the procedure assigns level 0 to nodes $a, b, \mathrm{son}(b)$ and $\{\mathrm{son}(c), z\}$ and level 1 to c and $\mathrm{son}(a)$ This yields the typing $b : o^0 o^0 T, c : \sharp^1 o^0 T, a : o^0 o^1 o^0 T$ for the process of Example 33.*

Type inference for our type system
A consequence of working with i/o subtyping is that one can infer several possible type assignments. Assuming that we are in the type system of [PS96] (i/o without the levels), the type of a in $a(x).\overline{x}$ can have several forms among which ioT is the most restrictive:

$$ioT \geq i\sharp T \geq \sharp\sharp T \qquad\qquad ioT \geq \sharp oT.$$

We also note that from ioT, all the other types can be deduce using subtyping. We say that these types are less precise, in the sense that they use more "$\sharp$". We would like therefore to exhibit type assignments, that are the most restrictive and that can generate all the other valid assignments.

Definition 35 *A* principal typing *for P is a typing environment Γ such that $\Gamma \vdash P : w$ and, whenever $\Gamma' \vdash P : w'$, $\Gamma' \leq \Gamma$, where the latter relation stands for $\forall a \in \mathrm{dom}(\Gamma), \Gamma(a) \leq \Gamma'(a)$.*

The i/o type system of [PS96], does not have a principal typing.

Example 36 *For process $Q_2 \overset{def}{=} \overline{t}\langle a\rangle \mid \overline{t}\langle b\rangle \mid !a.\overline{b}$ we have the following valid typing contexts $\Gamma_1 \overset{def}{=} t : oo\mathbb{U}, a : \sharp\mathbb{U}, b : o\mathbb{U}$ and $\Gamma_2 \overset{def}{=} t : oi\mathbb{U}, a : i\mathbb{U}, b : \sharp\mathbb{U}$, along with others, less precise. We cannot deduce one from the other, and both are using the same number of "$\sharp$".*

As we based our type system on the i/o one, we are not able to find a type inference procedure which yields a principal typing.

Late in our work we discovered a variation on the i/o type system, which has a principal typing [IK00]. Consider the process $t(a).a \mid t(b).\overline{b}$. The type $\Gamma(t) = i\sharp\mathbb{U}$ is the most restrictive one. It is a consequence of the fact that iT can be rewritten into iT' with $T' \leq T$. In order for $t(a)$ and $t(b)$ to be correct, the type under the input prefix in t, is the *glb* between the types of a and b. In our type system the *glb* is represented by $\#$. However, when we try to apply the same reasoning for the type under an output, one realizes that it needs the *lub* between input and output, which is not defined. In [IK00], such a type is introduced. We denote it with $\uparrow$. Using it, one can infer the following type assignment for example 36: $\Gamma \overset{def}{=} t : o\uparrow\mathbb{U}, a : i\mathbb{U}, b : o\mathbb{U}$.

It is unclear to us if combining this type system with level based termination leads to a principal typing. The $\uparrow$ capability, seems to be a notation introduced due to its convenient behaviour. Therefore we were not able to decide on its invariance and on whether it increases or not the expressiveness of a type system.

5 Implementation in Erlang and Caml

This section presents some details regarding an implementation in Erlang of the processes typed in the type system of section 4.1 and of a inference procedure for the same type system, this time in Caml.

Erlang offers the tools necessary for working with π-calculus processes. It has primitives for creating and handling processes, communication between them is easy to simulate and the Erlang processes emulate the expected behaviour of concurrent systems.

We created a tool, written in *Caml*, in which processes belonging to the type system of section 4.1 are translated into Erlang processes. In the following we only present the main ideas in our approach.

First, notice that Erlang is mostly using asynchronous communication between processes, and therefore our type system was constrained to asynchronous π-calculus. As in most programming languages a 'master' process is responsible for running the code. It simulates the environment behaviour using the following rules:

$$[\![P_1 \mid P_2]\!] = [\![P_1]\!] \mid [\![P_2]\!] \qquad [\![\boldsymbol{\nu} a.P]\!] = \texttt{register(a, spawn(program, channel, [0, 0])}$$

$$[\![a(x).P]\!] = \texttt{a ! \{input, x, }[\![P]\!]\texttt{ \}} \qquad [\![!a(x).P]\!] = \texttt{a ! \{input, x, }[\![!a(x).P \mid P]\!]\texttt{ \}}$$

$$[\![\overline{a}\langle u\rangle]\!] = \texttt{a ! \{output, u\}}$$

Whenever a new name is encountered a process is created using the function `spawn`. It requires the name of the program running, a pointer to the function the new process has to execute and its arguments. The function `channel` is common to all processes and is explained below. In Erlang, processes are completely independent one from another. In order for them to communicate they need to know each other identifiers. This is ensured by `register`.

We use the construct `Pid ! Message` to send a $Message$ to process Pid. In $a(x).P$, the process a waits for a message x to arrive before executing P. Therefore we send a message in which we specify the type of the message (either input or output) and a pointer to a function representing P's encoding. Similarly for $!a(x).P$, where the replication is treated as a new process, containing the replicated input. When sending a message with the output tag, we only attach to it the object u.

Example 37 *In the process $\boldsymbol{\nu} a.(a(x).P \mid \overline{a}\langle v\rangle)$, a channel a is created, on which we send an `input` and an `output` message:*

```
register(a, spawn(program, channel, [0, 0])),
(a ! {input, a, f00}), (a ! {output, p1})...
```

The function `channel` is the one responsible for receiving the `input`, `output` messages and for matching them accordingly. For storing these messages, we used a data structure called 'dictionary', which is a storage facility private to each process and to which one has access throughout its execution.

```
channel(N, M) ->
   receive
      {output, Msg} ->
       put(N, Msg), channel(N+1, M+1);
   ..........
```

When a message tagged with `output` is received, it is added to the queue, `put(N, Msg)` and the function is called recursively in order to wait for a new message.

Receiving an `input` tagged message requires more work. We distinguish between two situations according to whether or not an input message is the first to arrive. We keep track of the number of input (resp. output) messages received using a counter N (resp. M).

```
   ..........
{input, Me, Msg} ->
  if M == 0 -> %outputs are late
       receive {output, L} ->  put(N, L),
       end,
       apply(b, Msg, [L]),
       channel(N+1, 0);
   ..........
```

If there is no output message to match with the newly input message received, then the process has to wait for it to arrive. Afterwards it executes the function corresponding to the process under the input prefix, `Msg`. In Erlang, this is possible thanks to the function `apply`. Matching an input message with an output one, involves also identifying the names carried through those channels. Our approach is to assign an identifier to each message, and group the outputs and inputs that have the same identifier. Since we are using a queue to store them, the identifier of an input message (resp. output) is N (resp. M). Therefore we transmit to the function `Msg` the parameter L. After executing `Msg`, we increment the counter for the inputs, and wait for a new message.

Example 38 *We have that $a(x).P \mid \overline{a}\langle v\rangle \longrightarrow P[v/x]$. Msg is the function corresponding to the process P parametric on x, and $L = v$.*

```
 ..........
if M > 0 -> %inputs are late
     L = get(N-1),
     apply(b, Msg, [L]),
     channel(N, M-1).
```

If there are output messages waiting in the queue, the behaviour is very similar to the one described above, except that we no longer need to wait for another output message as we can take one from the queue.

Note that the processes representing the channels run continuously. In this manner implementing the replicated inputs requires simply to send a new input message whenever one is processed.

The λ-calculus to π-calculus translation was implemented in *Caml*. Also, the inference procedure for localised polyadic *pi*-calculus without levels was implemented in *Caml*.

6 Conclusions and Future Work

During this internship, we explored the possibility of proving termination for a type system combining subtyping and level assignment. We started from the work of [PS96] and later on discovered the type system of [IK00]. We have not studied throughly the constraints and benefits of using the latter type system with level based termination and that is something we would like to do in the future. Using the type system of [IK00] is mostly interesting for the type inference procedure.

We only have some ideas about the inference procedure for the type system of Section 4.1. We presented them in Section 4.5. As future work, we would like to find a complete inference procedure and prove its correctness. For the type system of Section 4.4 the inference procedure is also part of future work.

Moreover, the type system of Section 4.1 is a step towards a more complex one, suitable for proving termination of Erlang programs. This is an interesting continuation of the work presented in this document.

Acknowledgements. Romain Demangeon has provided insightful comments and suggestions on this work.

References

[CH11] I. Cristescu and D. Hirschkoff. Termination in a π-calculus with subtyping. http://perso.ens-lyon.fr/ioana.domnina.cristescu, 2011.

[Dem10] R. Demangeon. *Terminaison des systèmes concurrents.* PhD thesis, ENS Lyon, 2010.

[DHS09] R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Mobile Processes and Termination. In *Semantics and Algebraic Specification*, volume 5700 of *LNCS*, pages 250–273. Springer, 2009.

[DHS10] R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Termination in Impure Concurrent Languages. In *Proc. of CONCUR'10*, volume 6269 of *LNCS*, pages 328–342. Springer, 2010.

[DS06] Y. Deng and D. Sangiorgi. Ensuring termination by typability. *Inf. Comput.*, 204(7):1045–1082, 2006.

[IK00] A. Igarashi and N. Kobayashi. Type reconstruction for linear -calculus with i/o subtyping. *Inf. Comput.*, 161(1):1–44, 2000.

[KS10] N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5), 2010.

[Lab11] Ericsson Computer Science Laboratory. Erlang programming language website. http://www.erlang.org, 2011.

[PS96] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Math. Structures in Comput. Sci.*, 6(5):409–453, 1996.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.