Numéro d'ordre: 595 Numéroattribué par la bibliothèque: __ENSL595

Thèse

en vue d'obtenir le grade de

Docteur de l'Université de Lyon – École Normale Supérieure de Lyon
spécialité: Informatique
et Dottore di Ricerca dell'Università di Bologna

Laboratoire de l'Informatique du Parallélisme
Dipartimento di Scienze dell'Informazione

École doctorale de Mathématiques et Informatique Fondamentale

présentée et soutenue publiquement le 23/11/2010 par

Romain DEMANGEON

Terminaison des systèmes concurrents

Directeurs de thèse:	Daniel	HIRSCHKOFF
	Davide	SANGIORGI
Après avis de:	Delia	KESNER
	Nobuko	YOSHIDA

Devant la commission d'examen formée de:

Daniel	HIRSCHKOFF	(membre)
Delia	KESNER	(membre/rapporteur)
Pierre	LESCANNE	(membre/président)
Jean-Yves	MARION	(membre)
Davide	SANGIORGI	(membre)
Nobuko	YOSHIDA	(membre/rapporteur)

UNIVERSITÉ FRANCO ITALIENNE

Contents

Resumé en Français

Cette thèse propose une étude de la terminaison dans les langages concurrents. La terminaison est une propriété-clé pour les programmes. Cette propriété n'est pas seulement utile en *elle-même* (on veut pouvoir dire qu'un programme termine au bout d'un certain temps) mais elle est aussi une pré-condition à d'autres propriétés utiles: absence d'interblocage [KS10] ou correction de programmes.

Après avoir introduit le formalisme du π-calcul dans la Section 2, on commence par donner une nouvelle présentation de trois systèmes de types pour la terminaison en π-calcul existants, introduits dans [DS06], basés sur la décroissance d'une mesure bien fondée, appelée poids.

On présente ensuite dans la Section 3 les résultats proposés dans [DHS08], où l'expressivité de ces systèmes de types a été étendue de deux manières. Premièrement, on montre que le plus complexe des 3 systèmes de types sus-cités peut être amélioré en un système de types assurant la terminaison de structures de données inductives bien-fondées.

On aborde aussi la question de l'amélioration de l'expressivité d'un autre point de vue: il est en effet possible d'assurer la terminaison de systèmes créant dynamiquement des structures de données inductives en utilisant une analyse *hybride*. Ce système, dont l'inférence est polynomiale, accepte certains programmes *a priori* divergents, mais ces derniers sont exécutés selon une sémantique qui force la terminaison. Cette analyse permet, par exemple, de reconnaître comme terminants des programmes contenants des boucles dans du *code mort*.

On s'intéresse dans la Section 4 aux résultats présentés dans [DHS10a], concernants la terminaison pour les calculs concurrents d'ordres supérieurs dérivés du π-calcul où les messages communiqués sont du code de programme. On présente dans cette thèse différents systèmes de types assurant la terminaison de programmes dans HOpi_2, HOpi_ω, et PaPi, trois langages de ce type.

Dans la Section 5, on présente une étude de la complexité du problème de l'inférence pour les systèmes de types définis jusque ici (ces résultats sont présentés dans [DHKS07]). On prouve que si l'inférence des systèmes basés sur les comparaisons entre deux entiers est polynomiale, l'inférence des systèmes plus expressifs est NP-complète. Cette thèse propose en outre de nouveaux systèmes de type qui tout en ayant une expressivité comparable à ces derniers ont une inférence polynomiale.

On présente ensuite brièvement dans la Section 6 des méthodes sémantiques assurant la terminaison d'un sous-calcul "fonctionnel" du π-calcul (qui contient l'encodage du λ-calcul). Néanmoins, ces méthodes ne capturent qu'une partie confluente (donc non-concurrente) du π-calcul.

Enfin, dans la Section 7, on présente les résultats de [DHS10b]: on étudie la terminaison d'un π-calcul *impur* dans lequel on distingue les opérations fonctionnelles des opérations impératives. La terminaison des opérations fonctionnelles seules est prouvée dans [San06] en utilisant les relations logiques, tandis que la terminaison des opérations impératives peut être prouvée en utilisant un système basé sur le poids. Une nouvelle méthode de preuve est utilisée, basée sur l'élagage et la simulation, aboutissant à une contradiction avec le résultat de [San06].

Une dernière partie est consacrée à illustrer comment cette méthode a été utilisée avec succès pour prouver la terminaison d'un calcul *séquentiel* impur: le λ-calcul avec références. Une preuve de terminaison de ce calcul, basée uniquement sur la réalisabilité, est présente dans [Bou07]. Cette thèse propose ainsi une nouvelle méthode de preuve pour ce résultat.

Riassunto in italiano

Questa tesi studia la terminazione dei linguaggi concorrenti. La terminazione, oltre ad essere una proprietà utile di per sè , può servire a garantire l'assenza di deadlock [KS10] o la correzione dei programmi.

Dopo aver definito il pi-calcolo nella sezione 2, viene proposta una nuova presentazione dei sistemi di tipi per il pi-calcolo di Deng e Sangiorgi [DS06], che si basano sulla definizione di una misura ben fondata sui

processi.

Nella sezione 3 vengono esposti i risultati di [DHS08], che migliorano l'espressività dei sistemi di Deng e Sangiorgi. Si dimostra che il sistema più espressivo tra questi può essere ancora migliorato in modo da poter garantire la terminazione per dei processi che implementano delle strutture di dati ben fondate.

Un altro approccio viene esposto per migliorare l'espressività : si dimostra come dei processi che creano dinamicamente strutture di dati possono essere dimostrati terminanti usando un sistema ibrido. Questo sistema, per il quale il problema dell'inferenza è polinomiale, accetta certi programmi che sembrano a priori divergenti, organizzandone l'esecuzione secondo una semantica operazionale che garantisce la terminazione. Questa analisi permette in particolare di accettare dei programmi che comportano divergenze in certe parti che sono 'codice morto'.

Nella sezione 4 vengono esposti i risultati di [DHS10a] sui calcoli concorrenti di ordine superiore derivati dal pi-calcolo. Dei sistemi di tipi per la terminazione sono definiti, per tre calcoli di questo genere, HOpi_2, HOpi_ω, e PaPi.

Viene poi studiata la questione della complessità del problema dell'inferenza per i sistemi di tipi studiati fino a questo punto della tesi nella sezione 5 (questi risultati sono pubblicati in [DHKS07]). Si dimostra che l'inferenza è polinomiale per i sistemi per cui la tipabilità è basata su dei confronti tra due interi, mentre NP completa per gli altri sistemi, più espressivi. Vengono inoltre proposti dei sistemi la cui espressività è comparabile a questi ultimi, con un'inferenza che rimane polinomiale.

Nella sezione 6, si presentano dei metodi cosidetti semantici, che permettono di stabilire la terminazione per un sotto-calcolo "funzionale" del pi-calcolo (il sotto-calcolo contiene in particolare la traduzione del λ-calcolo in π-calcolo). Questi approcci permettono di tipare soltanto una parte confluente (e di conseguenza non concorrente) del π-calcolo.

Infine, nella sezione 7, si presentano i risultati di [DHS10b]: viene studiata la terminazione per un π-calcolo *impuro*, in cui si fà una distinzione tra operazioni funzionali e operazioni imperative. La terminazione della parte funzionale dimostrata in [San06] tramite le relazioni logiche, mentre la terminazione della parte imperativa si ottiene usando un sistema basato su un'assegnazione di un peso ai processi. Un metodo di prova nuovo viene usato in modo da dimostrare la correzione dell'analisi fatta dal tipaggio. Questo metodo si basa su una funzione di pruning e su un risultato di simulazione, in modo da derivare una contraddizione con il risultato di [San06].

L'ultima parte della tesi tratta della terminazione per un calcolo sequenziale, il λ-calcolo esteso con delle referenze. La prova di terminazione è costruita adattando l'approccio appena descritto.

Chapter 1

Introduction

In this section, we first introduce the main topic of this document through a simple example, then we present in details the notions of termination and concurrency and finally give an introduction to the contributions of this thesis.

1.1 A motivating example

As a starting point, we consider a very simple program, written in a message-passing pseudocode very loosely inspired from the well-known Web Services Description Language [CCMW01], describing an environment where several servers are organised into a list and requests to search values can be sent to them. We suppose that we are liable to define services (using the `service` keyword), which are able to receive requests (corresponding to the `input` definition in their code) and to answer back (using the `output` keyword). For both of these operations at least two logical fields have to be defined, what is contained in the message (using `message`) and what is the other party in the communication (using either `from` or `to`). We make explicit the use of standard operators such as conditional (using the keywords `if`, `then`, `else`).

```
<service name="Server_1" value="v_1">
     <input message="r" from="FromRequest">
     if r=v_1 then
          <output message="ack" to="FromRequest">
     else
          <output message="r" to="Server_2">
          <input message="x" from="FromAck">
          <output message="x" to="FromRequest">
</service>

<service name="Server_2" value=v_2>
     <input message="r" from="FromRequest">
     if r=v_2 then
          <output message="ack" to="FromRequest">
     else
          <output message="r" to="Server_3">
          <input message="x" from="FromAck">
          <output message="x" to="FromRequest">
</service>

...
```

```
<service name="Server_n" value="v_n">
     <input message=r from="FromRequest">
     if r=v_n then
          <output message="ack" to="FromRequest">
     else
          <output message="fail" to="FromRequest">
</service>
<service name="Client_1" value="r_1">
     <output message="r_1" to="Server_1">
     <input message="x" from="FromAnswer">
</service>

<service name="Client_2" value="r_2">
     <output message="r_2" to="Server_1">
     <input message="x" from="FromAnswer">
</service>
```

This example contains a bunch of servers $\texttt{Server}_\texttt{i}$, each one is associated to a value $\texttt{v}_\texttt{i}$. Requests containing an integer `r` can be sent to these servers. If the associated value and the requested one match, an acknowledgement `ack` is answered back (the sender of the request is stored in the variable `FromRequest`). If it is not the case, the request is propagated to the next server, if there is one, and an acknowledgement is expected. Conversely, when an acknowledgement is received, it is propagated back, if possible. If the end of the list is reached (the service $\texttt{Server}_\texttt{n}$) by a request, the signal `fail` is returned. Two client services are also defined, both sending a request on the initial service $\texttt{Server}_1$. One important point is that we want this program to be executed in a concurrent (or even distributed) setting, that is, we want the communications resulting from the requests of both clients to interleave.

During an execution of this program, messages are sent from one server to another one: requests can be propagated further in the list and acknowledgements are propagated back all the way to the clients. Yet, we guess that, as the server list is not cyclic, the execution of this program always terminates, no matter in which order the interleaving computations are performed.

In this document, we present methods for the analysis of termination of concurrent programs. A first idea to prove the termination of this simple example consists in stating that every time a request is sent from a server to another one, the index of the server being called increases, and every time an acknowledgement is sent, the index of the server being answered decreases. This allows us to state that no loop will arise, as the request will eventually either reach the last server, or reach a server containing the value searched for, and the acknowledgement will eventually reach the client, via the first server. Some of the proofs we will formalise in this document stem from the one we described above: making explicit a measure that decreases at each communication.

1.2 Termination

We briefly present here the notion of termination and a short state-of-the-art analysis of termination of programs.

Definition We say that a program *terminates* when every computation it can perform is finite (executed in a finite number of computation steps), or, in other words, that one cannot obtain an infinite computation by executing this program. Termination (sometimes called *strong normalisation* when considering programs validated by an underlying type system) is a key property in the domain of programming languages. Most of the programs written in the industrial world are meant to terminate in a small amount of time and the

termination property is required in order to obtain soundness of algorithms: indeed, it is not satisfying only to know that a program gives a correct answer to a problem, one has to be sure that this answer is returned in a finite amount of time.

Yet, termination is not only a property desirable in itself, it is also a prerequisite required to obtain other key properties in programming theory, such as fairness (see [Bou07]) or lock-freedom (see either [KS10], or "*reducing lock-freedom to termination*" in [GCPV09] or *the progress property* in [DCdY07]). In the first case, the termination of the execution of threads of computation ensures that the scheduler will eventually yield to each thread waiting to be executed, in the second case, termination of subprocedures ensures that no livelock situation appears for the whole system. In both cases, the termination of subprograms allows us to deduce that the whole system behaves as expected.

Moreover, in a world of distributed services, termination is a crucial property, as diverging behaviours may lead to *denial of service*. Indeed, when a request is sent on a network, the sender wants to be sure that it will receive an answer in a finite amount of time. Even if objects such as servers are by essence non-terminating (they are made to be permanently available), every computation induced by a single call to them is required to terminate, thus termination is often needed in order to ensure *responsiveness* (this property ensures that a given service will always be eventually available, see [AB08]).

Undecidability of termination The issue of termination for most of the "useful" languages (languages in which one is able to encode non-trivial systems such as Turing machines, rewriting systems and usual programming paradigms) is undecidable. Indeed, deciding the termination of a given Turing machine on a given argument is known as *the halting problem* [Tur36] and is considered, in the domain of the recursion theory, as the main undecidable problem, meaning that proving that another problem is undecidable often involves a reduction from the halting problem. This means that we cannot write any algorithm taking programs as arguments and deciding whether they terminate or not. The consequences of such a result are huge for one willing to design automated termination verifiers: this means that no analyser can fully capture termination and that every automated procedure checking whether a program terminates or not is not *complete* and bound to produce false positives, in other words, not to recognise as such some terminating programs.

As a consequence, an interesting task, for those willing to verify automatically programs written in interesting languages, is to design methods to ensure termination which are more and more expressive, i.e. that can recognise as terminating a greater number of programs. However, the gain in expressiveness does not come without a price, as the more expressive systems are often more technical and less efficient (when considering the number of elementary operations they perform in order to prove the termination of a program with respect to its size).

Abstract interpretation In this context, developing automated tools for termination, which are able to recognise terminating programs as such, becomes desirable. The automated verifier `Terminator` ([CPR$^+$07b]) developed by Cook, Podelski and Rybalchenko, has been used to ensure termination of complex libraries used by Microsoft operating systems, preventing serious bugs from arising. This tool is based on a method called *abstract interpretation* ([CC04]): to check termination, states of the system (seen as vectors of values representing the state of the memory, mapping each memory address to a value) are abstracted to sets of states (for instance, one of these sets can contain all states whose first value is greater or equal than some constant, e.g. all memory states mapping the address 0 to a value $\geq c$). Instead of considering reductions and reachability relations on the collection of all states, we study these relations on these abstracted sets. As a result we get a computable procedure which is able to derive statements about the behaviour of a given program such as "computation starting from this memory state may lead to the same state, thus the program is unsafe from the point of view of termination" (of course, this statement can be a false positive, that is, the program can be terminating but the abstract interpretation method is not precise enough to recognise it as such).

Termination through type systems A usual way to ensure termination (especially for functional languages) is the use of type systems: programs and resources (variables, addresses, values) are decorated with types. Typing rules are used inductively to construct typing derivations for some programs. The programs accepted by this procedure, that is, the programs for which there exists a typing derivation are said *typable*. The soundness of such a type system for termination is the property ensuring that each typable program is terminating.

In the case of the program defined above, we can use type systems to decorate the servers $\texttt{Server}_\texttt{i}$ with their indices (that is, to associate to $\texttt{Server}_\texttt{n}$ the natural number n) and derive, using typing rules, that all calls occurring inside the code of a server typed with integer n are done to servers typed with integers N, where $N > n$ and that all answers occurring inside the code of a server typed with integer n are sent to servers typed with integers N, where $N < n$ (we detail further how it can be done for this example). These typing informations allow us to state formally that some measure is decreasing when a call is performed. We will present below how we can obtain such a result. Notice that in this precise case, the use of a type system is quite similar to the abstract interpretation method described above, as our types are very simple (consisting in only integers). Yet, type systems allow us to be more precise in our analysis, for instance by describing behavioural properties.

Type systems are used to prove diverse properties: termination [GTL89], fairness [Bou07], security of communications [HVY00], and other behavioural properties [CYH09], ... One of their interesting features, compared to the abstract interpretation method, is the fact that typing derivations constructed in order to typecheck a given program P are inductively built from *typing rules*, following the syntax of P. Most of the type systems we will present in this document are *syntax-directed*, which means that only one typing rule can be chosen as the first rule to apply to a given program. This point makes type systems methods obviously desirable in practise, as an automated typechecker will be able construct such a derivation, if there exists one, by deconstructing the candidate program.

In this thesis, we will divide the methods we present to obtain termination into two parts: the methods inherited from rewriting theory, or measure-decreasing (weight-based) methods, and the methods inherited from proof theory (logical relations, realisability), or semantics-based methods.

Termination in rewriting theory Termination is well-explored in the domain of rewriting theory, both in the context of term-rewriting and string-rewriting. An interesting point is that termination of a term-rewriting system is ensured if and only if there exists a rewriting-ordering $>$ for this system, i.e. a way to compare all well-formed terms ensuring that if term t reduces to term t', then $t > t'$. Many authors have introduced practical methods for finding such orderings, such as *path orderings* (multiset path orderings or lexicographical path orderings) and *interpretations* (see [BN88] and [Ter03] for more details). In the latter case, a mapping function A maps each n-ary symbol of the vocabulary to an n-ary operation in a well-founded algebra, and the termination of a term rewriting system is obtain by proving that for each reduction rule $l \to r$, the interpretation of the left-hand side of the rule $\mathrm{A}(l)$ is an element of the algebra which is strictly greater than the interpretation of the right-hand side of the rule $\mathrm{A}(r)$. Of course, the presence of variables in l and r makes the task not trivial, as the latter property must hold for every substitution of variables by terms. If the algebra considered is the integers ($\mathbb{N}$) and if function symbols are interpreted by polynomials with integer coefficients, the whole procedure is called *polynomial interpretation* and can be efficiently implemented in an automated verifier [CL87]. Some simple terminating systems can require complicated interpretations in order to be proved as terminating. This is the case for the famous string-rewriting system $\{\mathrm{aa} \mapsto \mathrm{bc}, \mathrm{bb} \mapsto \mathrm{ac}, \mathrm{cc} \mapsto \mathrm{ab}\}$ which requires the use of several polynomials of high degree in order to be proved as terminating (see [HW06]).

One can notice that the method described here can be related to the abstract interpretation method we present above as programs are mapped to integers. This is no surprise, as both method are considered are based on interpretation. However, the latter allows one to perform a finer analysis, because the relations between abstract states can be described more precisely (for instance, one is able to state that a set of states is unreachable), whereas the relations obtained by the use of a sole integer are mono-dimensional comparisons. On the other side, the polynomial interpretation method is very efficient, as it only requires

ones to find suitable polynomials for each reduction rule. Indeed, interpretation methods in the rewriting theory are always defined on terms, which allows them to be efficiently computed.

In the following, we will adapt methods inspired from the rewriting theory to the concurrent setting, by mapping each concurrent program to an element of a well-founded algebra (natural numbers or multisets of natural numbers) and proving that computations let this well-founded measure decrease. Such a property ensures termination, as it contradicts the existence of an infinite reduction sequence of well-typed programs (of course, a *subject reduction* result has to hold, stating that every well-typed program always only reduces to well-typed programs).

Termination of functional languages On the other hand, termination for functional languages is also a well-studied domain. By the use of simple types, an expressive subset of the λ-calculus (a calculus considered as a standard model of functional computation) can be proved terminating (see [GTL89]). One can even refine this result to obtain the termination of the polymorphic λ-calculus (called *System F*). An interesting point is that the termination of System F implies the coherence of the second order arithmetic. By Gödel's incompleteness, this means that arithmetic methods (for instance, methods based on the decreasing of a well-founded measure of size smaller than the cardinal ϵ_0) cannot be used to ensure termination of this polymorphic calculus. Instead, one has to use semantics-based methods such as logical relations or realisability. This is not the case for the simply-typed λ-calculus, whose termination can be proved using arithmetical methods (see [Dav01]).

As a consequence, simple rewriting-based method, or interpretation into well-founded algebras smaller than ϵ_0 will sometimes not suffice to prove termination of expressive calculi. In these cases, more powerful tools have to be used.

In these methods, types for programs are constructed inductively and are interpreted by sets of (not necessarily typable) programs. The definition of interpretations ensures that each program in an interpretation is terminating. As a consequence, to obtain soundness, one has to prove that each well-typed program is contained in the interpretation of its type.

In the following, we try to use such semantics-based methods in order to obtain termination in the setting of concurrent programming. We even manage to obtain interesting results by combining these methods with the rewriting-based methods described above.

1.3 Concurrency theory

We present here termination in the concurrency theory, as well as existing related results.

In the past few years, concurrency has become more and more present in everyday computing. Indeed, the development of new distributed architectures: heterogeneous computing, embedded systems, GPUs and multicore platforms requires the development of more advanced tools able to verify the soundness of concurrent programs. Moreover, the quick development of web-based services tends to increase the demand of formal verification methods for distributed computing to answer questions such as "How can we ensure that a banking web service behaves as expected ?"

At first termination may not seem as fundamental in the concurrent setting as it is the sequential world. Indeed, when designing servers in a network, one may want them to run for an indefinite amount of time, or one may wish that they go back in the same initial state after each computation, considering such kind of "loop behavior" as sound. Yet, if termination of a whole system is not desirable (for instance there is no point considering the termination of the Internet), soundness of concurrent protocols strongly depends on the termination of subsystems: for instance, even if a bank server is running forever, we want every request sent by a client to terminate. In this context, we can safely say that termination is a sought-after property in a concurrent world.

Termination for the shared-memory model Concurrency is often studied in a shared-memory presentation: several threads (new threads can be created on the fly) which share a common memory are executed in parallel (that is, the executions of their instructions are interleaved, resulting in non-determinism). The

authors of [CPR+07b], cited above, explore the issue of termination in this shared-memory setting. To ensure termination one has to check that allowing two threads running in parallel to compete for the same resource does not lead to the creation of diverging systems (for instance, imagine that one thread keeps on incrementing a variable until it reaches 10, while the other one decrements the same variable until it reaches 0: both behaviours are terminating if taken independently but when put in parallel, a loop arises). The authors ensure termination in this setting by using abstract interpretations (see [CPR07a]). Yet, this thesis does not consider termination for threads sharing a common memory setting and focus on the message passing setting.

Termination for the message-passing model Another way to model concurrency is the message-passing presentation. In this setting, several programs are executed in parallel, their instructions including either emission or reception of messages on *channels*. When a program willing to output something on channel a is put in parallel with another one waiting to receive something on the same channel a, the two programs can perform a communication. Non-determinism is obtained as several senders on channel a can compete for the same receiver (and the other way around).

This document investigates the problem of termination in this message-passing setting, more precisely in the domain of mobile processes ([Mil89]). These processes communicate on channels, sending first-order values, addresses, names of other channels or even code of programs. One challenging issue is that the topology itself of a process can evolve after a few steps of reduction. For instance, new services can be created, old services can be updated, secret informations can be revealed to some agents, ... As a consequence, the calculi of mobile processes are expressive languages, allowing one to model a large range of concurrent paradigms. The downside to this expressiveness is that proving termination in this setting becomes more difficult. For instance, as said above, λ-calculus can be made terminating by enforcing a simple type discipline, this is not the case for mobile processes. The simple types discipline for channels exists in the formalism of the mobile processes, as presented in Section 2, but it does not ensure termination at all, as every diverging example presented in this document is simply-typable.

In this thesis, we will use as main languages to study termination the π-calculi ([Mil89],[SW01]), as they are commonly used to describe a large number of concurrent behaviours. In theses languages, names (or channels) are the first-class citizens. As written above, these channels can carry first-order values, other channels or code of processes. Computation is done by performing a communication when there exist two processes in parallel with matching actions: the first one trying to send something on a given channel and the second one waiting to receive something of this same channel. Syntaxes and semantics for these calculi are detailed in Section 2.

Termination in the π-calculus using logical relations This thesis considers two main results as starting points. We will present both of them briefly.

On one side, [San06] and [YBH04], which make use of the logical relations technique in order to prove as terminating small subsets of the π-calculus. Although this terminating subsets are not very expressive, seen from the point of view of the concurrency theory (they corresponds roughly to the encoding of the λ-calculus into the π-calculus, and, as a consequence, the processes in this subset are confluent), these are the first steps in the use of semantics-based techniques for termination of mobile processes. The framework of the termination proof follows the one for the λ-calculus: types are given to processes (corresponding, in [San06], to the types of the arguments and return values they offer), an interpretation of types into sets of terminating processes is defined, and, to conclude, the authors prove that every typable process belongs to the interpretation of its type. This technique is promising, as it boasts the advantages of its counterpart in the λ-calculus: it can be easily adapted for polymorphic languages, it relates computation and types with logics and proof theory, and it is quite elegant. Yet, the fact that the behaviours of the processes recognised as terminating cannot be considered as "truly concurrent" makes this result alone is not satisfying.

Yet, The simple program we presented at the beginning of this document can be typechecked using these results. Indeed, the services $\texttt{Server}_\texttt{i}$ are purely functional, in the sense that their behaviours do not change over time, and that they are always available (this can be related to *uniform receptiveness* [San99]).

Thus, using this method, and by giving to these services types corresponding to their "functional type", i.e. ensuring that they wait for an integer and answer a signal, we manage to typecheck them. Indeed, the crux of the result in [San06] is that a service shall not contain any recursive call to be typable (i.e. a service cannot call itself). This condition is enforced here, as each service $\texttt{Server}_\texttt{i}$ only calls the other service $\texttt{Server}_{\texttt{i}+1}$. Should we be willing to add more complex features to this example, such as a lock mechanism, this method would not be appliable, meaning that we have to find other ways to ensure termination in order to be able to recognise as terminating expressive examples.

Termination in the π-calculus using levels and weights On the other side, termination in [DS06] is enforced using a rewriting-based method: levels are assigned to services via a type system and the bodies of services are given weights relating them to the levels of the calls they can perform (if the code of a service contains calls to services of level up to n, its weight is n). The definition of a service is type-checked only if the level of the service is strictly greater than the weight of its body. As a result the global weight of a process decreases at each reduction step: at each communication, a server of level n is called and produces some computations interacting with servers whose levels are $< n$. Termination is enforced by stating that a measure is well-founded (we can count every outputs not inside the code of a service). If this technique allows us to prove the termination of a large range of processes modelling concrete systems (for instance, concurrent, inductive data structures), it fails to capture the termination of the encoding into the π-calculus of the simply-typed λ-calculus: it cannot deal with functional processes as easily as the previous method. Yet, as presented in Section 5, this method could be efficiently implemented in an automated procedure, as finding a suitable level assignment for names is not hard. Moreover, several refinements and extensions of this method can be developed in order to make it more expressive.

The example presented above can also be proved as terminating using such a method, we show here the different steps. The key is to separate the service calls originating from a request (the ones carrying integers) from the ones originating from acknowledgements and assigning levels according to this distinction. If the total number of servers is K, we give level $2.K - i$ to a request call to service $\texttt{Server}_\texttt{i}$ and level i to the output of an answer to service $\texttt{Server}_\texttt{i}$. Thus, if an output of level k is a request call (to $\texttt{Server}_{\texttt{k}-2.\texttt{K}}$), it can produce either a request to the following server (to $\texttt{Server}_{\texttt{k}-2.\texttt{K}+1}$), whose level is $k - 1$, or an answer output to the previous server (to $\texttt{Server}_{\texttt{k}-2.\texttt{K}-1}$), whose level is $k - 2.K - 1$; in both cases, the level of the call decreases. If an output of level k is an answer output (to $\texttt{Server}_\texttt{k}$), then it can only produce another answer output to the previous server (to $\texttt{Server}_{\texttt{k}-1}$), whose level is $k - 1$. As the levels of calls can only decrease at each reduction, the process is terminating.

We just proved that the weight-based methods allow us to recognise as terminating small examples in a very efficient manner. We will prove in the following that there exist also strong limitations to these methods.

1.4 Contributions of this thesis

Using the two methods presented above as a starting point, we investigate in this thesis how we can refine the weight-based methods and how we can obtain similar results in the setting of the higher-order π-calculus. Then we study the inference problem for the weight-based methods, that is, how they can be efficiently implemented. After briefly presenting the know results for semantical methods, we finish by proposing a method for putting together the advantages of the two kind of methods (i.e. semantical ones and weight-based ones) in order to build an expressive type system ensuring termination in *impure* languages.

Impure languages are settings in which a functional part and an imperative part can be distinguished inside programs, for instance, a π-calculus where functional definitions (as stated above, services which are always available and non-mutable) are made explicit or a λ-calculus with references (and operators for manipulating the memory).

We present in Section 2 the formalism we will work with in the remaining of this document: we define the concurrent languages (for both message-passing and process-passing calculi) we will use further, present the notion of termination in these settings and examples of diverging processes. We also briefly present

the simply-typed λ-calculus. It will be useful further, as we will study relations between termination in the λ-calculus and the π-calculus and use one of our termination techniques in a λ-calculus context.

1.4.1 Weight-based type systems for the message-passing languages

A simple system of levels and weights Section 3.1 studies the original weight-based type system for termination in the π-calculus found in [DS06]. We give a new presentation for this system and a new proof of its soundness. The advantage of this new proof is that it will be used as a skeleton for the termination proofs in the next sections. The proof goes as follows: after defining the type system, we first prove that it is stable by communication (the *Subject Reduction* proposition), that is, that a typable process always reduces to a process which is also typable. Then we define inductively a well-founded measure on processes. Then we prove that this measure decreases at each reduction step. We deduce the soundness of the type system by showing that the divergence of a typable process raises a contradiction with the well-foundedness of the measure.

This original type system has been briefly discussed above: we assign levels to channels on which the communications take place and give weights to processes by computing the maximum level of a call they can perform (for instance, if the body of a service make one call to a level-3 service and two calls to level-2 services, its weight will be 3). The crux of the termination is that when we define a service, we ensure that the weight of the body of the service is strictly smaller than the level of this service, i.e. than level of the call required in order to start the computation. As an immediate consequence, a service is not able to call itsef.

Let us present the π-calculus translation of $\texttt{Server}_\texttt{i}$:

$$!request_i(r, ans).\texttt{if } r = v_i \texttt{ then } \overline{ans}\langle \texttt{ok} \rangle \texttt{ else } (\nu\ ans_i)(\overline{request_{i+1}}\langle r, ans_i \rangle.ans_i(x).\overline{ans}\langle x \rangle)$$

We explain briefly the meaning of this presentation, formal details about the π-calculus syntax can be found in Section 2. The prefix $!request_i(r, ans)$ means that the service offered on the channel $request_i$ is persistent and that requests sent on this channel shall contain an integer r and a return channel ans. Then an equality test is performed between the value received r and the internal value of the server v_i. If they are equal, an acknowledgement $\overline{ans}\langle \texttt{ok} \rangle$ is sent on the return channel. If they are different, a request is propagated to the next server $\overline{request_{i+1}}\langle r, ans_i \rangle$. As multiple requests stemming from a large number of different clients can be sent in parallel, return channels associated to two different requests have to be made different. The operation $(\nu\ ans_i)$ is the creation of a new return channel, on which a signal is expected (the prefix $ans_i(x)$). When it is received, the signal is propagated back to the initial process using $\overline{ans}\langle x \rangle$. One can easily be convinced that the behaviour described here is the one of the service $\texttt{Server}_\texttt{i}$. As written above, termination of this process can be proved by giving level $2.K - i$ to the channel $request_i$ and level i to channel ans_i. Moreover, our type systems can ensure that the channel ans received on channel $request_i$ has the same type as the channel ans_{i-1} (found in the code of $\texttt{Server}_{i-1}$). Thus the weight of the process $\texttt{if } r = v_i \texttt{ then } \overline{ans}\langle \texttt{ok} \rangle \texttt{ else } (\nu ans_i)(\overline{request_{i+1}}\langle r, ans_i \rangle.ans_i(x).\overline{ans}\langle \texttt{ok} \rangle)$ is $2.K - i + 1$, the level of the channel $request_{i+1}$. As triggering this server requires a call on $request_i$, whose level is $2.K - i$, typability is ensured.

Refining the analysis to validate recursion The main idea we present above does not allow the typing of services containing recursive calls. In other words, a service which calls itself cannot be recognised as terminating, as the weight of the computation induced by triggering such a process is at least equal to the level of the call to this service. For instance, one is not able to type a program like this one:

```
<service name="Server" value="v">
     <input message="r" from="FromRequest">
     <input message="proceed" from="FromProc">
     if r=v then
          <output message="ack" to="FromRequest">
```

```
     else
          ...
          <output message="r" to="Server">
          ...
</service>
```

This service performs a computation similar to the one performed in the original example. However, an additional signal *proceed* has to be sent to the server in order to start the computation. Moreover, in the `else` branch, we add a recursive call: the server starts over by sending to itself the same request, but does not send the *proceed* signal. Such a modification can be useful if we want to update the server (for instance, by changing its internal value v), if the update is done when the server is treating the request, the updated version will start by computing the old request, thanks to the recursive call. One important point is that the server is not able to send to itself the `proceed` signal. As a consequence, termination is guaranteed: the `proceed` signal has to be sent from another process to the server in order to start over the computation: thus, no internal loop arises. Yet, this process is not typable using the previous analysis, as the recursive call will give to the body of the server the same weight as its level. The solution proposed in [DS06] to take into account such innocuous recursive calls consists in extending the analysis to the whole `input` sequence starting a service. By comparing the levels of both initial calls to the calls triggered by the computation they spawn, we prove that some quantity is decreasing.

In Section 3.2.1, we propose a new presentation for the type system of [DS06] based on this method. We give levels to channels, as above, but instead of making integer comparisons (of the form $n > k$, as in the previous system), we compare multisets of integers (operations on multisets are formally described in Section 2). For instance, the server described above in pseudocode can be written in the π-calculus as:

$$!request(r, ans).proceed.\texttt{if } r = v_i \texttt{ then } \overline{ans}\langle \texttt{ok} \rangle \texttt{ else } (\nu\ ans')(\overline{request}\langle r, ans' \rangle \mid \dots)$$

Here we compare the multiset of levels of the inputs (the two first prefixes) *request* and *proceed*, to the multiset of levels of the call in the body of the server, ans' and *request*. If we give level $1, 2, 3$ to, respectively, $ans', request, proceed$, then the multiset of levels of the inputs is $\{2, 3\}$ and the multiset of the calls is $\{2, 1\}$. A decreasing for the standard multiset ordering can be ensured, and termination is guaranteed. A soundness proof of this type system is presented in Section 3.2.1. It follows the lines of the soundness proof for the previous system, but involves more technicalities, as an auxilliary calculus is required. This proof differs from the one used in [DS06], which uses commutations of reductions, but follows more closely the general scheme common to all our proofs.

Refining further the analysis: partial orders The next refinment we present in order to obtain greater expressiveness is the introduction of a partial order in the analysis, bound to compare names that cannot be distinguished by types. Suppose that we add to the original example the ability to build new servers dynamically. We add to the web services language the keyword `spawn` which spawns a copy of a previously defined service.

```
<service name="Build">
     <input message_1="Server" message_2="Succ"
      message_3 ="v" from="Client">
     <service name="Server" value="v">
          <input message="r" from="FromRequest">
          if r=v then
               <output message="ack" to="FromRequest">
          else
               <output message="r" to="Succ">
               ...
     </service>
     <spawn service="Server">
```

```
</service>
```

Here `Build` is a service allowing dynamic creation of new services. A request to `Build` contains the name of the server being created, the name of its successor (we suppose we want to maintain a list structure among servers, as in the original example) and an internal value. After receiving these informations, `Build` uses them to build a server similar to the ones found in the original example. Termination is still guaranteed if there exists an ordering between names of servers such that the `Build` service is always called with an argument `server` higher in the ordering than the argument `succ`. Yet, the previous analysis fails here, as levels cannot be assigned dynamically (this is one limitation of the static analysis). Thus new services, created on-the-fly, are all considered as having the same level, corresponding to the one given by the analysis to the service `Server` and it is not possible, in this case, to obtain easily a termination proof by a decreasing measure. In [DS06], the authors show a way to bypass this constraint, by using a partial order between services of the same kind as a component of the type system. For instance, the previous program is typable if we ensure that each time the `Build` service is called, its argument received in the field $\texttt{message}_1$ is greater for this partial order than the one received in the field $\texttt{message}_2$.

Section 3.3 is dedicated to the development of a new presentation for this type system. When typing a server, we ensure that either the weight of the induced computation is strictly smaller than the levels of the sequence of calls required in order to trigger this computation, or that these two measures are equal, but a partial order decreases strictly between the two multisets of names. For instance, the `Build` service can be written in the π-calculus as:

$$!build(server, succ, v).!server(r).(\ldots \mid \overline{succ}\langle r\rangle \mid \ldots)$$

The type of *build* ensures that the name *server* is greater, for the partial order, than the name *succ*. Thus, in each process put in parallel with this service, we check that this constraint is satisfied. This prevents two calls $\overline{Build}\langle server_1, server_2, v_1\rangle$ and $\overline{Build}\langle server_2, server_1, v_2\rangle$ from building a loop.

Proving soundness of this system leads to more technicalities, as we have to deal with the interactions between the use of a partial order and the dynamic creation of services (in the π-calculus, this boils down to controlling the use of the restriction operator ν). Indeed, one shall not be able to create an infinite number of servers, each one smaller for the partial order than the previous one. Details are given in Section 3.3 and the soundness proof follows the usual framework.

Allowing the weight to grow In Section 3.4, we go further by allowing the weight to grow in the comparisons between the input sequence and the triggered calls we get by typing the definition of a server. This allows us to accept terminating processes modelling inductive data structures. For instance, consider this service:

```
<service name="Build">
     <input message_1="Server" message_2="Left"
      message_3="Right" message_4 ="v" from="Client">
     <service name="Server" value="v">
          <input message="r" from="FromRequest">
          if r=v then
               <output message="ack" to="FromRequest">
          else
               <output message="r" to="Left">
               <output message="r" to="Right">
               ...
     </service>
     <spawn service="Server">
</service>
```

This `Build` service allows the construction of binary-tree-like structures. If we can relate the names *server*, $left$ and *right* by a partial order, the former method of [DS06] cannot be applied here. In [DHS08], we propose an analysis that, under some conditions, allows us to type such processes, where the growth of the weight is compensated by a partial order. Details can be found in Section 3.4.

Combining dynamic and static analyses In Section 3.5, we explore another direction. We design a weight-based analysis similar to the one of Section 3.1 with looser constraints: levels are assigned to names, but now the weight of the computation a service performs has to be less *or equal* than its level. Of course, this allows the typing of diverging processes. The point is that, in this setting, we control the execution of the processes: every typable service is annotated with some *loop informations* which make explicit, at run-time, statements of the form "Service `A` has already called service `B`". If an actual loop is detected among the calls, the execution is aborted. This method allows us to study, for instance, programs containing a diverging routine which is never called during a normal execution.

We prove the soundness of our system, that is, that no processes with annotation diverges: any service either terminates or raise a failure in a finite amount of time.

1.4.2 Weight-based type systems for process-passing languages

Divergence in the process-passing setting Section 4 is dedicated to the study of weight-based termination in the setting of process-passing calculi. More precisely, we show that it is possible to apply the weight-based techniques made explicit in the previous sections to obtain termination analyses in the higher-order π-calculi. The task is not trivial, as the notion of persistence (what we present in the form of the "service definition") is not required in order to construct higher-order diverging processes.

Here is a common example on diverging behaviour in HOpi_2 (syntax and semantics for this calculus, as well as details on this example can be found in Section 2).

$$Q_0 = P_0 \mid \overline{a}\langle P_0\rangle, \text{ where } P_0 = a(X).(X \mid \overline{a}\langle X\rangle)$$

When receiving the source code of a service X on a, P_0 executes X in parallel with a request on a containing the code of X. Here, the service P_0 is executed in parallel with a request on a containing the code of P_0, giving birth to a loop.

Using weight-based methods in the process-passing setting In [DHS10a], we present a method, inspired by the previous analyses, to ensure termination in higher-order concurrent languages. The main idea is to rule out programs like the one above by preventing a service receiving code to output code to a service with the same name (this can be seen as another instance of recursive output). This can be done using levels and checking domination constraints, as in the name-passing case.

We study in Section 4.1 how this method can be applied to a standard process-passing calculus HOpi_2. As there exists an encoding from HOpi_2 into the π-calculus, we compare the expressiveness of this method with the one of the method obtained by encoding a process into the π-calculus and checking whether it is typable or not for the system of Section 3.1.

Then we apply this analysis to more complex higher-order languages: we study termination through weight-based techniques for both HOpi_ω (in Sections 4.2 and 4.2), which is a concurrent language in which values communicated are functional values, and PaPi, which is a calculus featuring both process-passing and message-passing operations. For HOpi_ω, we show in Section 4.2 that refinements developed for the first-order setting can also be applied here, and we show how to type a non-trivial example using this technique.

1.4.3 The issue of inference

As said above, one advantage of the weight-based methods is that they can be easily implemented by automated verifiers: one has to assign integers to variables and check that integer comparisons hold. One

can wonder how efficiently these analyses can be performed. We study in Section 5 the problem of inference for the type systems developed in the previous sections, presenting results from [DHKS07].

First we explain that the simplicity of the level constraints spawned during the analysis of a process allows the inference of the original type system (the one of Section 3.1) to be performed in a time polynomial in the size of the process. However, we prove further that the inference problem for the more expressive systems of Sections 3.2.1 and 3.3 is NP-complete. This is done by performing a reduction from the well-known 3SAT problem. It means that no known method to find a sound level assignment, if there exists one, is better than trying all possible solutions.

As a type system whose inference is NP-complete can be considered inefficient, for one willing to implement it, we develop in Section 5 two new type systems, whose expressiveness is similar to the ones of the systems of Sections 3.2.1 and 3.3, but whose inferences are polynomial.

1.4.4 An hybrid proof method for termination of impure languages

Comparison with type systems for termination in the λ-calculus In Section 6.1, we show the limits of the weight-based methods by proving that the standard encoding of the simply-typed λ-calculus (a well-known terminating functional calculus) cannot be recognised as terminating by such systems. Indeed, the weight-based type systems equate types of names which are used the same way, i.e. they assign to the same level two different names, because these two names happen to be arguments of a same third name, in two different parts of the process. As a consequence, when trying to type the encoding of the simply-typed λ-calculus into the π-calculus, different names created by the encoding are given the same type, although they correspond to different objects in the term we try to encode, leading to unsatifiable constraints. Actually, a counter-example can be constructed: a λ-calculus term which is simply-typable but whose encoding cannot be typed using a weight-based method. This results encourage us to study other methods of termination for process calculi, such as semantical methods.

Semantics-based methods We present briefly in Section 6 a semantics-based method ensuring termination in the π-calculus. We briefly recall the results found in [San06], where a functional subset of the π-calculus is proved terminating using logical relations. This subset corresponds roughly to the encoding of the λ-calculus into π.

Merging the two approaches In Section 7, we put together the two approaches (logical relations and weight-based methods) and design a method for proving termination in impure languages, that is, in languages which feature both functional and imperative components.

The main idea is that termination of the functional parts of a program can be proved using logical relations and termination of the imperative parts using weights. Of course, things are not as simple as the mere union of two existing methods, as the two components can collaborate inside a program in order to create loops. Thus, we extend the level-based constraints used for the imperative part to the functional part, but we allow them to be looser: functional services can call other services (either functional or imperative) of the same level but they cannot call services having strictly higher levels. This constraint is not present to ensure a decreasing in functional computations, but only to make sure that the functional computations does not hinder the control put on the imperative part.

We use a new proof technique, based on pruning and simulation: we construct a pruning, an operation mapping impure programs to functional ones, by removing all their imperative parts. Then, we prove that the pruned process is diverging if the initial process is diverging. As a consequence, the existence of a diverging impure process raises a contradiction with the proof of termination of the functional processes.

In an impure π-calculus We first apply this new technique, in Section 7.1, to an *impure* π-calculus. This setting can be seen as the standard π-calculus in which we distinguish some names as functional. By extending the weight-system (as hinted above, in a looser way) to these functional names, we are able to prove termination of complex services (see Section 7.1.5).

In a λ-calculus with stores In Section 7.2, we apply the same technique to an impure λ-calculus, called $\lambda_{\texttt{ref}}$, which is a call-by-value simply-typed λ-calculus with stores. The syntax of this calculus contains addresses and imperative operations: dynamic creation of new addresses in the memory, assignment of a value to an address and reading of an address. Although this calculus is different from the previous one in its structure and its semantics (for instance, it is sequential and deterministic), we are able to use a similar proof technique using pruning and simulation.

Chapter 2

Preliminary results

2.1 Preliminary results and notations

In this section, we present some basic notions and notations that will be used all along the thesis.

2.1.1 n-uples and substitutions

When no other assumption is made, $<$ is the standard, well-founded, ordering over integers.

Uples and sets The notation $\widetilde{a}$ usually stands for a sequence $(a_i)_{1\leq i\leq n}$ for some elements a_i's and some integer n (possibly 0).

Substitutions In the following we avoid the use of term variables in our terms as we do not want to distinguish variable names from names in the π-calculi, unless stated otherwise. Therefore, in the following, for the different formal languages we will use (π-calculi, λ-calculi), when considering terms (seen as abstract terms constructed by a free algebra, as usual in rewriting theory), we use $\{t/x\}$ (when t is an abstract term and x a constant, i.e. a 0-ary symbol) to stand for the capture-avoiding substitution of x by t defined by the following , where f is a symbol and c a constant:

$$f(t_1,\ldots,t_n)\{v/x\} = f(t_1\{v/x\},\ldots,t_n\{v/x\}) \qquad \begin{array}{ll} c\{t/x\} = c & \text{if } c \neq x \\ c\{t/x\} = t & \text{otherwise} \end{array}$$

2.1.2 Multisets

We will often use multisets of objects (with finite support), using the same notation $\{\}$ as the one for sets. When manipulating multisets of integers, we will sometimes use a vector notation: $[1;2;0;2]$ is $\{1,1,3,3,4\}$ (notice that we enumerate the values from the highest to the lowest). We denote multiset sum with $\uplus$ as in $\{1,2,3\}\uplus\{1,2,2\} = \{1,1,2,2,2,3\}$. Multiset inclusion is straightforwardly defined by $M_1 \subsetneq M_2$ if there exists $N \neq \emptyset$ s.t. $M_2 = M_1 \uplus N$. Multiset difference $M_1 - M_2$ is denoted by $-$ as in $\{3,2,2,1\} - \{3,2\} = \{2,1\}$ and can be seen as a shortcut for writing $\exists M_3, M_1 = M_2 \uplus M_3$ (this also means, of course, that $-$ is not defined for every multiset pair). Multiset extension $\leq_{\text{mul}}$ of any ordering $<$ is defined as $M_1 \leq_{\text{mul}} M_2$ if, N being the maximal multiset s.t. $M_1 = N \uplus N_1$, $M_2 = N \uplus N_2$, for all $e_1 \in N_1$, there exists e_2 in N_2, $e_1 < e_2$. The strict ordering extension $<_{\text{mul}}$ is defined by $M_1 <_{\text{mul}} M_2$ iff $M_1 \leq_{\text{mul}} M_2$ and $M_1 \neq M_2$.

For instance, we have $\{1,2,2,3\} <_{\text{mul}} \{1,2,4\}$, as $\{1,2,2,3\} = \{1,2\} \uplus \{2,3\}$, $\{1,2,4\} = \{1,2\} \uplus \{4\}$, $2 < 4$, $3 < 4$.

Immediately we get:

Proposition 2.1.1 (Multiset inclusion and ordering)
If $M_1 \subsetneq M_2$, then $M_1 <_{\text{mul}} M_2$.

We extend easily the standard notion of maximum element noted max for multiset of integers $\neq \emptyset$. As we restrict ourselves to multisets with finite supports, if $<^+$ is the standard ordering on $\mathbb{N} \cup -\infty$, $M_1 <_{\text{mul}} M_2$ if there exists N s.t. $M_1 = N \uplus N_1$, $M_2 = N \uplus N_2$, and $\max(N_1) <^+ \max(N_2)$ (by setting $\max(\emptyset) = -\infty$). This allows to state the following property:

Proposition 2.1.2 (Domination of integer multisets)
If $<$ is the standard ordering for $\mathbb{N}$ then for any non-empty multiset M, $\max(M) \leq n$ iff $M <_{\text{mul}} \{n+1\}$.

We will often use the following theorem to prove the soundness of weight-based type systems:

Theorem 2.1.3 (Well-foundedness of multiset ordering)
If $<$ is well-founded on E, then $<_{\text{mul}}$ is well-founded on multisets with finite support of elements of E.

As a corollary, $<_{\text{mul}}$ defines a well-founded order on multisets of integers.

2.1.3 Lexicographical ordering

We define the lexicographical composition of several orderings $(<_i)_{1 \leq i \leq n}$ by: $(e_1, \ldots, e_n) <_{lex(<_1,\ldots,<_n)} (e'_1, \ldots, e'_n)$ if there exists $1 \leq i \leq n$ s.t. $\forall j < i, e_j = e'_j$ and $e_i <_i e'_i$.

Theorem 2.1.4 (Well-foundedness of lexicographical ordering)
If the $(<_i)_{1 \leq i \leq n}$ are well-founded on $(E_i)_{1 \leq i \leq n}$, then $<_{lex(<_1,\ldots,<_i)}$ is well-founded on $E_1 \times \cdots \times E_n$.

To prove the soundness of our type systems, we will often use, in the following, multiset measures on processes that decreases along reductions. Thus we will often refer to Theorems 2.1.3 and 2.1.4.

2.2 A π-calculus

Most of the results from this thesis are presented in the name-passing, concurrent formalism of the π-calculus [SW01], or in one of its variants. We suppose the existence of an infinite set of names. We use roman letters $a, b, c, v, x, y, \ldots$ to denote names. The grammar of the monadic π-calculus can be found in Figure 2.1: $\mathbf{0}$ is the *inactive process*, $\mid$ the *parallel composition* of two processes, $a(x)$ and $\overline{a}\langle v \rangle$ are called, respectively, *input* and *output prefixes* or *actions*. $!a(x).P$ is a *replicated input prefixes*, (νa) is the *restriction* of name a. The name a is the *subject* of the prefixes $\overline{a}\langle v \rangle$, $a(x)$ and $!a(x)$. We will use $(\nu \widetilde{a})$ (which should be, according to what we wrote earlier $(\nu a_1, a_2, \ldots, a_n)$) to denote $(\nu a_1)(\nu a_2) \ldots (\nu a_n)$.

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

Figure 2.1: Grammar for π-calculus terms

Remark 2.2.1 *The grammar we give here does not contain τ actions, as they are irrelevant from the point of view of termination. For the same reason, we do not include non-deterministic choice $(+)$ in the grammar. However, most of the results presented in this thesis can be easily adapted to a π-calculus containing the $+$ operator.*

We define the set of free names of a process:

Definition 2.2.2 (Free names) *The set of free names of a process P is inductively defined by:*

$$\text{fn}(\mathbf{0}) = \emptyset \qquad \text{fn}(\overline{a}\langle v \rangle.P) = \text{fn}(P) \cup \{a, v\} \qquad \text{fn}(a(x).P) = \text{fn}(!a(x).P) = (\text{fn}(P) - \{x\}) \cup \{a\}$$

$$\text{fn}((\nu a)\ P) = \text{fn}(P) - \{a\} \qquad \text{fn}(P_1 \mid P_2) = \text{fn}(P_1) \cup \text{fn}(P_2)$$

The restriction $(\nu a)\ P$ and the input prefixes $a(x).P$ and $!a(x).P$ bind respectively the names a and x in P.

We suppose the processes we write in the whole document abide a Barendregt Convention; that is, all bound names are pairwise distinct and are distinct from free names.

Structural congruence is defined by the axioms in Figure 2.2 and the closure by prefixes, parallel composition and restriction.

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

$$\frac{}{(\nu a)(\nu b)\ P \equiv (\nu b)(\nu a)\ P} \qquad \frac{}{(\nu a)\ \mathbf{0} \equiv \mathbf{0}} \qquad \frac{a \notin \mathrm{fn}(Q)}{(\nu a)\ (P \mid Q) \equiv ((\nu a)\ P) \mid Q}$$

Figure 2.2: Structural congruence axioms for π-calculus terms

As one can notice, only processes guarded by an input can be replicated. Moreover, no structural rule $!P \equiv (!P \mid P)$ is presented, we use a dedicated semantics rule for replicated inputs instead.

We will use in the following a presentation *à la Wright-Felleisen* (i.e. context-based) for the operational semantics. Evaluation can take place under restriction and in parallel of some spectator processes, but not under prefixes.

Definition 2.2.3 (Evaluation contexts)
Evaluation contexts are defined by:

$$\mathbf{E} ::= [\,] \mid (\nu a)\ \mathbf{E} \mid \mathbf{E} \mid P$$

where $[\,]$ *is a special symbol called* the hole. *We write* $\mathbf{E}[P]$ *for the process obtained by syntactically replacing the hole in* $\mathbf{E}$ *by the process* P.

Thus semantics is defined by the three rules of Figure 2.3.

$$(\mathbf{com})\frac{}{\mathbf{E}[a(x).P_1 \mid \overline{a}\langle v\rangle.P_2] \to \mathbf{E}[P_1\{v/x\} \mid P_2]}$$

$$(\mathbf{trig})\frac{}{\mathbf{E}[!a(x).P_1 \mid \overline{a}\langle v\rangle.P_2] \to \mathbf{E}[!a(x).P_1 \mid P_1\{v/x\} \mid P_2]} \qquad (\mathbf{cong})\frac{P \equiv Q \qquad Q \to Q' \qquad Q' \equiv P'}{P \to P'}$$

Figure 2.3: Semantics rules for π-calculus reduction

We write $P \not\to$ to mean that P cannot reduce further, that is, we are unable to derive $P \to P'$ for any P'.

Here are some examples of reduction to illustrate this semantics.

1. $\overline{a}\langle b\rangle.\mathbf{0} \mid a(y).\overline{y}\langle v\rangle.\mathbf{0} \mid b(z).\mathbf{0} \to \mathbf{0} \mid \overline{b}\langle v\rangle.\mathbf{0} \mid b(z).\mathbf{0} \to \mathbf{0} \mid \mathbf{0} \mid \mathbf{0} \not\to$
2. $a(x).((\nu c)\ \overline{b}\langle c\rangle).\mathbf{0} \mid \overline{a}\langle v\rangle.\mathbf{0} \mid b(z).\mathbf{0} \to (\nu c)\ \overline{b}\langle c\rangle.\mathbf{0} \mid \mathbf{0} \mid b(z).\mathbf{0} \to (\nu c)\ (\mathbf{0} \mid \mathbf{0} \mid \mathbf{0}) \not\to$
3. $\overline{a}\langle b\rangle.\mathbf{0} \mid \overline{a}\langle c\rangle.\mathbf{0} \mid !a(y).\overline{y}\langle v\rangle.\mathbf{0} \to \mathbf{0} \mid \overline{b}\langle v\rangle.\mathbf{0} \mid \overline{a}\langle c\rangle.\mathbf{0} \mid !a(y).\overline{y}\langle v\rangle.\mathbf{0} \to \mathbf{0} \mid \overline{b}\langle v\rangle.\mathbf{0} \mid \mathbf{0} \mid \overline{c}\langle v\rangle.\mathbf{0} \mid !a(y).\overline{y}\langle v\rangle.\mathbf{0} \not\to$

(notice that in the second reduction of 2., we use rule (**Cong**) to extend the scope of (νc))

In the remaining of this thesis, we will omit the trailing occurrences of $\mathbf{0}$ and write, for instance, $\overline{a}\langle b\rangle \mid a(y).\overline{y}\langle v\rangle \mid b(z)$ instead of $\overline{a}\langle b\rangle.\mathbf{0} \mid a(y).\overline{y}\langle v\rangle.\mathbf{0} \mid b(z).\mathbf{0}$.

2.2.1 Simple types

In this section and the following ones, we will present type systems which associate types to names. Typing is given *à la Church*; that is, every name is given a type *a priori* and a typing judgement will correspond to a unique mapping from names to types. Typings contexts Γ are considered as oracles associating each name (bound or free) to a unique type. We suppose that there exist infinitely many names of each type.

Moreover, the concurrent languages we consider are always simply-typed. This means that we can associate to each name a type constructed with the following grammar:

$$T ::= \mathbb{1} \mid \sharp(T)$$

where $\mathbb{1}$ is a special type containing a unique name $\bigstar$. In the remaining, actions on names of type $\sharp(\mathbb{1})$ will be denoted as CCS actions, as in $!a.\overline{b} \mid \overline{a}$.

Typing judgements are of the form $\Gamma \vdash^{\pi-ST} P$, meaning that P can be simply-typed with the typing context Γ.

We present here the typing rules for simple-types, which can be straightforwardly adapted to accommodate polyadic name-passing (see below):

$$(\mathbf{Nil})\frac{}{\Gamma \vdash^{\pi-ST} \mathbf{0}} \qquad (\mathbf{Par})\frac{\Gamma \vdash^{\pi-ST} P_1 \mid \Gamma \vdash^{\pi-ST} P_2}{\Gamma \vdash^{\pi-ST} P_1 \mid P_2} \qquad (\mathbf{Res})\frac{\Gamma \vdash^{\pi-ST} P \qquad \Gamma(a) = \sharp(T)}{\Gamma \vdash^{\pi-ST} (\nu a)\ P}$$

$$(\mathbf{In})\frac{\Gamma \vdash^{\pi-ST} P \qquad \Gamma(a) = \sharp(T) \qquad \Gamma(x) = T}{\Gamma \vdash^{\pi-ST} a(x).P} \qquad (\mathbf{Rep})\frac{\Gamma \vdash^{\pi-ST} P \qquad \Gamma(a) = \sharp(T) \qquad \Gamma(x) = T}{\Gamma \vdash^{\pi-ST} !a(x).P}$$

$$(\mathbf{Out})\frac{\Gamma \vdash^{\pi-ST} P \qquad \Gamma(a) = \sharp(T) \qquad \Gamma(v) = T}{\Gamma \vdash^{\pi-ST} \overline{a}\langle v\rangle.P}$$

In the following, we will use $\lambda_{\mathbf{ST}}$ to denote the simply-typed λ-calculus.

2.2.2 Polyadic π-calculus

Although, for the sake of simplicity, we will often use the monadic π-calculus to present our results, all these results still hold when considering the polyadic π-calculus. However some results, especially the ones involving partial orders (see section 3.3), make sense only with polyadic channels. Notice that there exists a fully-abstract encoding of the simply-typed polyadic π-calculus into the simply-typed monadic π-calculus (see [Yos96]).

When considering polyadic actions $a(\widetilde{x}), \overline{a}\langle\widetilde{v}\rangle, !a(\widetilde{x})$, simple types of names are given by:

$$T ::= \mathbb{1} \mid \sharp(\widetilde{T})$$

As a consequence of simple-types discipline, the arity of a channel is fixed once for all by the typing context.

The operational semantics for the polyadic π-calculus is straightforward, for instance, (**com**) becomes:

$$(\mathbf{com})\frac{}{\mathbf{E}[a(\widetilde{x}).P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2] \to \mathbf{E}[P_1\{\widetilde{v}/\widetilde{x}\} \mid P_2]}$$

where $\{\widetilde{v}/\widetilde{x}\}$ stands for $\{v_1/x_1\}\ldots\{v_m/x_m\}$. As the v_i's and x_i's are names, and, thanks to the Barendregt Convention, the x_i's are pairwise distinct, $\{\widetilde{v}/\widetilde{x}\}$ is well-defined. Notice that, for $\{\widetilde{v}/\widetilde{x}\}$ to be defined, $\widetilde{x}$ and $\widetilde{v}$ must have the same length, which is ensured by the type system (the arity of a).

2.2.3 Divergence

This document presents several ways of ensuring that a process terminates. The notion of termination is formally defined as follows:

Definition 2.2.4 (Divergence and Termination)
A π-calculus process P diverges (is divergent) *if there exists an infinite sequence $(P_i)_{i\in\mathbb{N}}$ s.t. $P_0 = P$ and for each $i \in \mathbb{N}$, $P_i \to P_{i+1}$.*
A processes P terminates (is terminating) *if it does not diverge.*

We say that a set of processes (or a language, or a calculi) is terminating if all its elements are.

Notice that if we remove the replicated input $!a(x)$ form the syntax, the π-calculus is trivially terminating. Indeed, at each reduction step, the global number of prefixes strictly decreases (one output and one input are consumed) and no new prefix is created.

Yet, the introduction of the replication operator leads to diverging behaviours. Consider $D_1 = !a.\overline{a} \mid \overline{a}$. It is easy to see that $D_1 \to D_1$; the subprocess $!a.\overline{a}$ creates a loop, and each time the replication is triggered, that is, each time an output on a is consumed, an output on a is released. We aim at detecting such potentially diverging behaviours.

Notice that preventing a name a from appearing in the continuation of a replicated input on a is not enough to enforce termination; considering $D_2 = !a.\overline{b} \mid !b.\overline{a} \mid \overline{a}$, we notice that $D_2 \to\!\!\to D_2$.

The setting of name-passing adds more difficulties to the detection of diverging behaviours. For instance, in $D_3 = c(x).!a.\overline{x} \mid \overline{a} \mid \overline{c}\langle a\rangle$, one has to foresee that the name x will be instantiated by a, and that $D_3 \to D_1$.

The problem of knowing, given a π-calculus process P, if it terminates is not decidable and this is not surprising, considering the expressive power of the π-calculus. We will state it later, the π-calculus contains the λ-calculus and the undecidability of the termination problem for the latter calculus is well-known. The purpose of the following sections is presenting formal methods of static analysis recognising terminating processes, which aim at being decidable. Of course, as a trade-off, some terminating processes are not recognised as such. Thus, the soundness of a type system for termination, can be stated as "every typable process terminates".

2.3 Higher-order π-calculi

We will show in Section 4.1 how the works on termination for the π-calculus can be applied to several higher-order (process passing) calculi. We present here several such process-passing calculi (a more complete presentation is given in [SW01] and [San92]) derived from the π-calculus.

2.3.1 HOpi$_2$

HOpi$_2$ is the simpler process-passing language we will consider, as it contains no replicated prefixes and as it is process-passing, i.e. the messages sent and received by names are processes. In addition to prefixes, parallel composition, restriction and the inactive process, we introduce *process variables*, ranged over X:

$$P ::= \mathbf{0} \mid (P \mid P) \mid \overline{a}\langle P\rangle.P \mid a(X).P \mid X \mid (\boldsymbol{\nu} a)\ P$$

Structural congruence is defined somehow the same way as the one for the π-calculus (Figure 2.2).

The evaluation contexts are defined by the following grammar:

$$\mathbf{E} ::= [\,] \mid (\boldsymbol{\nu} a)\ \mathbf{E} \mid (\mathbf{E} \mid P)$$

Notice that evaluation cannot be performed in message position, that is, if $P \to P'$, $\overline{a}\langle P\rangle$ cannot reduce to $\overline{a}\langle P'\rangle$, in other words, messages can be viewed as containing the code of programs, the instantiation of top-level process variables as an execution of these programs.

The operational semantics is very close to the one of the π-calculus, where process-passing replaces message-passing:

$$(\mathbf{HOcom})\frac{}{\mathbf{E}[\overline{a}\langle Q\rangle.P_1 \mid a(X).P_2] \to \mathbf{E}[P_1 \mid P_2\{Q/X\}]} \qquad (\mathbf{HOcong})\frac{Q \equiv P \qquad P \to P' \qquad P' \equiv Q'}{Q \to Q'}$$

We wrote above that the replicated input prefix $!a(x)$ is required in the π-calculus in order to obtain divergent behaviour. This is not the case in HOpi_2, because the higher-order component alone allows us to write such diverging processes:

$$Q_0 = P_0 \mid \overline{a}\langle P_0\rangle, \qquad \text{where} \qquad P_0 = a(X).(X \mid \overline{a}\langle X\rangle)$$

Clearly, $Q_0 \to (X \mid \overline{a}\langle X\rangle)\{P_0/X\} = Q_0$.

2.3.2 HOpi_ω

We present hereHOpi_ω; a more expressive higher-order concurrent language. In this setting, messages exchanged do not contain processes directly, but functions whose codomains is the set of processes (we use $\diamond$ to denote the type of a process, thus values communicated have type $\ldots \to \diamond$). As a consequence, HOpi_ω contains HOpi_2, as a process can be seen as a function of type $\mathbb{1} \to \diamond$.

When studying HOpi_ω, we use the term *value* to denote either $\mathbb{1}$ or function from values to processes. Output and input prefixes carry values and we introduce a new constructor in the syntax: $v\lfloor w\rfloor$ which is the function application of the value v to the value w. We impose this application to be well-typed (in other words, v must have type $T \to \diamond$ and w type T).

The following grammar presents how processes and values are inductively defined:

$$P ::= \mathbf{0} \mid (P \mid P) \mid \overline{a}\langle v\rangle.P \mid v\lfloor v\rfloor \mid a(x).P \mid (\boldsymbol{\nu} a)P \qquad\qquad v ::= x \;\big|\; \star \;\big|\; x \mapsto P$$

Structural congruence is defined in HOpi_ω as in HOpi_2. Evaluation contexts are given by the following grammar:

$$\mathbf{E} ::= [\,] \mid (\boldsymbol{\nu} a)\ \mathbf{E} \mid (\mathbf{E} \mid P)$$

Again, we notice that evaluations, can only be performed under a restriction or in parallel of some other processes. That is, evaluation cannot happen in the body of a function (if $P \to P'$, then the process $(x \mapsto P)\lfloor w\rfloor$ cannot reduce to $(x \mapsto P')\lfloor w\rfloor$).

Besides the (**Homcong**) rule, there is two ways of reducing a process, either a communication between two matching prefixes is performed, or an function is applied to a value:

$$(\mathbf{HomCom})\frac{}{\mathbf{E}[\overline{a}\langle v\rangle.Q_1 \mid a(x).Q_2] \to \mathbf{E}[Q_1 \mid Q_2\{v/x\}]} \qquad (\mathbf{HomBeta})\frac{}{\mathbf{E}[(x \mapsto P)\lfloor v\rfloor] \to \mathbf{E}[P\{v/x\}]}$$

$$(\mathbf{Homcong})\frac{Q \equiv P \qquad P \to P' \qquad P' \equiv Q'}{Q \to Q'}$$

Notice that, as functions are all of type $\ldots \to \diamond$, there is no notion of strategy, i.e. we cannot choose to reduce inside the arguments, as reductions are defined for process only: facing $(x \mapsto \mathbf{0})\lfloor(y \mapsto \mathbf{0})\lfloor\bigstar\rfloor\rfloor$, we can only performed the "outermost" application.

As we wrote it above, HOpi_2 is contained inside HOpi_ω. Thus the diverging HOpi_2 process Q_0 can be written in HOpi_ω:

$$\overline{a}\langle x \mapsto S_0\rangle \mid S_0 \text{ where } S_0 = a(y).(y\lfloor\star\rfloor \mid \overline{a}\langle y\rangle)$$

2.3.3 PaPi

PaPi is an hybrid language which combines name-passing and process-passing. Messages carried along channels can be either names or processes (as in HOpi_ω). Moreover, a mechanism of *passivation* is added: a process being executed at some location can be frozen, captured, and send on a channel, in order to be executed somewhere else. Passivation can be found in calculi like Kells [SS04, HPH$^+$09] or Homer [HGB04].

As a result, the syntax of PaPi contains the syntax of π and the one of HOpi_2 but also *locations* $p(\!|P|\!)$ and *passivation actions* $p(X) \triangleright P$.

$$P ::= \mathbf{0} \mid P \mid P \mid (\boldsymbol{\nu} p)\ P \mid \overline{p}\langle q \rangle.P \mid p(x).P \mid !p(x).P \mid p(\!|P|\!) \mid \overline{p}\langle P \rangle.P \mid p(X).P \mid p(X) \triangleright P \mid X$$

Processes can be executed inside locations. Therefore, evaluation contexts can contain locations:

$$\mathbf{E} ::= [\,] \mid (\boldsymbol{\nu} p)\ \mathbf{E} \mid (\mathbf{E} \mid P) \mid l(\!|\mathbf{E}|\!)$$

The operational semantics, as expected, combines communications and replicated communications from π, communications from HOpi_2, and passivation.

$$(\textbf{PaComN})\ \frac{}{\mathbf{E}[\overline{p}\langle q \rangle.P_1 \mid p(x).P_2] \to \mathbf{E}[P_1 \mid P_2\{q/x\}]}$$

$$(\textbf{PaTrig})\ \frac{}{\mathbf{E}[\overline{p}\langle q \rangle.P_1 \mid !p(x).P_2] \to \mathbf{E}[P_1 \mid P_2\{q/x\} \mid !p(x).P_2]}$$

$$(\textbf{PaComP})\ \frac{}{\mathbf{E}[\overline{p}\langle Q \rangle.P_1 \mid p(X).P_2] \to \mathbf{E}[P_1 \mid P_2\{Q/X\}]} \qquad (\textbf{PaPass})\ \frac{}{\mathbf{E}[l(\!|Q|\!) \mid l(X) \triangleright P] \to \mathbf{E}[P\{Q/X\}]}$$

$$(\textbf{PaCong})\ \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

Here are some examples of PaPi processes:

$$(\mathit{Dup}) \quad c(r).l(X) \triangleright \left(l(\!|X|\!) \mid (\boldsymbol{\nu} l')\,(\overline{r}\langle l' \rangle \mid l'(\!|X|\!)) \right)$$
$$(\mathit{Res}) \quad c(l).l(X) \triangleright l(\!|P_0|\!) \qquad\qquad (\mathit{DynUpd}) \quad c(l).d(X).(\, l(Y) \triangleright l(\!|X|\!)\,)$$
$$(\mathit{Coloc}) \quad l_1(X) \triangleright \left(l_2(Y) \triangleright (l_1(\!|X|Y|\!) \mid l_2(\!|\mathbf{0}|\!)) \right)$$

We briefly explain these definitions.

(*Dup*) performs code duplication: when a message is received on channel c, the computation running at location l is duplicated, and the location of the new copy is sent back on r, the channel transmitted along c.

Process (*Res*) (reset): upon reception of a location name l along c, the computation taking place at l is replaced with P_0, that can be considered as a start state. Essentially the same "program" can be used when we want to replace the code running at l with a new version, that is transmitted along some channel d: this is a form of dynamic update (process *DynUpd*).

"Co-localisation": processes running at locations l_1 and l_2 are put together, and computation proceeds within location l_1. This might trigger new interactions between formerly separated processes. This is a form of *objective mobility* (running computations are being moved around).

2.4 λ-calculus

In this thesis, we will be willing to check the possibility of using our methods for termination in concurrent systems to ensure termination in sequential functional settings. Further, we will apply one of our proof

method to a sequential functional language with side-effects. In both cases we will use the λ-calculus as a model for functional computation. In this section, we propose a very short introduction to λ-calculus by presenting the syntax and operational semantics of the standard strong λ-calculus (see [Bar84] for further details), the common evaluation strategies, and the encoding of one this strategy into the π-calculus (see [SW01] for full details). The syntax and semantics we use for impure λ-calculus (see for instance [Bou07]) will be presented later, as to ease readability, we will put some typing information inside the syntax.

2.4.1 Terms and β-reduction

We suppose that we have an infinite number of variable $x, y, z, \ldots$ Terms, denoted in the following with $M, N, \ldots$ are constructed according to the following grammar:

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

x is a variable, $M\ N$ is the application of the function M to its argument N and $\lambda x.M$ is an abstraction, the function that maps x to M. As a consequence, the λ operator actually binds the variable x in M. As usual, we suppose that our terms abide a Barendregt convention enforcing that bound variables are pairwise distinct and are distinct from free variables. Moreover, we write $M\ N_1\ N_2$ for $(M\ N_1)\ N_2$.

As we did for the π-calculi, we give a context-based operational semantics for the λ-calculus. Evaluation contexts for the strong λ-calculus are given by the following grammar:

$$\mathbf{E} = [\,] \mid \lambda x.\mathbf{E} \mid \mathbf{E}\ N \mid M\ \mathbf{E}$$

The semantics is given by only one rule, called *β-reduction* in the following:

$$(\mathbf{Beta})\frac{}{\mathbf{E}[\lambda x.M\ N] \twoheadrightarrow \mathbf{E}[M\{N/x\}]}$$

Here are some examples of reductions:

- $(\lambda f.\lambda z.f\ z)\ (\lambda x.x)\ (\lambda y.y) \twoheadrightarrow (\lambda z.(\lambda x.x)\ z)\ (\lambda y.y) \twoheadrightarrow (\lambda x.x)\ (\lambda y.y) \twoheadrightarrow (\lambda y.y)$
- $(\lambda f.\lambda z.f\ z)\ (\lambda x.x)\ (\lambda y.y) \twoheadrightarrow (\lambda z.(\lambda x.x)\ z)\ (\lambda y.y) \twoheadrightarrow (\lambda z.z)\ (\lambda y.y) \twoheadrightarrow (\lambda y.y)$
- $(\lambda x.x\ x)\ (\lambda y.y\ y) \twoheadrightarrow (\lambda y_1.y_1\ y_1)\ (\lambda y_2.y_2\ y_2) \twoheadrightarrow \ldots$

The last example, often denoted by Ω, is the simplest example of non-termination in λ-calculus.

2.4.2 Simple types

Termination in λ has been thoroughly studied (see for instance [GTL89]). The use of simple types discipline ensures termination.

Simple types are given by the following grammar:

$$T ::= \mathbb{1} \mid T \to T$$

As in the simply-typed π-calculus, we use the typing context Γ as an oracle associating a unique type to each variable, be it bound or free. We use *à la Church* typing, that is that every typable term is given a type *a priori*. However, we allow us not to annotate bound variables with type annotations, as their types are given by the oracle Γ. A term is simply-typed if its typability can be deduced from the following rules:

$$(\mathbf{STVar})\frac{\Gamma(x) = T}{\Gamma \vdash^{ST} x : T} \qquad (\mathbf{STLam})\frac{\Gamma(x) = T_1 \qquad \Gamma \vdash^{ST} M : T_2}{\Gamma \vdash^{ST} \lambda x.M : T_1 \to T_2}$$

$$(\mathbf{STApp})\frac{\Gamma \vdash^{ST} M : T_1 \to T_2 \qquad \Gamma \vdash^{ST} N : T_1}{\Gamma \vdash^{ST} M\ N : T_2}$$

Notice that the term $\lambda x.x\ x$ cannot be typed, because it raised the unsatisfiable constraint $\Gamma(x) = \Gamma(x) \to T$. As a consequence, Ω is rejected by this type system. The following theorem states its soundness:

Theorem 2.4.1 (Soundness of simple types)
Let M be a λ-term, if $\Gamma \vdash^{ST} M : T$ then M strongly terminates.

Proof.
There exists several proofs of this theorem. Ones use combinatorial arguments (for instance [DN07]), but the most well-know proof ([GTL89]) use the power of realisability. □

Remark 2.4.2 (Simple types in λ and π) *One has to notice that the simple types we defined above for π-calculi do not ensure termination, as the diverging examples we proposed are all simply-typable.*

2.4.3 Strategies

One can deduce from the examples of reduction presented above that the reduction relation for the strong λ-calculus is not a function. In order to write concrete functional programs, on can be interested in a calculus for which it is the case. An *evaluation strategy for the λ-calculus* is a relation included into $\twoheadrightarrow$ which is a function. We present here the three standard evaluation strategies for λ-calculus, left-to-right and right-to-left Call-by-Value (written CbV in the following) and Call-by-Name (written CbN).

Values are defined by the following grammar: $V ::= \lambda x.M \mid x$.

Evaluation contexts for left-to-right CbV, right-to-left CbV and CbN are given, respectively by the following grammars:

1. $\mathbf{E} = \mathbf{E}\ M \mid V\ \mathbf{E}$,
2. $\mathbf{E} = \mathbf{E}\ V \mid M\ \mathbf{E}$,
3. $\mathbf{E} = \mathbf{E}\ M$.

β-reduction for the two CbV strategies is given by:

$$(\mathbf{Beta})\frac{}{\mathbf{E}[\lambda x.M\ V] \twoheadrightarrow \mathbf{E}[M\{V/x\}]}$$

β-reduction for CbN is given by

$$(\mathbf{Beta})\frac{}{\mathbf{E}[\lambda x.M\ N] \twoheadrightarrow \mathbf{E}[M\{N/x\}]}$$

One can check easily that this actually defines three evaluation strategies.

Left-to-right CbV begins by reducing the function, then it reduces its argument and endly apply the function to the argument. Right-to-left begins by reducing the argument and then reduces the function. CbN reduces the function and apply it directly to the argument. Notice that these three strategies do not reduces inside the scope of an abstraction $\lambda x.M$

2.4.4 Encodings into π

The message-passing π-calculus can be considered more expressive than λ in the sense that the strategies presented above can be encoded inside the π-calculus. We dot not recall here all the details of the encoding (found in [SW01]), which use the standard Continuation-Passing-Style encoding from λ to λ and an intermediate encoding in HOpi_ω.

As we will focus particularly on the left-to-right CbV encoding from the λ-calculus in the π-calculus in this document, we choose to present only this one.

Definition 2.4.3 (CbV-encoding from λ to π)
The encoding is parametric w.r.t. a π-channel p:

1. $[\lambda x.M]_p = (\nu y)\ \overline{p}\langle y\rangle.!y(x,q).[M]_q$
2. $[x]_p = \overline{p}\langle x\rangle$
3. $[M\ N]_p = (\nu q, r)([M]_q \mid [N]_r \mid q(y).r(z).\overline{y}\langle z, p\rangle)$

A useful property is that simply-typable λ-terms are encoded into simply-typable π-processes, following the discipline: $[\mathbb{1}]_p = \mathbb{1}$ and $[T_1 \to T_2]_p = \sharp([T_1]_p, \sharp([T_2]_p))$

Theorem 2.4.4 (Encoding of simple types)
If $\Gamma' \vdash^{ST} M : T$, then $\Gamma \vdash^{\pi-ST} [M]_p$ with $\Gamma(p) = T$.

Proof. *See Theorem 16.2.1 in [SW01].* □

Actually, the whole typing context Γ can be deduced only from Γ'. An interesting point is that the encoding ensures the following termination property:

Theorem 2.4.5 (Encoding termination)
If $[M]_p$ terminates, then M terminates for the call-by-value strategy.

Remark 2.4.6 (Termination for a strategy) *Notice that a term M may terminates for the call-by-value strategy and diverges in the full λ-calculus. Take for instance $(\lambda x.\star)\ (\lambda y.\Omega)$. In CbV-$\lambda$, one can only perform the outer application which yields $\star$, meaning that this terminates whereas in the setting of the full λ-calculus, one is able to reduce Ω under λy and, as result, obtain a diverging computation.*

Proof. *See Lemma 15.3.23 in [SW01].* □

Remark 2.4.7 (Termination of CbV-λ using the termination of π) *The result of Theorem 2.4.5 hints a new way to prove termination in the setting of call-by-value λ-calculus. One can encode a candidate λ-term M into the π-calculus and use a method to prove termination for π-processes to ensure that $[M]_p$ terminates. Theorem 2.4.5 ensures that it is sufficient to obtain termination, according to the Call-by-value strategy, of the original process M. As a consequence, in Section 6.1, we will study if it is possible to prove the termination of simply-typed λ-calculus using the termination method we will present in Section 3.1.*

An example of reduction As an example of reduction through the encoding, consider a very simple λ-term, the identity applied to itself $(\lambda x.x)\ (\lambda y.y)$. This process is trivially terminating as it can perform only one reduction step, yielding $(\lambda y.y)$. Its encoding into π on channel p_1, according to Definition 2.4.3 is:

$$P_{II} = (\nu q_1, r_1)\ (\nu y_2)\ (\overline{q_1}\langle y_2\rangle.!y_2(x,q_2).\overline{q_2}\langle x\rangle) \mid (\nu y_3)\ (\overline{r_1}\langle y_3\rangle.!y_3(y,q_3).\overline{q_3}\langle y\rangle) \mid q_1(y_1).r_1(z_1).\overline{y_1}\langle z_1, p_1\rangle)$$

P_{II} is able to perform a reduction on q_1 and then a reduction on r_1, these two reductions correspond to administrative reductions inside the application, the function send on q_1 its name y_2 and the argument send on r_1 its name, y_3. Thus:

$$P_{II} \to\to (\nu q_1, r_1, y_2, y_3)\ (!y_2(x,q_2).\overline{q_2}\langle x\rangle) \mid (!y_3(y,q_3).\overline{q_3}\langle y\rangle) \mid \overline{y_2}\langle y_3, p_1\rangle)$$

Now the application itself takes place, we send to the function y_2 the name of its argument y_3 and the return channel p_1. The name of the argument instantiates the bound variable x in the body of the function:

$$P_{II} \to\to\to (\nu q_1, r_1, y_2, y_3)\ (!y_2(x,q_2).\overline{q_2}\langle x\rangle) \mid (!y_3(y,q_3).\overline{q_3}\langle y\rangle) \mid \overline{p_1}\langle y_3\rangle)$$

The process cannot perform further reductions. The restricted names r_1, q_1, used for the administrative part of the application no longer exist in a prefix. The restricted name y_2 is not extruded, as a consequence,

the replication on y_2 can be seen as dead code. The resulting process send the name y_3 of the identity function on the return channel p_1. One can compare this process to $[\lambda y.y]_p = (\nu y_3)\ \overline{p_1}\langle y_3\rangle.!y_3(y, q_3).\overline{q_3}\langle y\rangle$ and hint that they are behaviourally equivalent.

We can see that the translation of simple λ-terms gives complex π-processes and that one step of β-reduction is simulated by several steps of communication. This makes the use of examples to illustrate the method quoted in Remark 2.4.7 a bit tedious.

Chapter 3

Type Systems for termination in message-passing π-calculus

In this section, we propose new presentations and original proofs (within a common framework) for three type systems found in [DS06], we also present the contributions of [DHS08]: a new, more expressive type system and a hybrid method for ensuring termination, combining a type system and a run-time analysis. The three systems from [DS06] have an increasing expressiveness (each one is contained in the following, in the sense that a process typable with the first system is still typable with the second one, and so on). These systems enforce termination by associating levels to names, weights to processes (based on the levels of the outputs inside the process) and ensure that at each reduction step, a well-founded measure (related to the weight) decreases.

3.1 A first type system

We present here a first type system, adapted from the first one presented in [DS06]. Most of the weight-based analysis we present in this document are based on this type system. The main principle of this system is the use of weights and levels to prevent potential loops from arising. The goal is to force the process being spawned in a replicated communication (i.e., when $\overline{a}\langle v\rangle \mid !a(x).P \to P\{v/x\} \mid !a(x).P$, the process $P\{v/x\}$) to be "lighter" than the prefix (in this case $\overline{a}\langle v\rangle$) consumed to trigger the replication. Thus, in every communication, something is "lost", and no infinite reduction sequence starting from a typable process can arise.

In this setting, each name is *a priori* associated to a *level* by the typing context. Indeed, our type system is an extension of the *à la Church* simple type systems presented in Section 2.

Definition 3.1.1 (Types for names)

$$T ::= \mathbb{1} \mid \sharp^k T$$

where k in a integer > 0.

In other words, types for names use the backbone of simple-types, and we decorate each type constructor $\sharp(\cdot)$ with an integer.

3.1.1 Typing rules

Typing rules gives weight to processes, intuitively corresponding to the maximum level of an output not guarded by a replication inside this process. As a consequence, typing judgements for processes are written $\Gamma \vdash P : n$, meaning that in the typing context Γ, the process P is well-typed and is given weight n. When

typing $!a(x).P$, we ensure that the weight of P is strictly smaller than the level of a. Thus, when the replicated input is triggered, the outputs released have levels strictly smaller than the output consumed. Typing rules for processes are given in Figure 3.1

$$(\mathbf{Nil})\frac{}{\Gamma \vdash \mathbf{0} : 0} \qquad (\mathbf{Par})\frac{\Gamma \vdash P_1 : n_1 \qquad \Gamma \vdash P_2 : n_2}{\Gamma \vdash P_1 \mid P_2 : \max(n_1, n_2)} \qquad (\mathbf{Res})\frac{\Gamma \vdash P : n \qquad \Gamma(a) = \sharp^k T}{\Gamma \vdash (\nu a)\ P : n}$$

$$(\mathbf{Out})\frac{\Gamma \vdash P : n \qquad \Gamma(a) = \sharp^k T \qquad \Gamma(v) = T}{\Gamma \vdash \overline{a}\langle v\rangle.P : \max(n, k)} \qquad (\mathbf{In})\frac{\Gamma \vdash P : n \qquad \Gamma(a) = \sharp^k T \qquad \Gamma(x) = T}{\Gamma \vdash a(x).P : n}$$

$$(\mathbf{Rep})\frac{\Gamma \vdash P : n \qquad \Gamma(a) = \sharp^k T \qquad \Gamma(x) = T \qquad k > n}{\Gamma \vdash !a(x).P : 0}$$

Figure 3.1: Typing rules for processes

The side-condition $\Gamma(a) = \sharp^k T$ in rule (**Res**) is only here to recall the type of the restricted name a.

Note that if a is a channel that carries names, for instance say a is of type $\sharp^k \sharp^l T$, then every name carried on a will have the same level l, according to the rules (**In**), (**Out**) and (**Rep**). This prevents diverging processes like D_3 (defined in previous section) from being typed. We study in Section 6.1 how this can be the cause of the difficulties we face when trying to type $\lambda_{\mathbf{ST}}$ through an encoding into the π-calculus.

3.1.2 Examples

The examples presented in Section 2.2.3 are ruled out by our type system. $D_1 = !a.\overline{a} \mid \overline{a}$ is not typable: we have to give type $\sharp^k \mathbb{1}$ for some k to a. Then the typing rules give $\Gamma \vdash \overline{a} : k$ and trying to type $!a.\overline{a}$ gives the constraint $k < k$.

For a similar reason, we cannot type $D_2 = !a.\overline{b} \mid !b.\overline{a} \mid \overline{a}$, as we have to give type $\sharp^k \mathbb{1}$ to a and type $\sharp^l \mathbb{1}$ to b, for some k and l. Thus, to type the two replicated subprocesses, k and l have to satisfy the constraints $k > l$ and $l > k$.

When trying to type $D_3 = c(x).!a.\overline{x} \mid \overline{a} \mid \overline{c}\langle a\rangle$, c is given type $\sharp^l \sharp^k \mathbb{1}$ for some k and l. Typing rules (**In**) and (**Out**) impose that x and a have the same type $\sharp^k \mathbb{1}$; therefore, the replicated subprocess raises the constraint $k > k$, as in D_1.

Consider $S_1 = !a.(\overline{b} \mid \overline{b} \mid \overline{c}) \mid !b.(\overline{c} \mid \overline{c}) \mid \overline{a} \mid \overline{c}$. We are able to type S_1 with the following type assignment:

$$\Gamma(a) = \sharp^3 \mathbb{1} \qquad \Gamma(b) = \sharp^2 \mathbb{1} \qquad \Gamma(c) = \sharp^1 \mathbb{1}$$

Thus, the constraints which have to be satisfied in order to typecheck the two replications are $3 > 2$ and $2 > 1$. One can remark that the multiset of levels of the *available* (not under a replication) outputs decreases along reductions. For instance, a possible reduction sequence gives $\{3, 1\} \to \{2, 2, 1, 1\} \to \{2, 1, 1, 1, 1\} \to \{1, 1, 1, 1, 1, 1\}$. This decreasing is actually the crux of the termination proof, and the multiset of levels of the available outputs will be used a a decreasing measure in the soundness proof.

3.1.3 Termination proof

As said above, proving soundness for a type system for termination is proving that if $\Gamma \vdash P : n$ for some n, then P terminates. We first prove some common properties of our type system which will be used later, that is, that typability is stable by the use of subject congruence, preserved by well-typed substitution (substitutions where the two names involved have same type) and by reduction.

Fact 3.1.2 (Subject Congruence)
If $P \equiv Q$, then $\Gamma \vdash P : n$ iff $\Gamma \vdash Q : n$.

Proof.
Easily done by induction on the derivation for $P \equiv Q$. We use the fact that associativity, commutativity and neutrality of 0 *hold for* max.

□

In the following lemma, we consider that the two names involved in the substitution (x and v) have the same type (this is done by the condition $\Gamma(x) = \Gamma(v)$). This will always hold for substitutions coming from of a reduction, as the two names are carried on a same channel.

Lemma 3.1.3 (Subject Substitution)
If $\Gamma(v) = \Gamma(x)$, $\Gamma \vdash P : n$ and x is not bound in P, then $\Gamma \vdash P\{v/x\} : n$.

The condition x is not bound in P is required as we do not want the substitution to interfere with bound names. As the processes we consider abide the Barendregt convention defined above, when a substitution occurs because of a reduction, this condition is always satisfied. As a consequence, to ease the readability of proofs, from now, when we state subject substitution properties, we will implicitly consider that the name x being substituted does not appear bound in the process to which the substitution is applied.
Proof. *By induction on the typing judgement:*

- *Case* (**Nil**). *Easy as* $\mathbf{0}\{v/x\}$ *is* $\mathbf{0}$.
- *Case* (**Par**). *We have* $P = P_1 \mid P_2$. *We derive* $\Gamma \vdash P_1 : n_1$ *and* $\Gamma \vdash P_2 : n_2$ *and* $n = \max(n_1, n_2)$. x *is not bound in* P_1 *nor in* P_2. *We use the induction hypothesis twice to get* $\Gamma \vdash P_1\{v/x\} : n_1$ *and* $\Gamma \vdash P_2\{v/x\} : n_2$. *We use rule* (**Par**) *to get* $\Gamma \vdash P_1\{v/x\} \mid P_2\{v/x\} : \max(n_1, n_2)$. *We conclude as* $(P_1 \mid P_2)\{v/x\} = (P_1\{v/x\}) \mid (P_2\{v/x\})$ *and* $n = \max(n_1, n_2)$.
- *Case* (**Res**). *We have* $P = (\nu a)\ P_1$. *We derive* $\Gamma \vdash P_1 : n$. *Notice, that the condition on* x *prevents the case* $a = x$ *from happening. We use the induction hypothesis to get* $\Gamma \vdash P_1\{v/x\} : n$. *We use rule* (**Res**) *to conclude.*
- *Case* (**Out**). *We have* $P = \overline{a}\langle w\rangle.P_1$. x *is not bound in* P_1. *We derive* $\Gamma \vdash P_1 : n_1$, $\Gamma(a) = \sharp^k T$, $\Gamma(w) = T$ *and* $n = \max(n_1, k)$. *The induction hypothesis gives* $\Gamma \vdash P_1\{v/x\} : n_1$. *We discuss:*
  - *Either* $x \neq w$ *and* $x \neq a$. *Then* $P\{v/x\}$ *is* $\overline{a}\langle w\rangle.(P_1\{v/x\})$. *We use rule* (**Out**) *to conclude, as* $n = \max(n_1, k)$.
  - *Or* $x \neq w$ *and* $x = a$. *Then* $P\{v/x\}$ *is* $\overline{v}\langle w\rangle.(P_1\{v/x\})$. *We use rule* (**Out**) *as* $\Gamma(v) = \Gamma(x) = \sharp^k T$ *to get* $\Gamma \vdash P\{v/x\} : \max(n_1, k)$. *We conclude.*
  - *Or* $x = w$ *and* $x \neq a$. *Then* $P\{v/x\}$ *is* $\overline{a}\langle v\rangle.(P_1\{v/x\})$. *We use rule* (**Out**) *as* $\Gamma(v) = \Gamma(w) = T$ *to get* $\Gamma \vdash P\{v/x\} : \max(n_1, k)$. *We conclude.*
  - *As the processes we consider are simply-typed, Case* $x = w = a$ *cannot happen. Indeed the typing rule* (**Out**) *gives* $\Gamma(a) = \sharp^k \Gamma(w)$ *from which we deduce* $\Gamma(a) \neq \Gamma(w)$.
- *Case* (**Rep**). *We have* $P = !a(y).P_1$ *and* x *is not bound in* P_1. *We derive* $\Gamma \vdash P_1 : n_1$, $\Gamma(a) = \sharp^k T$, $\Gamma(y) = T$, $k > n_1$ *and* $n = 0$. *The induction hypothesis gives* $\Gamma \vdash P_1\{v/x\} : n_1$. *Condition on* x *prevents the case* $x \neq y$ *from happening. We discuss:*
  - *Either* $x \neq a$. *Then* $P\{v/x\}$ *is* $!a(y).(P_1\{v/x\})$. *As* $k > n_1$, *we use rule* (**In**) *to conclude.*
  - *Or* $x = a$. *Then* $P\{v/x\}$ *is* $!v(y).(P_1\{v/x\})$. *We use rule* (**In**), *as* $\Gamma(v) = \Gamma(x) = \sharp^k T$ *and* $k > n_1$, *to get* $\Gamma \vdash P\{v/x\} : n$.
- *Case* (**In**) *is contained in Case* (**Rep**).

□

As we choose a context-based semantics, we need the following lemma in order to prove Subject Reduction (Lemma 3.1.5). This lemma states two things: first, when an evaluation context containing a process is typable with a weight l, then the process inside the hole is typable with a smaller weight $l' \leq l$. Moreover, we can replace this inner process by any typable process with a smaller weight, the resulting process will still be typable.

Lemma 3.1.4 (Context typing)
If $\Gamma \vdash \mathbf{E}[P] : l$ then:

1. *$\Gamma \vdash P : l'$ for some $l' \leq l$.*
2. *For all P_0 s.t. $\Gamma \vdash P_0 : l_0$ with $l_0 \leq l'$, there exists $l_{(0)}$ such that $\Gamma \vdash \mathbf{E}[P_0] : l_{(0)}$ with $l_{(0)} \leq l$.*

Proof. *By structural induction on $\mathbf{E}$:*

- *Case $[\,]$. Condition 1 holds trivially and condition 2 holds by setting $l_{(0)} = l_0 \leq l' = l$.*
- *Case $(\boldsymbol{\nu} a)\ \mathbf{E}_2$. Condition 1 holds with $l' = l$ and condition 2 holds by setting $l_{(0)} = l_0 \leq l' = l$.*
- *Case $\mathbf{E} = \mathbf{E}_2 \mid P_1$. We derive $\Gamma \vdash \mathbf{E}_2[P] : l_2$ and $\Gamma \vdash P_1 : l_1$ with $l = \max(l_2, l_1)$. The induction hypothesis gives $\Gamma \vdash P : l'$ with $l' \leq l_2$, as $l_2 \leq l$ we get condition 1. The induction hypothesis also gives $\Gamma \vdash \mathbf{E}_2[P_0] : l_{(2)}$ with $l_{(2)} \leq l_2$. We set $l_{(0)} = \max(l_{(2)}, l_1)$ and we get condition 2.*

□

The following lemma states that typability is preserved by reduction. The weight however, is not preserved. The weight accounts for the maximum level of the subject of an output not guarded by a replication. Thus performing a communication can make this weight decrease. Yet, the weight cannot increase.

Lemma 3.1.5 (Subject Reduction)
If $\Gamma \vdash P : n$ and $P \to P'$, then $\Gamma \vdash P' : n'$ with $n' \leq n$.

Proof. *By induction on the derivation of $P \to P'$:*

- *Case (**cong**). Easily done using twice Fact 3.1.2 and the induction hypothesis.*
- *Case (**comm**). We have $P = \mathbf{E}[a(x).P_1 \mid \overline{a}\langle v\rangle.P_2]$ and $P' = \mathbf{E}[P_1\{v/x\} \mid P_2]$. Notice that x is not bound in the process P_1, as it bound by $a(x)$. By Lemma 3.1.4.1, we get $\Gamma \vdash a(x).P_1 \mid \overline{a}\langle v\rangle.P_2 : l$ for some $l \leq n$. We deduce $\Gamma \vdash P_1 : l_1$, $\Gamma \vdash P_2 : l_2$, $\Gamma(a) = \sharp^k T$, $\Gamma(x) = \Gamma(v) = T$ and $l = \max(l_1, l_2, k)$. Lemma 3.1.3 gives $\Gamma \vdash P_1\{v/x\} : l_1$. Using rule (**Par**), we derive $\Gamma \vdash P_1\{v/x\} \mid P_2 : \max(l_1, l_2)$. As $\max(l_1, l_2) \leq \max(l_1, l_2, k) = l$, we use Lemma 3.1.4.2 to get $\Gamma \vdash \mathbf{E}[P_1\{v/x\} \mid P_2] : n'$ with $n' \leq n$.*
- *Case (**trig**). We have $P = \mathbf{E}[!a(x).P_1 \mid \overline{a}\langle v\rangle.P_2]$ and $P' = \mathbf{E}[!a(x).P_1 \mid P_1\{v/x\} \mid P_2]$. Notice that, as above, x is not bound in P_1. By Lemma 3.1.4.1, we get $\Gamma \vdash a(x).P_1 \mid \overline{a}\langle v\rangle.P_2 : l$ for some $l \leq n$. We derive $\Gamma \vdash !a(x).P_1 : 0$, $\Gamma \vdash P_1 : l_1$, $\Gamma \vdash P_2 : l_2$, $\Gamma(a) = \sharp^k T$, $\Gamma(x) = \Gamma(v) = T$, $l = \max(l_2, k)$ and $k > l_1$. Lemma 3.1.3 gives $\Gamma \vdash P_1\{v/x\} : l_1$. Using rule (**Par**) twice, we derive $\Gamma \vdash !a(x).P_1 \mid P_1\{v/x\} \mid P_2 : \max(l_1, l_2)$. As $k > l_1$, $\max(l_1, l_2) \leq \max(l_2, k) = l$. Therefore, we use Lemma 3.1.4.2 to get $\Gamma \vdash \mathbf{E}[!a(x).P_1 \mid P_1\{v/x\} \mid P_2] : n'$ with $n' \leq n$.*

□

To prove soundness we prove that the multiset of all available (not under another replication) outputs decreases at each reduction. We make this measure explicit, noted **Os**(), in the following. The multiset $\mathbf{Os}(P)$ contains the levels of the subject of every output prefix of P which is not under a replication.

Definition 3.1.6 (Available outputs)
We define the multiset of available outputs of P w.r.t. a typing judgement $\Gamma \vdash P : n$:

$$\mathbf{Os}(\mathbf{0}) = \emptyset \qquad \mathbf{Os}(P_1 \mid P_2) = \mathbf{Os}(P_1) \uplus \mathbf{Os}(P_2) \qquad \mathbf{Os}((\nu a)\ P) = \mathbf{Os}(P) \qquad \mathbf{Os}(a(x).P_1) = \mathbf{Os}(P_1)$$

$$\mathbf{Os}(\overline{a}\langle v\rangle.P_2) = \{k\} \uplus \mathbf{Os}(P_2) \text{ if } \Gamma(a) = \sharp^k T \qquad \mathbf{Os}(!a(x).P_1) = \emptyset$$

This notion is easily extended to typed evaluation contexts by considering $\mathbf{Os}([\,]) = \emptyset$

Remark 3.1.7 *One can notice that, by using typing judgements of the form $\Gamma \vdash P : \mathbf{Os}(P)$ (i.e. multisets for weights instead of integers), we can prove soundness along with subject reduction. However, for the sake of readability of the first instalment of a weight-based type system, we prefer to stick to an integer weight for processes. We will use multisets in the further sections.*

We first prove some easy results about available outputs. What we need for soundness, is to prove that, at each reduction step, the multiset of levels of the available outputs decreases. As we used a context-based semantics, we need a lemma allowing us to state that, when performing a reduction, the measure of the context is unaffected:

Lemma 3.1.8 (Available outputs and context)
If $\Gamma \vdash \mathbf{E}[P] : n$, then $\mathbf{Os}(\mathbf{E}[P]) = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P)$.

Proof. *By structural induction over $\mathbf{E}$:*

- *Case $[\,]$. Then $\mathbf{Os}(\mathbf{E}[P]) = \mathbf{Os}(P) = \mathbf{Os}(P) \uplus \mathbf{Os}([\,])$.*
- *Case $(\nu a)\ \mathbf{E}_2$. Then $\mathbf{Os}(\mathbf{E}[P]) = \mathbf{Os}((\nu a)\ \mathbf{E}_2[P])$ which is, by Definition 3.1.6, $\mathbf{Os}(\mathbf{E}_2[P])$. We use the induction hypothesis to get $\mathbf{Os}(\mathbf{E}_2[P]) = \mathbf{Os}(\mathbf{E}_2) \uplus \mathbf{Os}(P)$. As, by Definition 3.1.6, $\mathbf{Os}(\mathbf{E}) = \mathbf{Os}(\mathbf{E}_2)$, we conclude.*
- *Case $\mathbf{E}_2 \mid P_1$. Then $\mathbf{Os}(\mathbf{E}[P]) = \mathbf{Os}(\mathbf{E}_2[P] \mid P_1)$ which is, by Definition 3.1.6 $\mathbf{Os}(\mathbf{E}_2[P]) \uplus \mathbf{Os}(P_1)$. We use the induction hypothesis, to get $\mathbf{Os}(\mathbf{E}_2[P]) = \mathbf{Os}(\mathbf{E}_2) \uplus \mathbf{Os}(P)$. As, by Definition 3.1.6, $\mathbf{Os}(\mathbf{E}) = \mathbf{Os}(\mathbf{E}_2) \uplus \mathbf{Os}(P_1)$, we conclude.*

□

Then we state two facts saying that the available outputs are preserved by the use of subject congruence, which is necessary to accommodate the use of (**cong**) in reduction derivations and that they are also preserved by well-typed substitutions (i.e. substitutions where the two names involved have the same type).

Fact 3.1.9 (Available outputs and structural congruence)
If $P \equiv Q$ and $\Gamma \vdash P : n$ then $\mathbf{Os}(P) = \mathbf{Os}(Q)$.

Proof. *First we notice that we are allowed to write $\mathbf{Os}(Q)$ as Fact 3.1.2 states that Q is also typable. We proceed by induction on the derivation of $P \cong Q$ (using symmetry if necessary), using the associativity, commutativity and neutrality of $\emptyset$ for the operator $\uplus$.* □

Fact 3.1.10 (Available outputs and substitution)
If $\Gamma \vdash P : n$ and $\Gamma(v) = \Gamma(x)$, then $\mathbf{Os}(P) = \mathbf{Os}(P\{v/x\})$.

Proof.
First, Lemma 3.1.3 allows us to write $\mathbf{Os}(P\{v/x\})$. We proceed by induction on the typing judgement, the interesting case being $P = \overline{a}\langle v\rangle.P_1$. Suppose $\Gamma(a) = \sharp^k T$. Either $a \neq x$ and $\mathbf{Os}(P\{v/x\}) = \{k\} \uplus \mathbf{Os}(P_1\{v/x\})$ and we use the induction hypothesis to conclude, or $a = x$ and, as $\Gamma(v) = T$ and $\Gamma(x) = \Gamma(a) = \sharp^k T$ and $\mathbf{Os}(P\{v/x\}) = \{k\} \uplus \mathbf{Os}(P_1\{v/x\})$, we use the induction hypothesis to conclude. □

We need to relate the weight of a process and its available outputs, in order to link our type system with the decreasing measure we build. We already stated informally that the weight of a process is the maximum level of the subject of an available output prefix inside this process.

Lemma 3.1.11 (Available output domination)
If $\Gamma \vdash P : n$, *then* $\max(\mathbf{Os}(P)) = n$.

Proof. *By induction over the typing judgement:*

- *Cases* (**Nil**) *and* (**Rep**) *are easy as* $\max(\emptyset) = 0$.
- *Case* (**Res**). $P = (\nu a)\ P_1$ *and we use the induction hypothesis, as, by Definition 3.1.6,* $\mathbf{Os}((\nu a)\ P_1) = \mathbf{Os}(P_1)$ *and as* $\Gamma \vdash P_1 : n$.
- *Case* (**In**). $P = a(x).P_1$ *and we use the induction hypothesis, as, by Definition 3.1.6,* $\mathbf{Os}(a(x).P_1) = \mathbf{Os}(P_1)$ *and as* $\Gamma \vdash P_1 : n$.
- *Case* (**Out**). $P = \overline{a}\langle v\rangle.P_1$. *We derive* $\Gamma \vdash P_1 : n_1$, $\Gamma(a) = \sharp^k T$ *and* $n = \max(k, n_1)$. *The induction hypothesis gives* $\max(\mathbf{Os}(P_1)) = n_1$. *Definition 3.1.6 gives* $\mathbf{Os}(P) = \mathbf{Os}(P_1) \uplus \{k\}$. *Thus* $\max(\mathbf{Os}(P_1) \uplus \{k\}) = \max(\max(\mathbf{Os}(P_1)), k) = n$.

□

Here is the crux of the termination proof; at each reduction step, the measure of available outputs of a process decreases. This holds trivially when the reduction performed involves input as, as stated above, one output disappears and no new available output is spawned. Our type system ensures that, when a replication is triggered, the available outputs which appear (the ones inside the replicated process) are smaller, for the multiset comparison of levels, than the output consumed to trigger the replication.

Lemma 3.1.12 (Decreasing)
If $\Gamma \vdash P : n$ *and* $P \to P'$, *then* $\mathbf{Os}(P') <_{\text{mul}} \mathbf{Os}(P)$.

Proof. *First, we remark that we are allowed to write* $\mathbf{Os}(P')$, *as Lemma 3.1.5 states that* P' *is typable. We proceed by induction on the derivation of* $P \to P'$*:*

- *Case* (**cong**). *Easily done using the induction hypothesis and Fact 3.1.9 twice.*
- *Case* (**comm**). *Typability gives* $\Gamma(a) = \sharp^k T$. *Then* $P = \mathbf{E}[a(x).P_1 \mid \overline{a}\langle v\rangle.P_2]$ *and* $P' = \mathbf{E}[P_1\{v/x\} \mid P_2]$. *We use Lemma 3.1.8 twice and Definition 3.1.6 to get* $\mathbf{Os}(P) = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_1) \uplus \mathbf{Os}(P_2) \uplus \{k\}$ *and* $\mathbf{Os}(P') = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_1\{v/x\}) \uplus \mathbf{Os}(P_2)$. *We use Fact 3.1.10 to get* $\mathbf{Os}(P_1\{v/x\}) = \mathbf{Os}(P_1)$. *As* $\mathbf{Os}(P') \subsetneq \mathbf{Os}(P)$, *we conclude by Proposition 2.1.1.*
- *Case* (**trig**). *Then* $P = \mathbf{E}[!a(x).P_1 \mid \overline{a}\langle v\rangle.P_2]$ *and* $P' = \mathbf{E}[!a(x).P_1 \mid P_1\{v/x\} \mid P_2]$. *From* $\Gamma \vdash P : n$, *by Lemma 3.1.4, we derive* $\Gamma \vdash !a(x).P_1 : 0$, $\Gamma \vdash P_1 : l_1$, $\Gamma(a) = \sharp^k T$ *and* $k > l_1$. *We use Lemma 3.1.8 twice and Definition 3.1.6 to get* $\mathbf{Os}(P) = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_2) \uplus \{k\}$ *and* $\mathbf{Os}(P') = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_1\{v/x\}) \uplus \mathbf{Os}(P_2)$. *Lemma 3.1.3 gives* $\Gamma \vdash P_1\{v/x\} : l_1$. *We apply Lemma 3.1.11 to deduce that* $\max(\mathbf{Os}(P_1\{v/x\})) = l_1$. *By Proposition 2.1.2,* $\mathbf{Os}(P_1\{v/x\}) <_{\text{mul}} \{l_1 + 1\}$. *As* $k > l_1$, *we finally get* $\mathbf{Os}(P_1\{v/x\}) <_{\text{mul}} \{k\}$. *Applying the definition of multiset comparison, we get* $\mathbf{Os}(P') <_{\text{mul}} \mathbf{Os}(P)$.

□

The measure $\mathbf{Os}(\cdot)$ being well-founded, this allows us to conclude.

Theorem 3.1.13 (Soundness)
If $\Gamma \vdash P : n$, *then* P *terminates.*

Proof. *Suppose, towards a contradiction, that* P *diverges, then there exists an infinite sequence* $(P_i)_{i\in\mathbb{N}}$ *s.t.* $P = P_0$, $\Gamma \vdash P_0 : n$ *and* $\forall i \in \mathbb{N}, P_i \to P_{i+1}$. *Lemma 3.1.5 allows us to state that every* P_i *is typable. Thus we can consider the infinite sequence* $(\mathbf{Os}(P_i))_{i\in\mathbb{N}}$. *Lemma 3.1.12 allows us to state that* $\forall i, \mathbf{Os}(P_{i+1}) <_{\text{mul}} \mathbf{Os}(P_i)$. *This contradicts Theorem 2.1.3.* □

The type system defined here is used as a basis for several other type systems we present or study in this thesis. The notion of decreasing of a multiset measure is the crux of the soundness of these type systems.

3.1.4 Bound on the number of reductions of a typed process

In this section we recall the bound of the number of reduction steps of a typed process, w.r.t. its size. This result was first presented in [DS06]. This bound allows us to have an idea of the expressiveness of our type systems. Indeed, terminating behaviours which ends in a number of steps greater than the bound we propose cannot be captured by our methods.

Definition 3.1.14 (Size of a process)
We define the size of a process P, written $\sharp(P)$, as the number of prefixes inside P.

Remember that there does not exist a rule like $!a(x).P \equiv (a(x).P \mid !a(x).P)$. As a consequence, the size of a process is invariant by structural congruence.

For instance $\sharp(\overline{a}\langle v\rangle \mid !a(x).(\nu c)(\overline{c} \mid \overline{x})) = 4$.

This weight-based type system constraints the number of reduction steps that a typed process can initiate:

Proposition 3.1.15 (Bound on the number of reduction steps)
If $\Gamma \vdash P : n$, then P can perform at most $\sharp(P)^{\sharp(P)+1}$ reduction steps.

Proof.
First, we state that the number of different levels assigned to names being subjects of output prefixes is bounded by the size of the process. Then at each reduction step, one output is consumed and the number of available outputs released is smaller than the size of the initial process (as these outputs are spawned from a replication). Moreover, the levels of these outputs are strictly smaller than the level of the consumed outputs. We add that the initial process cannot have more available outputs than its size. Finally, a process cannot reduce further if it has 0 available outputs. We conclude by stating that the number of reductions is smaller than $\sharp(P).\sharp(P)^{\sharp(P)}$, as each available output found inside P generates at most $\sharp(P)^{\sharp(P)}$ reductions. □

3.2 Refining the analysis

In this section and the following one, we propose new presentations and termination proofs for two refinements of the previous systems found in [DS06]. These two systems have increased expressiveness, at the cost of some more technical proofs.

3.2.1 Input sequences

Processes such as $!a.b.\overline{a}$ (i.e. replicated processes containing recursive calls, but requiring more inputs than the outputs they release when they are triggered) are harmless from the point of view of termination: an output on a and another output on b have to be consumed to obtain a single output on a, preventing loops from arising. However, this process is not typable with the type system of Section 3.1. Indeed, the occurrences of an output on a in the continuation of a replicated input on a will lead to an unsatisfiable constraint of the form $k > k$. In this section, we describe a type system which takes into account the presence of additional inputs (like the one on b in the previous example) by considering replicated input sequences as a whole.

In this system, the types for names are unchanged, only the typing rules for processes are modified.

We no longer use integers as types, as in Section 3.1, instead we adopt a multiset definition of weight. Thus, typing judgements are of the form $\Gamma \vdash^{\kappa} P : M$ where M is a multiset of integers. The typing rules are presented in Figure 3.2.

The rules (**Nil**), (**Par**), (**Res**), (**In**) and (**Out**) are very similar to the ones of Figure 3.1, only using multiset operators ($\emptyset$,$\uplus$) instead of their integer counterpart (0,max). However, rule (**Rep**) changes, now the levels of the whole input sequence are compared, using multiset comparison, to the weight of the continuation.

$$(\mathbf{Nil})\frac{}{\Gamma \vdash^{\kappa} \mathbf{0} : \emptyset} \qquad (\mathbf{Par})\frac{\Gamma \vdash^{\kappa} P_1 : M_1 \qquad \Gamma \vdash^{\kappa} P_2 : M_2}{\Gamma \vdash^{\kappa} P_1 \mid P_2 : M_1 \uplus M_2} \qquad (\mathbf{Res})\frac{\Gamma \vdash^{\kappa} P : M \qquad \Gamma(a) = \sharp^k T}{\Gamma \vdash^{\kappa} (\nu a)\ P : M}$$

$$(\mathbf{In})\frac{\Gamma \vdash^{\kappa} P : M \qquad \Gamma(a) = \sharp^k T \qquad \Gamma(x) = T}{\Gamma \vdash^{\kappa} a(x).P : M} \qquad (\mathbf{Out})\frac{\Gamma \vdash^{\kappa} P : M \qquad \Gamma(a) = \sharp^k T \qquad \Gamma(v) = T}{\Gamma \vdash^{\kappa} \overline{a}\langle v\rangle.P : M \uplus \{k\}}$$

$$(\mathbf{Rep})\frac{\Gamma \vdash^{\kappa} P : M \qquad \forall i, \Gamma(a_i) = \sharp^{k_i} T_i \wedge \Gamma(x_i) = T_i \qquad M <_{\text{mul}} \{k_1, \ldots, k_n\}}{\Gamma \vdash^{\kappa} !a_1(x_1).a_2(x_2)\ldots.a_n(x_n).P : \emptyset}$$

Figure 3.2: Typing rules accommodating input sequences

Remark 3.2.1 *As we want to associate a single typing derivation to a typing judgement, we have to prevent the rule* (**Rep**) *to be used ambiguously. Indeed, the process* $!a.b.\overline{c}$ *can be typed in two different ways with the typing context* Γ*. Either by applying rule* (**Rep**) *to the input sequence* $!a$ *and then typing the continuation* $b.\overline{c}$*, or by applying rule* (**Rep**) *to the input sequence* $!a.b$ *and then typing the continuation* $\overline{c}$*. One can easily prove that we do not lose generality by considering only the maximal input sequences, i.e. by supposing that in rule* (**Rep**)*,* $P \neq a(x).P'$*.*

With this type system, we can recognise as terminating the process $T_2 = !a.\overline{b} \mid !a.b.\overline{a} \mid \overline{a} \mid \overline{b}$ with the typing context $\Gamma(a) = \sharp^2 \mathbb{1}, \Gamma(b) = \sharp^1 \mathbb{1}$. The first replication is typed as $\{1\} <_{\text{mul}} \{2\}$ and the second one is typed as $\{2\} <_{\text{mul}} \{2, 1\}$. Process T_2 is not typable with the previous type system, the second replication is rejected as an available output on a is found in the continuation of a replicated input on a.

From the point of view of the termination proof, we cannot apply directly the one of Section 3.1, as the measure $\mathbf{Os}(P)$ does not necessarily decrease with reduction. Notice that in this section, the measure $\mathbf{Os}(P)$ is identified with the weight of a process P. For instance, consider this reduction sequence from the typable process T_2 (we set $T_2' = !a.\overline{b} \mid !a.b.\overline{a}$):

$$T_2 = (T_2' \mid \overline{a} \mid \overline{b}) \to (T_2' \mid \overline{b}) \to (T_2' \mid \overline{a})$$

The corresponding sequence of weights is $\{1, 2\} \to \{1\} \to \{2\}$. Indeed, the measure increases in the second reduction and the property stated in Lemma 3.1.12 no longer holds. However, one can consider that the process $T_2' \mid \overline{b}$ is an intermediate process where the input sequence $!a.b$ is partially consumed and that the actual decreasing takes place between $T_2' \mid \overline{a} \mid \overline{b}$ and $T_2' \mid \overline{a}$, i.e. between the triggering of the replication and the consumption of the last input prefix of the input sequence.

Termination is proved using commutation of reductions in [DS06]. Here, we use an auxiliary calculus where we remember the outputs we consumed when progressing through input sequences. Not only we find this approach more natural (we do not change the order in which the events happen), but this method will also be used in termination proofs for further calculi where the commutation technique cannot be applied.

Remark 3.2.2 (Structural congruence not closed under prefixes) *The use of structural congruence in rule* (**cong**) *has to be controlled, as input sequences should not disappear. For instance, in our case, we do not want the replicated process* $!a.b.\overline{a}$ *to be structurally congruent to the process* $!a.(\mathbf{0} \mid b.\overline{a})$*, even if they are behaviourally equivalent, as the typing rules cannot be applied the same way to both. Indeed, the first process could be typed using the* (**Rep**) *with the input sequence* $!a.b$ *but not the second one. To sum up, we want input sequences to be something definitely fixed inside a process.*

As a consequence, we prevent structural congruence to be closed under replicated input. Apart from this change, it is defined by the same rules as the one presented in Section 2. We allow us to stick to the symbol $\equiv$ *to denote this new structural congruence.*

Its impact on the semantics is null, should we use the notation $(\mathbf{cong})_o$ *to denote the reduction rule given by the "old" definition of congruence, the following holds:* $P \to P'$ *using the rules* (**com**)*,* (**trig**) *and* $(\mathbf{cong})_o$ *if and only if* $P \to P'$ *using the rules* (**com**)*,* (**trig**) *and* (**cong**)*.*

$$(\mathbf{KCong})\frac{P \equiv Q \qquad Q \to Q' \qquad Q' \equiv P'}{P \to P'} \qquad (\mathbf{KComm})\frac{}{\mathbf{E}[a(x).P_1 \mid \overline{a}\langle v\rangle.P_2] \to \mathbf{E}[P_1\{v/x\} \mid P_2]}$$

$$(\mathbf{KTrig})\frac{}{\begin{array}{r}\mathbf{E}[!a_1(x_1)^{\mathtt{free}}.a_2(x_2)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1 \mid \overline{a_1}\langle v\rangle.P_2] \\ \to \mathbf{E}[!a_1(x_1)^{\mathtt{free}}.a_2(x_2)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1 \mid !a_1(x_1)^{\mathtt{ok}}.((a_2(x_2)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1)\{v/x_1\}) \mid P_2]\end{array}}$$

$$(\mathbf{KProg})\frac{}{\begin{array}{r}\mathbf{E}[!a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{free}}.a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1 \mid \overline{a_i}\langle v\rangle.P_2] \\ \to \mathbf{E}[!a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{ok}}.((a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1)\{v/x_i\}) \mid P_2]\end{array}}$$

$$(\mathbf{KFire})\frac{}{\mathbf{E}[!a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_n(x_n)^{\mathtt{free}}.P_1 \mid \overline{a_n}\langle v\rangle.P_2] \to \mathbf{E}[P_1\{v/x_n\} \mid \overline{a_n}\langle v\rangle.P_2]}$$

Figure 3.3: Semantics for annotated calculus

Soundness proof

As stated above, termination is obtained through an auxiliary calculus, which is basically an annotated version of the π-calculus. Annotations are used in order to keep track of partially consumed input sequences. For instance, when $!a.b.P \mid \overline{a} \to !a.b.P \mid b.P$, instead of writing $b.P$ we write $!a^{\mathtt{ok}}.b.P$, which is, behaviourally, the same process, but which allows us to remember that an output on a has been consumed. Thus when $!a^{\mathtt{ok}}.b.P$ reduces to P, we will be able to compare the weight of P with the levels of both b and a. This procedure will allow us to state precisely when the measure decreases (when the last input of a sequence is consumed), by remembering every output process consumed when new available outputs are released by a reduction.

More precisely, when we trigger a replicated input sequence $!a_1(x_1).a_2(x_2).\ldots.a_n(x_n).P$, the whole input sequence appears in the spawned process and stays until the sequence is entirely consumed and P is freed. To make explicit how deep the input sequence is consumed, we use the annotations `free` and `ok`. Intuitively, $!a_1(x_1)^{\mathtt{ok}}.\ldots.a_i(x_i)^{\mathtt{ok}}.a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n).P$ means that the i first prefixes of the sequence are consumed, thus this process offers interaction on a_{i+1} (and no interaction on a_1).

Here is the syntax for the annotated calculus:

$$P ::= \mathbf{0} \mid P \mid P \mid a(x).P \mid (\nu a)\ P \mid !a_1(x_1)^{l_1}.\ldots.a_n(x_n)^{l_n}.P \mid \overline{a}\langle v\rangle.P$$

where the l_is are either `ok` or `free`. Moreover, we only consider well-annotated input sequences, which are such that there exists $1 \leq i < n$ s.t. $l_j = \mathtt{ok}$ for $j < i$ and $l_j = \mathtt{free}$ for $i \geq j \leq n$. This reflects the fact that, at a given time, only an initial segment of an input sequence can be consumed.

The definition of evaluation contexts is the same as the one for the unannotated calculus:

$$\mathbf{E} ::= [\,] \mid (\boldsymbol{\nu} a)\ \mathbf{E} \mid (\mathbf{E} \mid P)$$

We present the semantics for the annotated calculus in Figure 3.3. Notice that the actual "replication" of a process is done when an input sequence containing only the annotation `free` is triggered (rule (**KTrig**)). We can consider such input sequences as actual replications whereas partially consumed input sequences are not "true replications", from the behavioural point of view (but are considered as such for typing purposes).

Substitutions are applied to input sequences at each reduction step. This can actually modify the input sequence itself, for instance $!a^{\mathtt{ok}}.c(x)^{\mathtt{free}}.x^{\mathtt{free}}.\overline{a} \mid \overline{c}\langle a\rangle$ reduces to $!a^{\mathtt{ok}}.c(x)^{\mathtt{ok}}.a^{\mathtt{free}}.\overline{a}$.

In the following, we consider only processes whose guarded input sequences (an input sequence is guarded in P if it appears inside a subprocess $a(x).P'$, $\overline{a}\langle v\rangle.P'$ or $a_1(x_1)^{l_1} \ldots a_n(x_n)^{l_n}.P'$) are labelled only with `free`.

$$(\mathbf{KNil})\frac{}{\Gamma \vdash^{\kappa,a} \mathbf{0} : \emptyset} \qquad (\mathbf{KPar})\frac{\Gamma \vdash^{\kappa,a} P_1 : M_1 \qquad \Gamma \vdash^{\kappa,a} P_2 : M_2}{\Gamma \vdash^{\kappa,a} P_1 \mid P_2 : M_1 \uplus M_2}$$

$$(\mathbf{KRes})\frac{\Gamma \vdash^{\kappa,a} P : M \qquad \Gamma(a) = \sharp^k T}{\Gamma \vdash^{\kappa,a} (\nu a)\ P : M} \qquad (\mathbf{KIn})\frac{\Gamma \vdash^{\kappa,a} P : M \qquad \Gamma(a) = \sharp^k T \qquad \Gamma(x) = T}{\Gamma \vdash^{\kappa,a} a(x).P : M}$$

$$(\mathbf{KOut})\frac{\Gamma \vdash^{\kappa,a} P : M \qquad \Gamma(a) = \sharp^k T \qquad \Gamma(v) = T}{\Gamma \vdash^{\kappa,a} \overline{a}\langle v\rangle.P : M \uplus \{k\}}$$

$$(\mathbf{KRep})\frac{\Gamma \vdash^{\kappa,a} P : M \qquad \forall i, \Gamma(a_i) = \sharp^{k_i} T_i \wedge \Gamma(x_i) = T_i \qquad M <_{\text{mul}} \{k_1, \ldots, k_n\}}{\Gamma \vdash^{\kappa,a} !a_1(x_1)^{l_1}.\ldots.a_n(x_n)^{l_n}.P : \emptyset}$$

Figure 3.4: Typing rules for the annotated calculus

This property is trivially preserved by reduction, as reduction cannot change the annotation of guarded input sequences.

Typing judgments are written $\Gamma \vdash^{\kappa,a} P : M$ where Γ is a typing context, P a process and its weight M a multiset of integers . Typing rules for the annotated calculus are presented in Figure 3.4. They are basically the same rules as the ones for the unannotated calculus. Notice that the typing rule for replicated input sequences is independent from the annotations of the sequence.

First, we prove that the annotated calculus simulates the unannotated one. What we use more precisely is that an unannotated diverging process can be annotated in such a way that we get a diverging process. We begin by defining a way to go from annotated processes to unannotated ones and conversely.

Definition 3.2.3 (Annotation removal)

We inductively define the annotation removal *as a mapping from annotated processes into unannotated ones (we suppose the input sequences to be maximal, as stated above):*

$$\mathbf{Rem}(\mathbf{0}) = \mathbf{0} \qquad \mathbf{Rem}(P_1 \mid P_2) = (\mathbf{Rem}(P_1) \mid \mathbf{Rem}(P_2)) \qquad \mathbf{Rem}((\nu a)\ P) = (\boldsymbol{\nu} a)\ \mathbf{Rem}(P)$$

$$\mathbf{Rem}(\overline{a}\langle v\rangle.P) = \overline{a}\langle v\rangle.\mathbf{Rem}(P) \qquad \mathbf{Rem}(a(x).P) = a(x).\mathbf{Rem}(P)$$

$$\mathbf{Rem}(!a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P) = a_i(x_i).\ldots.a_n(x_n).\mathbf{Rem}(P)$$

$$\mathbf{Rem}(!a_1(x_1)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P) = !a_1(x_1).\ldots.a_n(x_n).\mathbf{Rem}(P)$$

Corresponding to what we suggested above, replicated input sequences annotated with `free` only are mapped to actual replications when annotations are removed, and partially consumed replicated input sequences are mapped to unreplicated processes. In the latter case, the annotation removal forgets the input prefixes labelled by `ok`.

Moreover, the semantics is defined such a way that if the guarded input sequences of the process involved in a reduction are annotated by `free` (as explained above), a name x_i of a process

$$a_1(x_1)^{\mathtt{ok}} \ldots a_i(x_i)^{\mathtt{ok}}.a_{i+1}(x_{i+1})^{\mathtt{free}}.a_n(x_n)^{\mathtt{free}}.P$$

does not appear in P. Indeed, when a input prefix $a_i(x_i)$ is labelled by `ok`, it means that the name x_i has been instantiated. Thus, when performing the annotation removal on a partially consumed input sequences, no name is "freed" by the operation.

Definition 3.2.4 (Annotating processes)

We use $\texttt{free}(P)$ *to denote the process obtained by adding the annotation* `free` *to each replicated input sequence in* P *(we suppose the input sequences to be maximal, as stated above):*

$$\texttt{free}(\mathbf{0}) = \mathbf{0} \qquad \texttt{free}(P_1 \mid P_2) = \texttt{free}(P_1) \mid \texttt{free}(P_2) \qquad \texttt{free}((\nu a)\ P) = (\nu a)\ \texttt{free}(P)$$

$$\texttt{free}(\overline{a}\langle v\rangle.P) = \overline{a}\langle v\rangle.\texttt{free}(P) \qquad \texttt{free}(a(x).P) = a(x).\texttt{free}(P)$$

$$\texttt{free}(!a_1(x_1).\ldots.a_n(x_n).P) = !a_1(x_1)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.\texttt{free}(P)$$

Notice that, clearly, $\mathbf{Rem}(\texttt{free}(P)) = P$. Again, using a semantics with evaluation contexts forces us to state a fact relating evaluation contexts and annotation removal.

Remark 3.2.5 *We did not define the typability of the annotated calculus from the typability of the unannotated one. Actually, defining* $\Gamma \vdash^{\kappa,a} P : M$ *as* $\Gamma \vdash^{\kappa} \texttt{free}(P) : M$ *would raise presentation problems, as we want to remember the domination constraint (the* $<_{\text{mul}}$ *comparison) used to type an input sequence when the last input of this sequence is consumed.*

Fact 3.2.6 (Relating annotations and evaluation contexts)
If $P = \mathbf{E}[P']$ *and* $P = \mathbf{Rem}(Q)$*, then there exists* $\mathbf{E}_1$ *s.t.* $Q = \mathbf{E}_1[Q']$ *and* $P' = \mathbf{Rem}(Q')$*.*

Proof.
Easily done by structural induction over $\mathbf{E}$*.*

□

Here is the first result that justifies the use of this auxiliary calculus. The following lemma states that the annotation removal induces a simulation between the unannotated process and the annotated one. Essentially, this lemma states that the definition of the semantics for annotated processes is sound.

Lemma 3.2.7 (Annotated calculus - Simulation)
Define $P \leq Q$ *as* $P = \mathbf{Rem}(Q)$*. Then* $\leq$ *is a simulation, i.e. if* $P \leq Q$ *and* $P \to P'$*, there exists* Q' *s.t.* $Q \to Q'$ *and* $P' \leq Q'$*.*

Proof.
By induction over the derivation of $P \to P'$*,*

- *Case* (**Cong**)*. Note that congruence does not longer hold under prefixes as stated in Remark 3.2.2. One can prove that the following holds; if* $P \cong P^{(1)}$*,* $P^{(1)} \to P^{(2)}$ *and* $P^{(2)} \cong P'$ *and* $P =\leq Q$*, then there exist* $Q^{(1)} \equiv Q$*,* $Q^{(2)}$ *s.t.* $Q^{(1)} \to Q^{(2)}$ *and* $Q' \equiv Q^{(2)}$ *s.t.* $P' = \mathbf{Rem}(Q')$*. The process* $Q^{(1)}$ *is obtained by mimicking the structural congruence axioms used to get* $P^{(1)}$ *from* P*. For instance if* $P = P_1 \mid P_2$ *and* $P^{(1)} = P_2 \mid P_1$*, then by looking at Definition 3.2.3, we derive that* $Q = a_1(x_1)^{l_1}.\ldots.a_n(x_n)^{l_n}.P_1 \mid b_1(y_1)^{b_1}.\ldots.b_m(y_m)^{l'_m}.P_2$*, for some possibly null integers* n *and* m*; actually,* n *(resp.* m*) is non-null only if* $P_1 = a(x).P'_1$ *(resp.* $P_2 = b(y).P'_2$*). We obtain* $Q^{(2)}$ *with the induction hypothesis, and* Q' *by the same procedure.*

- *Case* (**Trig**)*. We have* $P = \mathbf{E}[!a_1(x_1).P_1 \mid \overline{a_1}\langle v\rangle.P_2]$*. Fact 3.2.6 gives* $\mathbf{E}_1$ *s.t.* $Q = \mathbf{E}_1[Q^1]$ *and* $(!a(x).P_1 \mid \overline{a_1}\langle v\rangle.P_2) = \mathbf{E}_1[Q^1]$*. By looking at Definition 3.2.3, we deduce* $Q^1 = !a_1(x_1)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.Q_1 \mid \overline{a_1}\langle v\rangle.Q_2$ *with* $P_1 = \mathbf{Rem}(a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.Q_1)$ *and* $P_2 = \mathbf{Rem}(Q_2)$*. We use rule* (**KTrig**) *to derive* $\mathbf{E}_1[Q^1] \to \mathbf{E}_1[Q^2] = \mathbf{E}_1[!a_1(x_1)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.Q_1 \mid Q_2 \mid !a_1(x_1)^{\texttt{ok}}.((\ldots.a_n(x_n)^{\texttt{free}}.Q_1)\{v/x_1\})]$*. We conclude by setting* $Q' = \mathbf{E}_1[Q^2]$ *and checking that*

  $$\mathbf{Rem}(Q') = \mathbf{E}[!a_1(x_1).P_1 \mid P_2 \mid \mathbf{Rem}(!a_1(x_1)^{\texttt{ok}}.((\ldots.a_n(x_n)^{\texttt{free}}.Q_1)\{v/x_1\}))]$$

  This result holds as:

  $$\mathbf{Rem}(!a_1(x_1)^{\texttt{ok}}.((\ldots.a_n(x_n)^{\texttt{free}}.Q_1)\{v/x_1\})) = P_1\{v/x_1\}$$

  .

- *Case* (**Comm**). *We have* $P = \mathbf{E}[a_i(x_i).P_1 \mid \overline{a_i}\langle v\rangle.P_2]$. *Fact 3.2.6 gives* $\mathbf{E}_1$ *s.t.* $Q = \mathbf{E}_1[Q^1]$ *and* $(a_i(x_i).P_1 \mid \overline{a_i}\langle v\rangle.P_2) = \mathbf{E}_1[Q^1]$. *By looking at Definition 3.2.3, three cases can occur:*
  - *Either* $Q^1 = a_i(x_i).Q_1 \mid \overline{a_i}\langle v\rangle.Q_2$ *with* $P_1 = \mathbf{Rem}(Q_1)$ *and* $P_2 = \mathbf{Rem}(Q_2)$. *We use rule* (**KComm**) *to get* $Q^2 = Q_1\{v/x\} \mid Q_2$ *and conclude.*
  - *Or* $Q^1 = !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.Q_1 \mid \overline{a_i}\langle v\rangle.Q_2$ *with* $P_1 = \mathbf{Rem}(a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots, a_n(x_n)^{\mathtt{free}}.Q_1)$ *and* $P_2 = \mathbf{Rem}(Q_2)$. *We use rule* (**KProg**) *to get* $\mathbf{E}_1[Q^1] \to Q' = \mathbf{E}_1[!a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{ok}}.((\ldots.a_n(x_n)^{\mathtt{free}}.Q_1)\{v/x_i\}) \mid Q_2]$. *Definition 3.2.3 gives* $\mathbf{Rem}(Q') = P_1\{v/x_i\} \mid P_2$, *we conclude.*
  - *Or* $Q^1 = !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{free}}.Q_1 \mid \overline{a_i}\langle v\rangle.Q_2$ *with* $P_1 = \mathbf{Rem}(Q_1)$ *and* $P_2 = \mathbf{Rem}(Q_2)$. *We use rule* (**KFire**) *to get* $\mathbf{E}_1[Q^1] \to Q' = \mathbf{E}_1[Q_1\{v/x_i\}) \mid Q_2]$. *Definition 3.2.3 gives* $\mathbf{Rem}(Q') = P_1\{v/x_i\} \mid P_2$, *we conclude.*

□

The following fact claims that if an unannotated process is typable, so is its annotated counterpart, according to the typing rules we defined above.

Fact 3.2.8 (Typing stability)
If $\Gamma \vdash^{\kappa} P : N$, *then* $\Gamma \vdash^{\kappa,a} \mathtt{free}(P) : N$.

Proof. *Easily done by induction over the typing judgement* $\Gamma \vdash^{\kappa} P : N$, *the only annotated parts of* $\mathtt{free}(P)$ *are replicated input sequences annotated with* `free`, *which are typed the same way as the unannotated ones.* □

Now, Fact 3.2.8 and Lemma 3.2.7 allow us to prove termination for the annotated typable processes and derive termination for the unannotated typable ones. The reasoning goes as follows: take P an unannotated typed process, by Fact 3.2.8, $\mathtt{free}(P)$ is typable. Yet, Lemma 3.2.7 states that $\mathtt{free}(P)$ simulates P, thus if $\mathtt{free}(P)$ is terminating, then so is P.

We prove for this calculus the standard Subject Reduction property we proved for the system of Section 3.1.

Fact 3.2.9 (Subject Congruence)
If $P \equiv Q$ *then* $\Gamma \vdash^{\kappa,a} P : N$ *iff* $\Gamma \vdash^{\kappa,a} Q : N$.

Proof. *By induction over the derivation of* $P \equiv Q$, *using the associativity, the commutativity, and the neutrality of* $\emptyset$ *for the operator* $\uplus$. □

The statement of the Substitution Lemma is more complicated this time. We have to state that terms are still typable after a well-typed substitution is applied. But we have also to prove that typability is preserved when a well-typed substitution is applied to a partially consumed input sequence. This case is not trivial, as the input sequence itself changes.

Lemma 3.2.10 (Subject Substitution)

1. *If* $\Gamma(x) = \Gamma(v)$ *and* $\Gamma \vdash^{\kappa,a} P : N$, *then* $\Gamma \vdash^{\kappa,a} P\{v/x\} : N$,
2. *if* $\Gamma(x) = \Gamma(v)$ *and* $\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P : N$, *then* $\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.((a_i(x_i)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P)\{v/x\}) : N$.

Proof.
The proof of this lemma is similar to the one of Lemma 3.1.3 and can be found in Appendix A. One has to be careful when treating the case (**KRep**).

□

Again, we need to relate evaluation contexts and typability. This lemma is the counter-part of Lemma 3.1.4.

Lemma 3.2.11 (Context Typing)
If $\Gamma \vdash^{\kappa,a} \mathbf{E}[P] : N$ *then:*

1. $\Gamma \vdash^{\kappa,a} P : N'$ *for some* $N' \leq_{\text{mul}} N$.
2. *For all* P_0 *s.t.* $\Gamma \vdash^{\kappa,a} P_0 : N_0$, $\Gamma \vdash^{\kappa,a} \mathbf{E}[P_0] : N_{(0)}$ *for some* $N_{(0)}$.

Notice that in Case 2, if $N_0 \leq N'$, then $N_{(0)} \leq N$, but this result is not used in the following (we rely on the decreasing of a different measure).

Proof.
The proof of this lemma is similar to the one of Lemma 3.1.4 and can be found in Appendix A. □

As in the previous section, we introduce a measure that decreases at each reduction step. As announced in the beginning of this section, counting levels of available outputs is not sufficient. Thus, we also count the levels of the consumed part of the input sequences, defining the *available resources multiset* of a process. For instance in $!a^{\mathtt{ok}}.b^{\mathtt{free}}.c^{\mathtt{free}}.P \mid \overline{b}$, the available resources multiset is $\{\mathtt{lvl}(a)\} \uplus \{\mathtt{lvl}(b)\}$. This process reduces to $!a^{\mathtt{ok}}.b^{\mathtt{ok}}.c^{\mathtt{free}}.P$ whose available resources multiset is also $\{\mathtt{lvl}(a)\} \uplus \{\mathtt{lvl}(b)\}$. This way, we do not forget the $\mathtt{lvl}(a)$ component of the measure. This measure does not decrease at each reduction, more precisely, it stays the same when a replication is triggered, or when we progress inside an input sequence (rules (**KTrig**) and (**KProg**)), however, in these situations, the actual number of outputs in the process strictly decreases. The definition is constructed for a typed process, as the levels of the subjects of prefixes are taken into account.

Definition 3.2.12 (Available resources) *The multiset of available resources of a* typed *process is inductively defined by:*

$$\mathbf{AvRes}(\mathbf{0}) = \emptyset \qquad \mathbf{AvRes}(P_1 \mid P_2) = \mathbf{AvRes}(P_1) \uplus \mathbf{AvRes}(P_2) \qquad \mathbf{AvRes}((\boldsymbol{\nu} a)\ P) = \mathbf{AvRes}(P)$$

$$\mathbf{AvRes}(\overline{a}\langle v\rangle.P) = \{k\} \uplus \mathbf{AvRes}(P) \text{ if } \Gamma(a) = \sharp^k T \qquad \mathbf{AvRes}(a(x).P) = \mathbf{AvRes}(P)$$

$$\mathbf{AvRes}(!a_1(x_1)^{\mathtt{ok}}.\ldots.a_i(x_i)^{\mathtt{ok}}.a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P) = \{k_1\} \uplus \cdots \uplus \{k_i\} \text{ if } \forall i, \Gamma(a_i) = \sharp^{k_i} T_i$$

The operator $\mathbf{AvRes}()$ *is straightforwardly extended to contexts by setting* $\mathbf{AvRes}([\,]) = \emptyset$.

The following facts respectively relate evaluation contexts and available resources and claim that multisets of available resources are stable by well-typed substitution and structural congruence. Both these facts will be required in order to prove the Subject Reduction Proposition.

Fact 3.2.13 (Context and available resources)
If $\Gamma \vdash^{\kappa,a} \mathbf{E}[P] : N$, *then* $\mathbf{AvRes}(\mathbf{E}[P]) = \mathbf{AvRes}(\mathbf{E}) \uplus \mathbf{AvRes}(P)$.

Proof. *By structural induction over* $\mathbf{E}$*:*

- *Case* $[\,]$. *Then* $\mathbf{AvRes}(\mathbf{E}[P]) = \mathbf{AvRes}(P) = \mathbf{AvRes}(P) \uplus \mathbf{AvRes}([\,])$.
- *Case* $(\nu a)\,\mathbf{E}_2$. *Then* $\mathbf{AvRes}(\mathbf{E}[P]) = \mathbf{AvRes}((\nu a)\,\mathbf{E}_2[P])$ *which is, by Definition 3.1.6,* $\mathbf{AvRes}(\mathbf{E}_2[P])$. *We use the induction hypothesis to get* $\mathbf{AvRes}(\mathbf{E}_2[P]) = \mathbf{AvRes}(\mathbf{E}_2) \uplus \mathbf{AvRes}(P)$. *As, by Definition 3.1.6,* $\mathbf{AvRes}(\mathbf{E}) = \mathbf{AvRes}(\mathbf{E}_2)$, *we conclude.*
- *Case* $\mathbf{E}_2 \mid P_1$. *Then* $\mathbf{AvRes}(\mathbf{E}[P]) = \mathbf{AvRes}(\mathbf{E}_2[P] \mid P_1)$ *which is, by Definition 3.1.6,* $\mathbf{AvRes}(\mathbf{E}_2[P]) \uplus \mathbf{AvRes}(P_1)$. *We use the induction hypothesis, to get* $\mathbf{AvRes}(\mathbf{E}_2[P]) = \mathbf{AvRes}(\mathbf{E}_2) \uplus \mathbf{AvRes}(P)$. *As, by Definition 3.1.6* $\mathbf{AvRes}(\mathbf{E}) = \mathbf{AvRes}(\mathbf{E}_2) \uplus \mathbf{AvRes}(P_1)$*; we conclude.*

□

Fact 3.2.14 (Substitution and available resources)
If $\Gamma \vdash^{\kappa,a} P : N$ *and* $\Gamma(v) = \Gamma(x)$*, then* $\mathbf{AvRes}(P) = \mathbf{AvRes}(P\{v/x\})$.

Proof. *First, Lemma 3.2.10 allows us to write* $\mathbf{AvRes}(P\{v/x\})$*. We proceed by induction on the typing judgement, the interesting case being* $P = \overline{a}\langle v\rangle.P_1$*. Suppose* $\Gamma(a) = \sharp^k T$*. Either* $a \neq x$ *and* $\mathbf{AvRes}(P\{v/x\}) = \{k\} \uplus \mathbf{AvRes}(P_1\{v/x\})$*, or* $a = x, \Gamma(v) = \Gamma(x) = \Gamma(a) = \sharp^k T$ *and* $\mathbf{AvRes}(P\{v/x\}) = \{k\} \uplus \mathbf{AvRes}(P_1\{v/x\})$*, in both cases we use the induction hypothesis to conclude.* □

Fact 3.2.15 (Structural congruence and available resources)
If $P \equiv Q$*,* $\Gamma \vdash P : n$ *and* $\Gamma \vdash Q : m$*, then* $\mathbf{AvRes}(P) = \mathbf{AvRes}(Q)$.

Proof. *By induction over the derivation of* $P \equiv Q$*, using the commutativity, the associativity and the neutrality of* $\emptyset$ *for* $\uplus$*.* □

Like in the previous section, we have to relate the weight system and the measure we defined. Here, we state that if a process is annotated only with annotations `free`, (remember this is the case of processes guarded by a prefix), then its weight and its available resources multiset coincide.

Lemma 3.2.16 (Available Resource domination)
If P *contains only* `free` *in annotations and* $\Gamma \vdash^{\kappa,a} P : N$*, then* $\mathbf{AvRes}(P) = N$.

Proof. *By induction over the typing judgement:*

- *Case* (**KNil**) *is trivial.*
- *Cases* (**KRes**) *and* (**In**) *are done using the induction hypothesis.*
- *Case* (**KPar**) *is done using the induction hypothesis and the compatibility of* $\uplus$ *with* $=$*.*
- *Case* (**KOut**)*. We have* $P = \overline{a}\langle v\rangle.P_1$*. The induction hypothesis gives* $\mathbf{AvRes}(P_1) = N_1$*. Definition 3.2.12 and typing rule* (**KOut**) *allows us to conclude.*
- *Case* (**KRep**)*. We have* $P = !a_1(x_1)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1$ *as we supposed annotations in* P *contain only* `free`*. Definition 3.2.12 gives* $\mathbf{AvRes}(P) = \emptyset$*. We conclude.*

□

As written above, when a reduction using rules (**Ktrig**) or (**KProg**) is performed, the multiset of available resources stays the same, but the actual number of outputs decreased. Thus, we will use $\mathbf{Os}(\cdot)$, as defined in the previous section, as a second component for our measure.

Definition 3.2.17 (Available outputs)
We extend the definition of available outputs (Definition 3.1.6) to the annotated calculus with the following rule:

$$\mathbf{Os}(!a_1(x_1)^{l_1}.\ldots.a_n(x_n)^{l_n}.P) = \emptyset$$

It is easy to check that Lemmas 3.1.8 and 3.1.11 and Facts 3.1.9, 3.1.10 still hold.

The following proposition is the crux of the proof, as it proves that a well-founded measure decreases strictly at each reduction. As a decreasing measure, we use a lexicographical ordering composed of a multiset comparison of $\mathbf{AvRes}()$ and a multiset comparison of $\mathbf{Os}()$. When a reduction with rule (**KFire**) is performed, the continuation of the replicated input sequence will be taken into account in the reduced process by $\mathbf{AvRes}(\cdot)$. However the typing rule (**KRep**) ensures that the multiset of available resources associated to this continuation is strictly smaller than the multiset sum of the singleton composed of the levels of the names composing the input sequence.

We give here a few intuitions to understand the proof. We proceed by discussing which reduction is used. The case (**KCong**) is treated using the facts we proved above, stating that measures and typability are stable by structural congruence.

If the rule (**KComm**) is used, then the measure decreases directly from P to P', as an output of P is consumed in P' and the outputs guarded by the unreplicated input prefix are already taken into account when computing the measure of P.

If the rule (**KTrig**) is used, it means that a replication whose annotations contain only `free` (which is mapped, using `free`() to an actual replication) is triggered by an output. A process is spawned, containing a partially eaten input sequence (only the first annotation is `ok`). We prove that the measure associated to this process is equal to the level of the output consumed to trigger the replication. Then we show that the number of outputs in the process P' has decreased.

When the rule (**KProg**) is used, it means that we progress by consuming further a partially consumed input sequence. We prove that the measure of the process containing the partially consumed input sequence is raised, between P and P' by the level of the output consumed. As previously, we also show that the number of outputs decreases.

Invoking the rule (**KFire**) is the crucial point: a subprocess of P guarded by a nearly-consumed input sequence (only the last input prefix is annotated with `free`) is released. We show that our typing rules ensure that the measure associated to this released process is strictly smaller than the one we associated to guarded prefix, augmented by the level of the output consumed by the reduction.

Proposition 3.2.18 (Subject reduction)
If $\Gamma \vdash^{\kappa,a} P : N$ *and* $P \to P'$*, then*

1. $\Gamma \vdash^{\kappa,a} P' : N'$ *for some* N'*.*
2. $(\mathbf{AvRes}(P'), \mathbf{Os}(P')) <_{lex} (\mathbf{AvRes}(P), \mathbf{Os}(P))$*.*

Notice that here, the weight N of a process can grow after a reduction step.

Proof. *By induction on the reduction derivation.*

- *Case* (**KCong**)*. We use the induction hypothesis and Facts 3.2.9, 3.1.9 and 3.2.15 to conclude.*
- *Case* (**KComm**)*. We have* $P = \mathbf{E}[a(x).P_1 \mid \overline{a}\langle v\rangle.P_2]$ *and* $P' = \mathbf{E}[P_1\{v/x\} \mid P_2]$*. From* $\Gamma \vdash^{\kappa,a} P : N$ *and Lemma 3.2.11 we derive* $\Gamma \vdash^{\kappa,a} a(x).P_1 \mid \overline{a}\langle v\rangle.P_2 : M$ *for some* $M \leq N$*. Then, we derive* $\Gamma \vdash^{\kappa,a} P_1 : M_1$*,* $\Gamma \vdash^{\kappa,a} P_2 : M_2$*,* $\Gamma(a) = \sharp^k T$ *and* $M = M_1 \uplus M_2 \uplus \{k\}$*. Fact 3.2.13 gives* $\mathbf{AvRes}(P) = \mathbf{AvRes}(\mathbf{E}) \uplus \mathbf{AvRes}(P_1) \uplus \mathbf{AvRes}(P_2) \uplus \{k\}$*. Lemma 3.2.10 gives* $\Gamma \vdash^{\kappa,a} P_1\{v/x\} : M_1$*. Using* (**KPar**) *we derive* $\Gamma \vdash^{\kappa,a} P_1\{v/x\} \mid P_2 : M_1 \uplus M_2$ *and Lemma 3.2.11 gives* $\Gamma \vdash^{\kappa,a} P' : N'$ *for some* N'*. Fact 3.2.13 gives* $\mathbf{AvRes}(P') = \mathbf{AvRes}(\mathbf{E}) \uplus \mathbf{AvRes}(P_1\{v/x\}) \uplus \mathbf{AvRes}(P_2)$*. Fact 3.2.14 gives* $\mathbf{AvRes}(P_1\{v/x\}) = \mathbf{AvRes}(P_1)$*. We conclude, as* $\mathbf{AvRes}(P) >_{\text{mul}} \mathbf{AvRes}(P')$*.*
- *Case* (**KTrig**)*. We have* $P = \mathbf{E}[!a_1(x_1)^{\texttt{free}}.a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1 \mid \overline{a_1}\langle v\rangle.P_2]$ *and* $P' = \mathbf{E}[!a_1(x_1)^{\texttt{free}}.a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1 \mid !a_1(x_1)^{\texttt{ok}}.((a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1)\{v/x_1\}) \mid P_2]$*. From* $\Gamma \vdash^{\kappa,a} P : N$ *and Lemma 3.2.11, we derive* $\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\texttt{free}}.a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1 \mid \overline{a_1}\langle v\rangle.P_2 :$ *M for some* $M \leq N$*. Then, we derive* $\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\texttt{free}}.a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1 : \emptyset$*,* $\Gamma \vdash^{\kappa,a} P_1 : M_1$*,* $\Gamma \vdash^{\kappa,a} P_2 : M_2$*, for all* i*,* $\Gamma(a_i) = \sharp^{k_i} T_i$*,* $\biguplus_i \{k_i\} >_{\text{mul}} M_1$ *and* $M = M_2 \uplus \{k_1\}$*. Fact 3.2.13 gives* $\mathbf{AvRes}(P) = \mathbf{AvRes}(\mathbf{E}) \uplus \mathbf{AvRes}(P_2) \uplus \{k_1\}$*. Lemma 3.1.8 gives* $\mathbf{Os}(P) = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_2) \uplus \{k_1\}$*. Lemma 3.2.10 gives* $\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\texttt{ok}}.((a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1)\{v/x_1\}) : \emptyset$*. Using* (**KPar**) *twice we derive* $\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\texttt{free}}.a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1 \mid !a_1(x_1)^{\texttt{ok}}.((a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1)\{v/x_1\}) \mid P_2 :$ $\emptyset \uplus M_2$ *and Lemma 3.2.11 gives* $\Gamma \vdash^{\kappa,a} P' : N'$ *for some* N'*. Fact 3.2.13 gives* $\mathbf{AvRes}(P') = \mathbf{AvRes}(\mathbf{E}) \uplus \mathbf{AvRes}(!a_1(x_1)^{\texttt{ok}}.((a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1)\{v/x_1\})) \uplus \mathbf{AvRes}(P_2)$*. From Definition 3.2.12 and Fact 3.2.14 gives* $\mathbf{AvRes}(!a_1(x_1)^{\texttt{ok}}.((a_2(x_2)^{\texttt{free}}.\ldots.a_n(x_n)^{\texttt{free}}.P_1)\{v/x_1\})) = \{k_1\}$*. Lemma 3.1.8 and Fact 3.1.10 we get* $\mathbf{Os}(P') = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_2)$*. We conclude, as* $\mathbf{AvRes}(P) = \mathbf{AvRes}(P')$ *and* $\mathbf{Os}(P') <_{\text{mul}} \mathbf{Os}(P)$*.*

- *Case* (**KProg**). *We have*

$$P = \mathbf{E}[!a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{free}}.a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1 \mid \overline{a_i}\langle v\rangle.P_2]$$

and

$$P' = \mathbf{E}[!a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{ok}}.((a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1)\{v/x_i\}) \mid P_2]$$

From $\Gamma \vdash^{\kappa,a} P : N$ *and Lemma 3.2.11, we derive*

$$\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{free}}.a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1 \mid \overline{a_i}\langle v\rangle.P_2 : M$$

for some $M \leq N$. *Then, we derive*

$$\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{free}}.a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1 : \emptyset,$$

$\Gamma \vdash^{\kappa,a} P_1 : M_1$, $\Gamma \vdash^{\kappa,a} P_2 : M_2$, *for all* i, $\Gamma(a_i) = \sharp^{k_i} T_i$, $\biguplus_i\{k_i\} >_{\mathrm{mul}} M_1$ *and* $M = M_2 \uplus \{k_i\}$. *Fact 3.2.13 gives* $\mathbf{AvRes}(P) = \mathbf{AvRes}(\mathbf{E}) \uplus \{k_i\} \uplus \mathbf{AvRes}(P_2) \uplus \{k_1\} \uplus \cdots \uplus \{k_{i-1}\}$. *Lemma 3.1.8 gives* $\mathbf{Os}(P) = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_2) \uplus \{k_i\}$. *Lemma 3.2.10 gives*

$$\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{ok}}.((a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1)\{v/x_i\}) : \emptyset.$$

Using (**KPar**) *we derive*

$$\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{ok}}.((a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1)\{v/x_i\}) \mid P_2 : \emptyset \uplus M_2$$

and Lemma 3.2.11 gives $\Gamma \vdash^{\kappa,a} P' : N'$ *for some* N'. *Fact 3.2.13 gives*

$$\mathbf{AvRes}(P') = \mathbf{AvRes}(\mathbf{E}) \uplus \mathbf{AvRes}(!a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{ok}}.((a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1)\{v/x_i\})) \uplus \mathbf{AvRes}(P_2).$$

From Definition 3.2.12 and Fact 3.2.14 gives

$$\mathbf{AvRes}(!a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{ok}}.((a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1)\{v/x_i\})) = \{k_1\} \uplus \cdots \uplus \{k_{i-1}\} \uplus \{k_i\}.$$

Lemma 3.1.8 and Fact 3.1.10 we get $\mathbf{Os}(P') = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_2)$. *We conclude, as* $\mathbf{AvRes}(P) = \mathbf{AvRes}(P')$ *and* $\mathbf{Os}(P') <_{\mathrm{mul}} \mathbf{Os}(P)$.

- *Case* (**KFire**). *We have*

$$P = \mathbf{E}[!a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_n(x_n)^{\mathtt{free}}.P_1 \mid \overline{a_n}\langle v\rangle.P_2]$$

and

$$P' = \mathbf{E}[P_1\{v/x_n\} \mid P_2].$$

From $\Gamma \vdash^{\kappa,a} P : N$ *and Lemma 3.2.11, we derive* $\Gamma \vdash^{\kappa,a} !a_1^{\mathtt{ok}}(x_1).\ldots.a_{i-1}^{\mathtt{ok}}(x_{i-1}).a_n^{\mathtt{free}}(x_n).P_1 \mid \overline{a_n}\langle v\rangle.P_2 : M$ *for some* $M \leq N$. *Then, we derive* $\Gamma \vdash^{\kappa,a} !a_1^{\mathtt{ok}}(x_1).\ldots.a_{i-1}^{\mathtt{ok}}(x_{i-1}).a_n^{\mathtt{free}}(x_n).P_1 : \emptyset$, $\Gamma \vdash^{\kappa,a} P_1 : M_1$, $\Gamma \vdash^{\kappa,a} P_2 : M_2$, *for all* i, $\Gamma(a_i) = \sharp^{k_i} T_i$, $\biguplus_i\{k_i\} >_{\mathrm{mul}} M_1$ *and* $M = M_2 \uplus \{k_n\}$. *Fact 3.2.13 gives* $\mathbf{AvRes}(P) = \mathbf{AvRes}(\mathbf{E}) \uplus \{k_n\} \uplus \mathbf{AvRes}(P_2) \uplus \{k_1\} \uplus \cdots \uplus \{k_{n-1}\}$. *Using* (**KPar**) *we derive* $\Gamma \vdash^{\kappa,a} P_1\{v/x_n\} \mid P_2 : M_1 \uplus M_2$ *and Lemma 3.2.11 gives* $\Gamma \vdash^{\kappa,a} P' : N'$ *for some* N'. *Fact 3.2.13 gives* $\mathbf{AvRes}(P') = \mathbf{AvRes}(\mathbf{E}) \uplus \mathbf{AvRes}(P_1\{v/x_n\}) \uplus \mathbf{AvRes}(P_2)$. *As* P_1 *is guarded by a prefix in* P, *we know that every annotation in* P_1 *is* `free`, *we can apply Lemma 3.2.16 to get* $\mathbf{AvRes}(P_1)\{v/x_n\}) = M_1$. *The condition* $\biguplus_i\{k_i\} >_{\mathrm{mul}} M_1$ *allows us to conclude, as* $\mathbf{AvRes}(P) >_{\mathrm{mul}} \mathbf{AvRes}(P')$.

□

This proposition allows us to derive soundness in a way similar to the one we used in the previous section.

Proposition 3.2.19 (Soundness)
If $\Gamma \vdash^{\kappa} P : N$, then P terminates.

Proof.
Suppose that P diverges. Then, by Lemma 3.2.7, $\texttt{free}(P)$ diverges and by Fact 3.2.8, $\Gamma \vdash^{\kappa,a} \texttt{free}(P) : N$. This means that we have an infinite reduction sequence $(Q_i)_{i\in\mathbb{N}}$ s.t. $Q_i \to Q_{i+1}$ and $Q_0 = \texttt{free}(P)$. Proposition 3.2.18 ensures that each Q_i is typable and that the measure $(\mathbf{AvRes}(Q_i), \mathbf{Os}(Q_i))$ induces a strictly decreasing sequence for the lexicographical composition of two well-founded orders. We use Theorems 2.1.3 and 2.1.4 to derive a contradiction.

□

3.2.2 Further refinements: prefix trades

We very briefly present here an extension of the previous type system which seems natural, although it leads to far more technical proofs, which are omitted here. Consider the following replicated process $R_1 = !a.(b.\overline{c} \mid b)$. From the point of view of divergence, this process is "less dangerous" than $R_2 = !a.b.(\overline{c} \mid b)$ in the sense that for every context $\mathbf{E}$, $\mathbf{E}[R_1]$ diverges only if $\mathbf{E}[R_2]$ diverges. However R_1 is not typable with the previous type system, although R_2 is. The reason is that we do not make a refined analysis of a replicated process beyond the first input sequence.

Our goal is to build a type system able to analyse every possible way of consuming the inputs inside a replicated process and ensuring that every way is safe, from the point of view of termination. The following informal definition presents the set of *trades* a replicated process is able to make, that is, which outputs are made available for each possible way of partially consuming inputs of the replicated process. In the case of the R_1 process above, we would have $\{\{a\} \mapsto \emptyset, \{a,b\} \mapsto \{c\}, \{a,b,b\} \mapsto \{c\}\}$.

For a replication $!a(x).P$, we define $\texttt{Trades}(!a(x).P)$ as the sets of every trade $I \mapsto O$ *where I, O are multisets of names and O is the multiset of all outputs made available by the consumption of the input prefixes of I in $!a(x).P$.* As hinted above, we use here an informal definition. A formal one can be obtained at the cost of greater technical details, by using inductive structures to abstract replications and by inductively defining a way of computing which outputs we get for the input we consumed.

The type system of this section is based on a complete analysis of a replicated process. For each replicated process, we enumerate every way we are able to partially or totally consume this process and prove that whatever way we choose, there is always a decreasing between the input consumed and the outputs made available. For instance, consider the replicated process $!a.b.(b.\overline{a} \mid \overline{d} \mid b.(\overline{d} \mid \overline{d} \mid \overline{d}))$. Its set of trades is $\{\{a\} \mapsto \emptyset, \{a,b\} \mapsto \{d\}, \{a,b,b\} \mapsto \{d,a\}, \{a,b,b\} \mapsto \{d,d,d,d\}, \{a,b,b,b\} \mapsto \{d,d,d,d,a\}\}$. We have to assign levels to names such that every trade presents a decreasing. This is possible by assigning level 2 to a and b and level 1 to d.

We build a new typing rule for replicated process (as usual, if M is a multiset of names,$\texttt{lvl}(M)$ stands for the multiset of levels of the names in M). The remaining rules of this type system are the same as ones we presented in the previous section.

$$(\mathbf{TrRep})\frac{\Gamma \vdash P : m \qquad \Gamma(a) = \sharp^k T \qquad \Gamma(x) = T \qquad \forall (I \mapsto O) \in \texttt{Trades}(!a(x).P),\ \texttt{lvl}(I) > \texttt{lvl}(O)}{\Gamma \vdash !a(x).P : 0}$$

Soundness of this system would be stated as follows: *If $\Gamma \vdash P : m$, then P terminates.*

We derive termination, although more technical details are required in order to handle the termination proof. Indeed, in the previous system, even if the weight could grow during a reduction step we were able, by grouping together reductions (we performed it using annotations), to exhibit a weight decreasing between two (possibly non-consecutive) steps. Things get more complicated here, as there does not exist such steps (corresponding to the consuming of the last input of the sequence). As a consequence, we know that the

weight decreases, but the interleavings in the consumption of different replications prevent us from stating, in an infinite reduction sequence $(P_i)_i$ from a typable process P_0, that between two explicit steps i and $j > i$, the weight of P_j is smaller than the weight of P_i. One way to proceed to obtain soundness is to associate to each trade of each replication a function $\sigma : x \mapsto \mathtt{lvl}(I) - \mathtt{lvl}(O)$ and prove that the weight of a process P_i of the infinite reduction sequence is less than the weight of P_0 to which are applied several functions of this type. More precisely, one function is applied for each replication which has been triggered during the sequence. Typing rules ensure that these functions let their arguments decrease (i.e. $x - \mathtt{lvl}(I) + \mathtt{lvl}(O)$ is always less than x), and we conclude using the well-foundedness of the same multiset ordering as in Section 3.2.1.

3.3 Introducing a partial order

3.3.1 Partial orders between names

The purpose of this section is to give a more clear presentation of the fourth type system from [DS06]. This system allows the decreasing measure to be more complex and to accommodate comparisons based on a partial order between names, which is more powerful than comparisons based solely on the types of the names (as it allows us to compare two names of the same type).

Such a feature is useful, because it allows us to compare elements which have the same nature (for instance several nodes in a list structure), and is required to type any inductively defined well-founded data structure (lists, trees).

For instance, the crux of encoding lists data-structures in π is accommodating a constructor of the form $C = !p(a,b).a.(\overline{b} \mid \overline{p}\langle a, b\rangle)$. Then we can initiate a list by putting in parallel of C outputs creating nodes, for instance $C' = C \mid \overline{p}\langle a_1, a_2\rangle \mid \overline{p}\langle a_2, a_3\rangle \mid \overline{p}\langle a_3, nil\rangle$ models a list composed of three elements a_1, a_2, a_3. Notice that C' can perform three reductions to $C \mid a_1.(\overline{a_2} \mid \overline{p}\langle a_1, a_2\rangle) \mid a_2.(\overline{a_3} \mid \overline{p}\langle a_2, a_3\rangle) \mid a_3.(\overline{nil} \mid \overline{p}\langle a_3, nil\rangle)$. When a request $\overline{a_1}$ is sent to the head of the list, it is passed to the next node by the output on a_2 and the constructor C creates a new instance of node a_1 by consuming the output $\overline{p}\langle a_1, a_2\rangle$. The termination of such a program is not trivial, as the constructor is able to call itself. Moreover, the fact that the a_i's are carried both in the first and in the second component of p forces our type systems to give them the same level. Thus the outputs present in the continuation of the input sequence $!p(a,b).a$ have the same level than p and a. As a consequence, such processes cannot be typed by the previous type systems. One can remark that what makes this process terminating is that the a_i's are ordered following a well-founded relation. For instance, this would not be the case for the diverging process $C \mid \overline{p}\langle a_1, a_2\rangle \mid \overline{p}\langle a_2, a_3\rangle \mid \overline{p}\langle a_3, a_1\rangle \mid \overline{a_1}$. The type system we present here accommodates the use of an external partial order on names found in a process, in order to recognise as terminating such processes.

We first present some technical notions on partial orders and multisets that will be required later:

Definition 3.3.1 (Partial orders)

In the following we will use the symbol $\mathcal{R}$ to denote partial orders on names and $\mathtt{R}$ to denote partial orders on natural numbers. A partial order is an antisymmetric, antireflexive and transitive relation. It is presented as a set of pairs, the transitive closure being implicit: $\mathcal{R} = \{(a,b),(b,c)\}$ means that $a\mathcal{R}b$, $b\mathcal{R}c$, and $a\mathcal{R}c$. The domain of a partial order is the set of names (or natural numbers) it relates.

$\mathcal{R} - \{a\}$ stands for the relation $\mathcal{R}$ where every pair containing the element a is removed. It follows immediately that $\mathcal{R} - \{a\}$ is a partial order.

We write $\mathtt{R}\lfloor\widetilde{a}\rfloor$ to denote the ordering on the elements of $\widetilde{a}$ obtained by setting $a_i \ \mathtt{R}\lfloor\widetilde{a}\rfloor \ a_j$ iff $i\mathtt{R}j$ (for all a_i, a_j in $\widetilde{a}$).

We write $\mathcal{R} = \mathcal{R}_1 \uplus \mathcal{R}_2$ to denote that $\mathcal{R}$ is a relation composed of two relations whose domains are disjoint. Notice that if $\mathcal{R}_1$ and $\mathcal{R}_2$ are partial orders, then so is $\mathcal{R}$.

We write $\mathcal{R}_1 \cup \mathcal{R}_2$ to denote the transitive closure of the union of the two relations $\mathcal{R}_1$ and $\mathcal{R}_2$. Notice that if $\mathcal{R}_1$ and $\mathcal{R}_2$ are partial orders, $\mathcal{R}_1 \cup \mathcal{R}_2$ is not necessarily a partial order. For instance, take $\mathcal{R}_1 = \{(a,b)\}$ and $\mathcal{R}_2 = \{(b,a)\}$.

Types for names change from the previous sections. Indeed, the information about the partial order has to be stored somewhere by the type systems when ordered names are sent or received. For instance if x and y necessarily have the same level, $D = !a(x,y).x.(\overline{y} \mid \overline{a}\langle x,y\rangle)$ could be typed provided $x\mathcal{R}y$. But we have to prevent the diverging process $D \mid \overline{a}\langle c,d\rangle \mid \overline{a}\langle d,c\rangle \mid \overline{c}$ from being typable. We use the type of a to perform this. The type of a notifies that its first argument has to be greater than the second one. Here is the new syntax for types for names:

$$T ::= \sharp^k_{\mathtt{R}}\, \widetilde{T} \mid \mathbb{1}$$

$\mathtt{R}$ being a partial order on the elements of $\{1,\ldots,n\}$ where n is the arity of T (or the number of element in $\widetilde{T}$).

Definition 3.3.2 (Composing two multiset orderings)
If $>_1$ and $>_2$ are two comparisons on elements of a set $\mathcal{E}$, $\mathtt{mul}_{(>_1,>_2)}$ denotes the ordering on multisets of elements in $\mathcal{E}$ obtained by composing lexicographically the multiset extension of $>_1$ and the multiset extension of $>_2$.

$\mathtt{mul}_{(>_1,>_2)}$ tries to compare two multisets with a first multiset orderings, and it the two multisets are uncomparable, it uses the second multiset ordering. For instance, suppose that $\mathrm{par}(n) = 1$ if n is odd and $\mathrm{par}(n) = 0$ if n is even and that the relation $>^{\mathrm{par}}$ on integers is defined by $n_1 >^{\mathrm{par}} n_2$ when $\mathrm{par}(n_1) > \mathrm{par}(n_2)$. We write for the multiset extension of $>^{\mathrm{par}}$, written $>^{\mathrm{par}}_{\mathrm{mul}}$. Then we have $\{3,2,2,1\}\mathtt{mul}_{(>^{\mathrm{par}},>)}\{6,4,2,1\}$ because $\mathrm{par}(\{3,2,2,1\}) = \{1,0,0,1\}$ and $\mathrm{par}(\{6,4,2,1\}) = \{0,0,0,1\}$ and thus $\{3,2,2,1\} >^{\mathrm{par}}_{\mathrm{mul}} \{6,4,2,1\}$. We also have $\{6,4,3,1\}\mathtt{mul}_{(>^{\mathrm{par}},>)}\{3,2,2,1\}$ because $\mathrm{par}(\{3,2,2,1\}) = \mathrm{par}(6,4,3,1) = \{1,1,0,0\}$ and $\{6,4,3,1\} >_{\mathrm{mul}} \{3,2,2,1\}$.

In the following, we will use $>_{\mathtt{lvl}}$ to denote the ordering between names, w.r.t. a typing context Γ, defined by $a >_{\mathtt{lvl}} b$ if $\Gamma(a) = \sharp^{k_a}_{\mathcal{R}_a}\, T_a$, $\Gamma(b) = \sharp^{k_b}_{\mathcal{R}_b}\, T_b$, and $k_a > k_b$.

We will use Definition 3.3.2 in the following way: $M_1\mathtt{mul}_{(>_{\mathtt{lvl}},\mathcal{R})}M_2$, meaning that:

- either the multiset of names M_1 *dominates* M_2 *by levels*, which is, the multiset of levels of the names in M_1 is greater for the multiset extension of the standard ordering on natural numbers than the multiset of levels of the names in M_2 (exactly the condition presented in Section 3.2.1),
- or these two multisets (written $\mathtt{lvl}(M_1)$ and $\mathtt{lvl}(M_2)$) are the same but M_1 *dominates* M_2 *according to the partial order* $\mathcal{R}$.

Typing judgements are written as in the previous sections. The only differences are that we use multisets of names as types for processes instead of multisets of natural numbers and that a process is typed w.r.t. a partial order between names $\mathcal{R}$:

$$\Gamma, \mathcal{R} \vdash_{\mathtt{po}} P : M$$

with the type M of a process being a multiset of names and $\mathcal{R}$ being a partial order on free names of P.

We have to say a word about restrictions, as we want diverging processes like the following one to be rejected by our type system:

$$!p(x,y).x.(\overline{y} \mid (\nu c)\ (\overline{p}\langle y,c\rangle)) \mid \overline{p}\langle a,b\rangle$$

that could be typed with $\Gamma(p) = \sharp^0_{\{(1,2)\}}\, \sharp^1\, \mathbb{1} \times \sharp^1\, \mathbb{1}$. However, this process creates an infinite sequence of names c_i, all having the same type as the one of a and b, each name being higher in the partial order than the following one. To sum up, every this process would be typable (and so would be the processes obtained by reducing this process), but the partial order associated to the infinite reduction sequence is not well-founded (it contains an infinite number of names).

Thus, we have to forbid the creation of new names when we use the partial order to compare names having the same level, as it could lead to an infinite number of decreasing steps on an infinite set of names. The safety condition presented in the following ensures that such a problem cannot arise in our type system.

Definition 3.3.3 (Safety condition)

Let P be a process, and M_1, M_2 two multisets of names.

We define the maximum decreasing level *of (M_1, M_2), with respect to a typing context for names in M_1 and M_2, by the maximum level, according to Γ, of an element of the multiset N_1, which is defined by the three following conditions: $M_1 = N \uplus N_1$, $M_2 = N \uplus N_2$ and N is maximal for inclusion.*

The safety condition **safe**(M_1, M_2, P)*, defined with respect to a typing context Γ for names in M_1 and M_2 holds when there is no restriction in P on any level greater or equal to the maximum decreasing level of (M_1, M_2).*

For instance consider the example above with $P = \overline{y} \mid (\nu c)\ (\overline{p}\langle y, c\rangle)$, $M_1 = \{p, x\}$ and $M_2 = \{p, y\}$. Suppose that $\texttt{lvl}(p) = 1$ and $\texttt{lvl}(x) = \texttt{lvl}(y) = \texttt{lvl}(c) = 2$. In this case the maximal submultiset N of M_1 and M_2, as defined in Definition 3.3.3, is $\{p\}$. Thus $N_1 = x$ and the maximum decreasing level is 2. Thus the safety condition is not met, as P contains a restriction on c of level 2. Another example: if $P = (\nu c)P'$, $M_1 = \{a, b, d\}$ and $M_2 = \{a, x, c\}$ and if $\texttt{lvl}(c) = \texttt{lvl}(d) = 2$, $\texttt{lvl}(a) = 1$ and $\texttt{lvl}(b) = \texttt{lvl}(x) = 3$, the maximum decreasing level is 3 and thus the safety condition is met (providing P' does not contain other restriction).

Remark 3.3.4 (Removing the safety condition) *The safety condition we propose here allows us to rely on a well-founded partial order in the soundness proof. One main difference with the previous systems is that new levels cannot be created at run-time, thus the multiset measures induced by levels are trivially well-founded. However, here, new names can be added to the partial order during execution. Such a safety condition is thus mandatory.*

We believe that the condition we propose here is a good compromise between a drastic condition (forbidding creation of names) which would implies a limited expressiveness and a set of ad-hoc *conditions (verifying, for each operator ν, that it is not used in a diverging loop) which would imply too much technical details.*

That is, when we compare M_1 and M_2, there exists a maximum level l on which the comparison takes place (i.e. $\forall L > l$, the subsets $M_1|_L$ (i.e. $\{e \in M_1, \texttt{lvl}(e) = L\}$) and $M_2|_L$ are the same). And we do not want names whose level is greater or equal than this level to be restricted inside the process.

In our case, this safety condition ensures that when a replicated process is spawned by a reduction, if new names are created because of restrictions, the output consumed to create them is strictly greater than their levels. As a consequence, we cannot trade an output at level n for another one at level l, but being smaller for the partial order, and at the same time, creating a new name of level l in order to raise a loop.

The counter-example presented above is directly ruled out, as the comparison used is $x\mathcal{R}y$ and there is in the continuation a restriction (νc) on the same level as y, which is the maximum decreasing level.

3.3.2 Type System with partial order

Figure 3.5 presents the typing rules for our type system. Even if the rule (**PoRep**) is, again, the crux of this type system, rules (**PoOut**) and (**PoIn**) also require some explanations. These rules are used to propagate the partial order. When binding a uple of names $\widetilde{x}$ by input (either replicated or not), we force the partial order used to type the continuation to be separated in two disjoint parts, one for the names in $\widetilde{x}$, another for the other names. That means we prevent partial order comparison between free names and bound names, or between names bound by two different inputs. This is necessary as the only way we have to propagate the partial order is the type of channel names. Moreover, when typing such an input, we force the partial order component on the name in $\widetilde{x}$ to abide the constraints found in the type of the channel binding these names. In the case of the output rule, we check that the partial order typing the process abides to the constraint found in the type of the channel on which the names $\widetilde{v}$ are sent (Condition $\mathcal{R}_a\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R}$). In rule (**PoRes**) we mask in the partial order the occurrences of the name being restricted.

Remark 3.3.5 (How comparisons are applied.) *We discuss here the rule* (**PoRed**)*, it is very similar to the rule presented in Figure 3.2 except that if the multisets of levels are equal, a comparison using the multiset extension of the partial order $\mathcal{R}$ is made. One can write the rule of Figure 3.2 by removing the*

$$(\textbf{PoNil})\frac{}{\Gamma,\mathcal{R}\vdash_{\mathtt{po}}\mathbf{0}:\emptyset}\qquad\qquad(\textbf{PoRes})\frac{\Gamma,\mathcal{R}\vdash_{\mathtt{po}}P:M}{\Gamma,\mathcal{R}-\{c\}\vdash_{\mathtt{po}}(\nu c)\ P:M}$$

$$(\textbf{PoPar})\frac{\Gamma,\mathcal{R}\vdash_{\mathtt{po}}P_1:M_1\qquad\Gamma,\mathcal{R}\vdash_{\mathtt{po}}P_2:M_2}{\Gamma,\mathcal{R}\vdash_{\mathtt{po}}P_1\mid P_2:M_1\uplus M_2}$$

$$(\textbf{PoOut})\frac{\Gamma,\mathcal{R}\vdash_{\mathtt{po}}P:M\qquad\Gamma(a)=\sharp^k_{\mathtt{R}_a}\ \widetilde{T}\qquad\Gamma(\widetilde{v})=T_i\qquad\mathcal{R}_a\lfloor\widetilde{v}\rfloor\subseteq\mathcal{R}}{\Gamma,\mathcal{R}\vdash_{\mathtt{po}}\overline{a}\langle\widetilde{v}\rangle.P:M\uplus\{a\}}$$

$$(\textbf{PoIn})\frac{\Gamma,\mathcal{R}'\vdash_{\mathtt{po}}P:M\qquad\Gamma(a)=\sharp^k_{\mathtt{R}_a}\ \widetilde{T}\qquad\Gamma(\widetilde{x})=\widetilde{T}\qquad\mathcal{R}'=\mathtt{R}_a\lfloor\widetilde{x}\rfloor\uplus\mathcal{R}}{\Gamma,\mathcal{R}\vdash_{\mathtt{po}}a(\widetilde{x}).P:M}$$

$$(\textbf{PoRep})\frac{\begin{array}{c}\Gamma,\mathcal{R}'\vdash_{\mathtt{po}}P:M\qquad\Gamma(a_i)=\sharp^{k_i}_{\mathtt{R}_{a_i}}\ \widetilde{T_i}\qquad\Gamma(\widetilde{x_i})=\widetilde{T_i}\qquad\mathcal{R}'=\mathtt{R}_{a_1}\lfloor\widetilde{x_1}\rfloor\uplus\cdots\uplus\mathtt{R}_{a_n}\lfloor\widetilde{x_n}\rfloor\uplus\mathcal{R}\\ \{a_1,\ldots,a_n\}\mathtt{mul}_{(>_{\mathtt{1v1}},\mathcal{R})}M\qquad\textbf{safe}(\{a_1,\ldots,a_n\},M,P)\end{array}}{\Gamma,\mathcal{R}\vdash_{\mathtt{po}}!a_1(\widetilde{x_1}).\ldots.a_n(\widetilde{x_n}).P:\emptyset}$$

Figure 3.5: Typing rules making use of a partial order

condition on $\mathcal{R}$ and by writing $\{a_1,\ldots,a_n\}\mathtt{mul}_{(>_{\mathtt{1v1}},=)}M$. Note that every time we go under an input prefix, the associated ordering grows, but only a disjoint partial order component is added. Thus, names bound by input cannot be related with free names. Names bound by restriction however, can be related with free names. As a consequence, we can distinguish two classes of names (names bound by input on one side, names bound by restriction and free names on the other side), and partial order cannot relate a name from one class with name from the other.

As stated in the previous remark, information on restricted names is hidden and does not appear in the partial order. However, as evaluation contexts can have the hole under a restriction, it is important to consider names bound by restriction as free, to compute the effective partial order typing a process; in other words, the partial order taking into account the comparisons of the names which are free in the hole of the evaluation context. Informally, this partial order is obtained by making explicit the partial order informations hidden by the occurrences of (**PoRes**) in the typing derivation and can be viewed as the extension of $\mathcal{R}$ on restricted names of the process.

Definition 3.3.6 (Effective partial order)
If $\Gamma,\mathcal{R}\vdash_{\mathtt{po}}P:M$, we define $\mathcal{R}_{\blacktriangleright(P)}$, the effective partial order of P, as the relation defined by:

$$\mathcal{R}_{\blacktriangleright((\nu c)\ P)}=\mathcal{R}'_{\blacktriangleright(P)}\ \textit{if}\ \mathcal{R}=\mathcal{R}'-\{c\}\ \textit{and}\ \Gamma,\mathcal{R}'\vdash_{\mathtt{po}}P:M\qquad\qquad\mathcal{R}_{\blacktriangleright(P_1\mid P_2)}=\mathcal{R}_{\blacktriangleright(P_1)}\cup\mathcal{R}_{\blacktriangleright(P_2)}$$

$$\mathcal{R}_{\blacktriangleright(\overline{a}\langle v\rangle.P)}=\mathcal{R}_{\blacktriangleright(!a(x).P)}=\mathcal{R}_{\blacktriangleright(a(x).P)}=\mathcal{R}_{\blacktriangleright(\mathbf{0})}=\mathcal{R}$$

As we use unions of partial orders, the following lemma is required, for the previous definition to be sound.

Lemma 3.3.7 (Soundness of Definition 3.3.6)
If $\Gamma,\mathcal{R}\vdash_{\mathtt{po}}P:M$, then $\mathcal{R}_{\blacktriangleright(P)}$ is a partial order.

Proof. *By induction on the typing judgement, we prove "$\mathcal{R}_{\blacktriangleright(P)}$ is a partial order and if $\widetilde{c}$ is the set of restricted names in P not under a prefix, $\mathcal{R}_{\blacktriangleright(P)}-\{\widetilde{c}\}=\mathcal{R}$". Cases related to prefixes are trivial, the case related to restrictions is treated using the induction hypothesis.*

$$(\textbf{PoRep}+)\ \frac{\Gamma, \mathcal{R}' \vdash^{+}_{\mathtt{po}} P : M \qquad \Gamma(a_i) = \sharp^{k_i}_{\mathtt{R}_{a_i}} \widetilde{T_i} \qquad \Gamma(\widetilde{x_i}) = \widetilde{T_i} \qquad \mathcal{R}' = \mathtt{R}_{a_1}\lfloor\widetilde{x_1}\rfloor \uplus \cdots \uplus \mathtt{R}_{a_n}\lfloor\widetilde{x_n}\rfloor \uplus \mathcal{R} \quad \{a_1,\ldots,a_n\}\mathtt{mul}_{(>_{\mathtt{1v1}},\mathcal{R})}M \qquad \textbf{safe}(\{a_1,\ldots,a_n\}, M, P)}{\Gamma, \mathcal{R} \vdash^{+}_{\mathtt{po}} !a_1^{l_1}(\widetilde{x_1}).\ldots.a_n^{l_n}(\widetilde{x_n}).P : \emptyset}$$

Figure 3.6: Typing rule for the annotated calculus with partial order

If $P = P_1 \mid P_2$, we have to show that $\mathcal{R}_{\blacktriangleright(P_1)} \cup \mathcal{R}_{\blacktriangleright(P_2)}$ does not include loops. Suppose, toward a contradiction, that this is the case, i.e. that, for instance, $a\ \mathcal{R}_{\blacktriangleright(P_1)}\ b$ and $b\ \mathcal{R}_{\blacktriangleright(P_2)}\ a$. Then either a or b is restricted in either P_1 or P_2, or $\mathcal{R}$ contains the loop (using the induction hypothesis on P_1 and P_2). Suppose a is restricted in P_1, then the fact that $b\ \mathcal{R}_{\blacktriangleright(P_2)}\ a$ contradicts the induction hypothesis $\mathcal{R}_{\blacktriangleright(P_2)} - \{\widetilde{c}\} = \mathcal{R}$, as $a \notin \widetilde{c}$. □

3.3.3 Termination proof

Like in Section 3.2.1, the termination property is obtained through using an auxiliary calculus, which is actually the same as the one we used above. The syntax and operational semantics for the annotated calculus are the same as the one presented in Figure 3.3. The typing rule for the annotated calculus are obtained as follows: the rule for unannotated prefixes are the same as the ones for he unannotated calculus and are given by Figure 3.5, the rule for input sequences is given by Figure 3.6.

Note that the Simulation Lemma (Lemma 3.2.7) still holds. Again, we prove that typability is stable when we add annotations.

Fact 3.3.8 (Annotated calculus - Stability of typability)
If $\Gamma, \mathcal{R} \vdash^{+}_{\mathtt{po}} P : M$, then $\Gamma, \mathcal{R} \vdash_{\mathtt{po}} \mathtt{free}(P) : M$ for some $\mathcal{R}$.

Proof. *Easily done by induction over the typing judgement $\Gamma, \mathcal{R} \vdash^{+}_{\mathtt{po}} P : M$, as in Fact 3.3.8.* □

Now we enter a more technical part. First, we have to prove that we can relax the multiset comparison: when we decompose the two multisets being compared M_1 and M_2 into $N \uplus N_1$ and $N \uplus N_2$, we suppose N maximal for the inclusion relation. This is actually not necessary.

Lemma 3.3.9 (Relaxing the multiset comparison)
If there exists N s.t. $M_1 = N_1 \uplus N$,$M_2 = N_2 \uplus N$, and $\forall e_2 \in N_2, \exists e_1 \in N_1, e_1 > e_2$ and $>$ is a partial order then $M_1 >_{\mathrm{mul}} M_2$.

Proof.
Take N' maximum s.t. $M_1 = N_1' \uplus N'$ and $M_2 = N_2' \uplus N'$. Notice that $N \subseteq N'$ and thus $N_1' \subseteq N_1$ and $N_2' \subseteq N_2$. Consider e_2 in N_2', as $e_2 \in N_2$, by hypothesis, there exists e_3 in N_1 s.t. $e_3 > e_2$. Either e_3 is in N_1' and we conclude or e_3 is not in N_1' and thus is in N'. Finally, $e_3 \in N_1 \cap N'$. We can rephrase this as "there is one copy of e_3 which is in N' but not in N''. Thus $e_3 \in N_2$, and we repeat the process until we find e_i which is in N_1'. The process terminates as there is only a finite number of names involved, and $>$ is well-founded (as it is a partial order on a finite number of names). By transitivity of $>$, we get $e_i > e_2$ and we conclude.

□

Then we state that when applying a type-preserving, order-preserving substitution, we preserve the multiset comparison checked in the typing rules. This result is not trivial, as the maximal common subset of the two multisets can change. For instance, take $M_1 = \{a, x, b\}, M_2 = \{a, y, c\}$ and the substitution $\sigma = \{b/x\}\{d/y\}$ with $y\mathcal{R}x, c\mathcal{R}b, b\mathcal{R}d$. $M_1 \mathcal{R}_{\mathtt{mul}} M_2$ as, in this case, the maximal N is $\{a\}$, $y\mathcal{R}x$ and $c\mathcal{R}b$. However $M_1\sigma = \{a, b, d\}$, $M_2\sigma = \{a, b, c\}$, and in this case the maximal N is $\{a, b\}$ and $M_1 \mathcal{R}_{\mathtt{mul}} M_2$ as $c\mathcal{R}d$ (because $c\mathcal{R}b$ and $b\mathcal{R}d$).

Lemma 3.3.10 (Multiset comparison and substitution)
Let M and N be two multisets of names, if $M_1 \mathtt{mul}_{(>_{\mathtt{lvl}}, \mathcal{R} \uplus \mathtt{R}_0\lfloor \widetilde{x} \rfloor)} M_2$, $\Gamma(\widetilde{x}) = \Gamma(\widetilde{v})$, $\mathtt{R}_0\lfloor \widetilde{v} \rfloor \subseteq \mathcal{R}$, then $M_1\{\widetilde{v}/\widetilde{x}\}$ $\mathtt{mul}_{(>_{\mathtt{lvl}}, \mathcal{R})}$ $M_2\{\widetilde{v}/\widetilde{x}\}$.

Proof.
As $\Gamma(\widetilde{v}) = \Gamma(\widetilde{x})$, $\mathtt{lvl}(M_1\{\widetilde{v}/\widetilde{x}\}) = \mathtt{lvl}(M_1\{\widetilde{v}/\widetilde{x}\})$ and $\mathtt{lvl}(M_1\{\widetilde{v}/\widetilde{x}\}) = \mathtt{lvl}(M_1\{\widetilde{v}/\widetilde{x}\})$, thus, $M_1\{\widetilde{v}/\widetilde{x}\} >_{\mathtt{lvl}}$ $M_2\{\widetilde{v}/\widetilde{x}\}$ if and only if $M_1 >_{\mathtt{lvl}} M_2$.
We know $(i) : \mathtt{R}_0\lfloor \widetilde{v} \rfloor \subseteq \mathcal{R}$. We call N the maximum multiset s.t. $M_1 = N \uplus N_1$ and $M_2 = N \uplus N_2$. We write $N' = N\{\widetilde{v}/\widetilde{x}\}$, $N_1' = N_1\{\widetilde{v}/\widetilde{x}\}$ and $N_2' = N_2\{\widetilde{v}/\widetilde{x}\}$.
We note $\sigma = \{\widetilde{v}/\widetilde{x}\}$ for the sake of clarity. We prove that the condition (i) ensures e_1 $(\mathcal{R} \uplus \mathtt{R}_0\lfloor \widetilde{x} \rfloor)$ e_2 implies $\sigma(e_1)$ $\mathcal{R}$ $\sigma(e_2)$. Indeed,

- *either e_1 is in $\widetilde{x}$, i.e. $e_1 = x_i$, in this case e_2 is also in $\widetilde{x}$, according to the definition of $\uplus$, i.e. $e_2 = x_j$, we deduce i $\mathtt{R}_0$ j. As $\mathtt{R}_0\lfloor \widetilde{v} \rfloor \subseteq \mathcal{R}$ we have v_i $\mathcal{R}$ v_j. We conclude as $v_i = \sigma(e_1)$ and $v_j = \sigma(e_2)$.*
- *or e_1 is not in $\widetilde{x}$. In this case, e_2 is not in $\widetilde{x}$ neither, according to the definition of $\uplus$. We deduce $e_1 \mathcal{R} e_2$ and conclude, as $\sigma(e_1) = e_1$ and $\sigma(e_2) = e_2$.*

Take an element e_2' in N_2', there exists e_2 in N_2, s.t. $e_2' = \sigma(e_2)$. By definition of the multiset comparison, there exists e_1 in N_1, s.t. $e_1(\mathcal{R} \uplus \mathtt{R}_0\lfloor \widetilde{x} \rfloor)e_2$. We use the above result to get $\sigma(e_1)\mathcal{R}\sigma(e_2)$.
By Lemma 3.3.9, we conclude. □

As usual, we state that types are preserved by structural congruence.

Fact 3.3.11 (Subject Congruence)
If $P \equiv Q$ then $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P : M$ iff $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} Q : M$.

Proof. *By induction over the derivation of $P \equiv Q$, using the associativity, the commutativity, the neutrality of $\emptyset$ for the operator $\uplus$ and the fact that $(\mathcal{R} - \{a\}) - \{b\} = (\mathcal{R} - \{b\}) - \{a\}$.* □

The following fact states that we can always add a disjoint component to the partial order used to type a process. Notice that this result no longer holds if we use $\cup$ instead of $\uplus$. Think of $\mathcal{R} = \{(a, b)\}$ and $\mathcal{R}'' = \{(b, a)\}$.

Fact 3.3.12 (Weakening Lemma)
If $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P : M$, and $\mathcal{R}' = \mathcal{R} \uplus \mathcal{R}''$, then $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} P : M$.

Proof. *Easily done by induction on the typing judgement. The interesting part is to remark that when $\mathcal{R} = \mathcal{R}_1 \uplus \mathtt{R}\lfloor \widetilde{x} \rfloor$, $\mathcal{R} = (\mathcal{R}_1 \uplus \mathcal{R}'') \uplus \mathtt{R}\lfloor \widetilde{x} \rfloor$.* □

As in the previous section, Subject Substitution states two separate results: that typability is preserved by types-preserving, partial order-preserving substitutions, and that it is also the case for partially consumed input sequences. However, this lemma is not the direct application of the one of Section 3.2.1, because in the case of rule (**PoRep**+), one has to accommodate a much more complicated comparison. We have to use the results stated above to prove that substitution can be applied to multisets being compared with partial orders.

Lemma 3.3.13 (Subject Substitution)

1. *If $\Gamma(\widetilde{x}) = \Gamma(\widetilde{v})$, $\mathcal{R} = \mathcal{R}' \uplus \mathtt{R}_0\lfloor \widetilde{x} \rfloor$, $\mathtt{R}_0\lfloor \widetilde{v} \rfloor \subseteq \mathcal{R}$ and $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P : M$, then $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P\{\widetilde{v}/\widetilde{x}\} : M\{\widetilde{v}/\widetilde{x}\}$,*
2. *if $\Gamma(\widetilde{x}) = \Gamma(\widetilde{v})$, $\mathcal{R} = \mathcal{R}' \uplus \mathtt{R}_0\lfloor \widetilde{x} \rfloor$, $\mathtt{R}_0\lfloor \widetilde{v} \rfloor \subseteq \mathcal{R}$, $\{a_1, \ldots, a_{i-1}\} \cap \widetilde{x} = \emptyset$ and $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.a_i(x_i)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P : M\{\widetilde{v}/\widetilde{x}\}$, then $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{i-1}(x_{i-1})^{\mathtt{ok}}.((a_i(x_i)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P)\{v/x\}) : M\{\widetilde{v}/\widetilde{x}\}$.*

Remember that, although it is not made explicit in the statement of the lemma, we suppose that the names of $\widetilde{x}$ are not bound in P.

Proof.

1. *We prove the first result by induction over the typing judgement:*
   - *Case* (**PoNil**) *is trivial, as* $\mathbf{0}\{\widetilde{v}/\widetilde{x}\} = \mathbf{0}$.
   - *Case* (**PoPar**) *is easily done by using the induction hypothesis twice, as* $(P_1 \mid P_2)\{\widetilde{v}/\widetilde{x}\} = (P_1\{\widetilde{v}/\widetilde{x}\} \mid P_2\{\widetilde{v}/\widetilde{x}\})$.
   - *Case* (**PoRes**). *From* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} (\nu c)\ P_1 : M$ *we derive* $\Gamma, \mathcal{R}_1 \vdash^+_{\mathtt{po}} P_1 : M$ *and* $\mathcal{R} = \mathcal{R}_1 - \{c\}$. *Suppose* $\mathcal{R} = \mathcal{R}' \uplus \mathtt{R}_0\lfloor\widetilde{x}\rfloor$, *as the domain of* $\mathtt{R}_0\lfloor\widetilde{x}\rfloor$ *is* $\widetilde{x}$ *and as* $c \notin \widetilde{x}$ *(because our processes abide the Barendregt Convention), we deduce* $\mathcal{R}_1 = \mathcal{R}'_1 \uplus \mathtt{R}_0\lfloor\widetilde{x}\rfloor$ *with* $\mathcal{R}' = \mathcal{R}'_1 - \{c\}$. *If* $\mathtt{R}_0\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R}$, *we obtain directly* $\mathtt{R}_0\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R}_1$. *This allows us to use the induction hypothesis to get* $\Gamma, \mathcal{R}_1 \vdash^+_{\mathtt{po}} P_1 : M\{\widetilde{v}/\widetilde{x}\}$. *We conclude using* (**PoRes**).
   - *Case* (**PoIn**). *We have* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} a(\widetilde{y}).P_1 : M$. *We derive* $\Gamma, \mathcal{R}_1 \vdash^+_{\mathtt{po}} P_1 : M$, $\mathcal{R}_1 = \mathcal{R} \uplus \mathtt{R}_a\lfloor\widetilde{y}\rfloor$, $\Gamma(a) = \sharp^k_{\mathtt{R}_a}\widetilde{T}$ *and* $\Gamma(\widetilde{y}) = \widetilde{T}$. *Thus* $\mathcal{R}_1 = \mathcal{R} \uplus \mathtt{R}_a\lfloor\widetilde{y}\rfloor = (\mathcal{R}' \uplus \mathtt{R}_a\lfloor\widetilde{y}\rfloor) \uplus \mathtt{R}_0\lfloor\widetilde{x}\rfloor$. *Moreover, as* $\mathtt{R}_0\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R}$, $\mathtt{R}_0\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R}_1$. *This allows us to use the induction hypothesis to get* $\Gamma, (\mathcal{R}' \uplus \mathtt{R}_a\lfloor\widetilde{y}\rfloor) \vdash^+_{\mathtt{po}} P_1\{\widetilde{v}/\widetilde{x}\} : M\{\widetilde{v}/\widetilde{x}\}$. *Fact 3.3.12 ensures* $\Gamma, \mathcal{R}_1 \vdash^+_{\mathtt{po}} P_1\{\widetilde{v}/\widetilde{x}\} : M\{\widetilde{v}/\widetilde{x}\}$. *The Barendregt convention ensures* $\widetilde{x} \cap \widetilde{y} = \emptyset$ *and two cases can occur:*
     - *Either* $a \notin \widetilde{x}$ *and* $(a(y).P_1)\{\widetilde{v}/\widetilde{x}\} = a(y).(P_1\{\widetilde{v}/\widetilde{x}\})$, *we use* (**PoIn**) *to conclude.*
     - *Or* $a \in \widetilde{x}$ *(say* $a = x_i$*) and* $(a(y).P_1)\{\widetilde{v}/\widetilde{x}\} = v_i(y).(P_1\{\widetilde{v}/\widetilde{x}\})$. *As* $\Gamma(v_i) = \Gamma(a) = \sharp^k_{\mathtt{R}_a}\ T_i$, *we use* (**PoIn**) *to conclude.*
   - *Case* (**PoOut**). *We have* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} \overline{a}\langle\widetilde{w}\rangle.P_1 : M$. *We derive* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P_1 : M_1$, $\mathtt{R}_a\lfloor\widetilde{w}\rfloor \subseteq \mathcal{R}$,$\Gamma(a) = \sharp^k_{\mathtt{R}_a}\widetilde{T}$, $\Gamma(\widetilde{w}) = \widetilde{T}$ *and* $M = \{a\} \uplus M_1$ *and use induction hypothesis to get* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P_1\{\widetilde{v}/\widetilde{x}\} : M_1$. *Three cases can occur:*
     - *Either* $a \notin \widetilde{x}$ *and* $w \notin \widetilde{x}$ *and* $(\overline{a}\langle w\rangle.P_1)\{\widetilde{v}/\widetilde{x}\} = \overline{a}\langle w\rangle.(P_1\{\widetilde{v}/\widetilde{x}\})$, *we use* (**PoOut**) *to conclude.*
     - *Or* $a \in \widetilde{x}$ *(say* $a = x_i$*) and* $w \notin x$, *and* $(\overline{a}\langle w\rangle.P_1)\{\widetilde{v}/\widetilde{x}\} = \overline{v_i}\langle w\rangle.(P_1\{\widetilde{v}/\widetilde{x}\})$. *As* $\Gamma(v_i) = \Gamma(a) = \sharp^k_{\mathtt{R}_a}\ T_i$, *we use* (**PoOut**) *to get* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} (\overline{v_i}\langle w\rangle.P_1)\{\widetilde{v}/\widetilde{x}\} : \{v_i\} \uplus M$. *We conclude, as* $M\{\widetilde{v}/\widetilde{x}\} = \{v_i\} \uplus M_1\{\widetilde{v}/\widetilde{x}\}$.
     - *Or* $a \notin \widetilde{x}$ *and* $w \in \widetilde{x}$ *(say,* $w = x_i$*), and* $(\overline{a}\langle w\rangle.P_1)\{\widetilde{v}/\widetilde{x}\} = \overline{a}\langle v_i\rangle.(P_1\{\widetilde{v}/\widetilde{x}\})$. *As* $\Gamma(v_i) = \Gamma(w) = T$, *we use* (**PoOut**) *to conclude* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} (\overline{a}\langle v\rangle.P_1)\{\widetilde{v}/\widetilde{x}\} : \{k\} \uplus M$).
     - *Case* $a = w = x$ *cannot happen as the calculus is simply-typed.*
   - *Case* (**PoRep**+). *We have* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} !a_1(\widetilde{y_1})^{l_1}.\ldots.a_n(\widetilde{y_n})^{l_n}.P_1 : \emptyset$. *We derive* $\Gamma, \mathcal{R}_1 \vdash^+_{\mathtt{po}} P_1 : M_1$, *for all* i, $\Gamma(a_i) = \sharp^{k_i}_{\mathtt{R}_{a_i}}\widetilde{T_i}$, $\Gamma(\widetilde{y_i}) = \widetilde{T_i}$, $\mathcal{R}_1 = \mathtt{R}_{a_1}\lfloor\widetilde{y_1}\rfloor \uplus \cdots \uplus \mathtt{R}_{a_n}\lfloor\widetilde{y_n}\rfloor \uplus \mathcal{R}$, $\{a_1, \ldots, a_n\}\mathtt{mul}_{(>_{1\mathrm{v}1}, \mathcal{R})}M_1$ *and* $\mathbf{safe}(\{a_1, \ldots, a_n\}, M_1, P_1)$. *The Barendregt convention prevents any element of* $\widetilde{y_i}$ *from being in* $\widetilde{x}$. *We note* σ *the mapping of names into names which is the identity except on* $\widetilde{x}$ *which is mapped to* $\widetilde{v}$. *Clearly, for all* i, $\Gamma(\sigma(a_i)) = \Gamma(a_i) = \sharp^{k_i}_{\mathtt{R}_{a_i}}\widetilde{T_i}$. *As in case* (**PoIn**), *we have* $\mathcal{R}_1 = (\mathcal{R}' \uplus \mathcal{R}_2) \uplus \mathtt{R}_0\lfloor\widetilde{x}\rfloor$ *and* $\mathtt{R}_0\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R} \subseteq \mathcal{R}_1$. *We use the induction hypothesis to get* $\Gamma, (\mathcal{R}' \uplus \mathcal{R}_2) \vdash^+_{\mathtt{po}} P_1\{\widetilde{v}/\widetilde{x}\} : M\{\widetilde{v}/\widetilde{x}\}$ *and Fact 3.3.12 to get* $\Gamma, \mathcal{R}_1 \vdash^+_{\mathtt{po}} P_1\{\widetilde{v}/\widetilde{x}\} : M\{\widetilde{v}/\widetilde{x}\}$. *By Lemma 3.3.10,* $\{\sigma a_1, \ldots, \sigma a_n\}\mathtt{mul}_{(>_{1\mathrm{v}1}, \mathcal{R})}M_1\{\widetilde{v}/\widetilde{x}\}$. *Condition* $\mathbf{safe}(\{\sigma a_1, \ldots, \sigma a_n\}, M_1\{\widetilde{v}/\widetilde{x}\}, P_1\{\widetilde{v}/\widetilde{x}\})$ *still holds as level are unaffected by* σ. *We use rule* (**PoRep**+) *to conclude* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} !\sigma(a_1)(\widetilde{y_i})^{l_1}.\ldots.\sigma(a_n)(\widetilde{y_n})^{l_n}.P_1 : \emptyset$.

2. *We have* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{q-1}(x_{q-1})^{\mathtt{free}}.a_q(x_q)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1 : N$. *We derive* $\Gamma, \mathcal{R}_1 \vdash^+_{\mathtt{po}} P_1 : M_1$, *for all* i, $\Gamma(a_i) = \sharp^{k_i}_{\mathtt{R}_{a_i}}\widetilde{T_i}$, $\Gamma(\widetilde{y_i}) = \widetilde{T_i}$, $\mathcal{R}_1 = \mathtt{R}_{a_1}\lfloor\widetilde{y_1}\rfloor \uplus \cdots \uplus \mathtt{R}_{a_n}\lfloor\widetilde{y_n}\rfloor \uplus \mathcal{R}$, $\{a_1, \ldots, a_n\}\mathtt{mul}_{(>_{1\mathrm{v}1}, \mathcal{R})}M_1$ *and* $\mathbf{safe}(\{a_1, \ldots, a_n\}, M_1, P_1)$. *By reasoning similarly to case 1, we derive* $\Gamma, \mathcal{R}_1 \vdash^+_{\mathtt{po}} P_1\{\widetilde{v}/\widetilde{x}\} : M\{\widetilde{v}/\widetilde{x}\}$. *We conclude as in case* (**PoRep**+) *of result* 1, *only applying* σ *to* a_i *where* $i \geq q$, *noticing that, as* $\{a_1, \ldots, a_{i-1}\} \cap \widetilde{x} = \emptyset$, $\{a_1, \ldots, a_{q-1}, \sigma a_q, \ldots, \sigma a_n\} = \{\sigma a_1, \ldots, \sigma a_n\}$, *and thus, the reasoning using Lemma 3.3.10 and the one with* $\mathbf{safe}(\cdot, \cdot, \cdot)$ *still holds.*

□

Relating contexts and typability is again necessary. Yet, we also need to relate evaluation contexts and partial orders, as the partial order used to type the process found inside the hole could be different from the one used to type the whole process, because evaluation contexts go under restrictions.

Lemma 3.3.14 (Context Typing)
If $\Gamma, \mathcal{R} \vdash^{+}_{\mathtt{po}} \mathbf{E}[P] : M$ *then there exists* $\mathcal{R}' \subseteq \mathcal{R}_{\blacktriangleright(P)}$ *such that:*

1. $\Gamma, \mathcal{R}' \vdash^{+}_{\mathtt{po}} P : M'$ *for some* $M' \subseteq M$.
2. *For all* P_0 *s.t.* $\Gamma, \mathcal{R}' \vdash^{+}_{\mathtt{po}} P_0 : M_0$, *we have* $\Gamma, \mathcal{R} \vdash^{+}_{\mathtt{po}} \mathbf{E}[P_0] : M_{(0)}$ *for some* $M_{(0)}$.

Proof. *By structural induction over* $\mathbf{E}$.

- *Case* . *Condition 1 holds trivially and condition 2 holds by setting* $N_{(0)} = N_0$.
- *Case* $(\boldsymbol{\nu} a)\ \mathbf{E}_2$. *We derive* $\Gamma, \mathcal{R}_2 \vdash^{+}_{\mathtt{po}} \mathbf{E}_2[P] : M$ *with* $\mathcal{R} = \mathcal{R}_2 - \{c\}$. *We use the induction hypothesis to deduce condition 1 and condition 2 holds by setting* $N_{(0)} = N_0$. *We conclude as* $\mathcal{R}_{2\blacktriangleright(\mathbf{E}_2[P])} = \mathcal{R}_{\blacktriangleright((\nu a)\ \mathbf{E}_2[P])}$, *by Definition 3.3.6.*
- *Case* $\mathbf{E} = \mathbf{E}_2 \mid P_1$. *We derive* $\Gamma, \mathcal{R} \vdash^{+}_{\mathtt{po}} \mathbf{E}_2[P] : N_2$ *and* $\Gamma, \mathcal{R} \vdash^{+}_{\mathtt{po}} P_1 : N_1$ *with* $N = N_2 \uplus N_1$. *The induction hypothesis gives* $\Gamma, P \vdash^{+}_{\mathtt{po}} N' :$ *for some* N', *thus we deduce condition 1. The induction hypothesis also gives* $\Gamma, \mathcal{R} \vdash^{+}_{\mathtt{po}} \mathbf{E}_2[P_0] : N_{(2)}$ *for some* $N_{(2)}$. *We set* $N_{(0)} = N_{(2)} \uplus N_1$ *and we get condition 2. We conclude as* $\mathcal{R}_{2\blacktriangleright(\mathbf{E}_2[P])} \subseteq \mathcal{R}_{\blacktriangleright(\mathbf{E}_2[P] \mid P_1)}$, *by Definition 3.3.6 and Fact 3.3.12.*

□

The measure used here is not different from the one used in the previous section. However here we use multisets of names, instead of multiset of levels. Yet, the facts and lemmas proved on $\mathbf{AvRes}()$ still hold for $\mathbf{AvRes}^{\mathtt{po}}()$ by adapting easily the corresponding proofs.

Definition 3.3.15 (Available resources) *The multiset of available resources is inductively defined by:*

$$\mathbf{AvRes}^{\mathtt{po}}(\mathbf{0}) = \emptyset \qquad \mathbf{AvRes}^{\mathtt{po}}(P_1 \mid P_2) = \mathbf{AvRes}^{\mathtt{po}}(P_1) \uplus \mathbf{AvRes}^{\mathtt{po}}(P_2)$$

$$\mathbf{AvRes}^{\mathtt{po}}((\boldsymbol{\nu} a)\ P) = \mathbf{AvRes}^{\mathtt{po}}(P) \qquad \mathbf{AvRes}^{\mathtt{po}}(\overline{a}\langle v\rangle.P) = \{a\} \uplus \mathbf{AvRes}^{\mathtt{po}}(P)$$

$$\mathbf{AvRes}^{\mathtt{po}}(a(x).P) = \mathbf{AvRes}^{\mathtt{po}}(P)$$

$$\mathbf{AvRes}^{\mathtt{po}}(!a_1(x_1)^{\mathtt{ok}}.\ldots.a_i(x_i)^{\mathtt{ok}}.a_{i+1}(x_{i+1})^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P) = \{a_1\} \uplus \cdots \uplus \{a_i\}$$

The operator $\mathbf{AvRes}^{\mathtt{po}}()$ *is straightforwardly extended to evaluation contexts.*

Fact 3.3.16 (Context and available resources)
If $\Gamma, \mathcal{R} \vdash^{+}_{\mathtt{po}} \mathbf{E}[P] : N$, *then* $\mathbf{AvRes}^{\mathtt{po}}(\mathbf{E}[P]) = \mathbf{AvRes}^{\mathtt{po}}(\mathbf{E}) \uplus \mathbf{AvRes}^{\mathtt{po}}(P)$.

Proof. *Easily adapted from the proof of Fact 3.2.13.* □

Fact 3.3.17 (Substitution and available resources)
If $\Gamma, \mathcal{R} \vdash^{+}_{\mathtt{po}} P : N$ *and* $\Gamma(\widetilde{v}) = \Gamma(\widetilde{x})$, *then* $\mathbf{AvRes}^{\mathtt{po}}(P)\{\widetilde{v}/\widetilde{x}\} = \mathbf{AvRes}^{\mathtt{po}}(P\{\widetilde{v}/\widetilde{x}\})$.

Proof. *Adapted from the proof of Fact 3.2.14.* □

Fact 3.3.18 (Structural congruence and available resources)
If $P \equiv Q$ *then* $\mathbf{AvRes}^{\mathtt{po}}(P) = \mathbf{AvRes}^{\mathtt{po}}(Q)$.

Proof. *Easily adapted from the proof of Fact 3.2.15.* □

Fact 3.3.19
If the annotations in P contain only `free` *and* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P : M$*, then* $\mathbf{AvRes}^{po}(P) = M$*.*

Proof. *Adapted from the proof of Lemma 3.2.16* □

We state the main result, the subject reduction. In this proof, two processes P and P' are compared w.r.t. a typing context Γ, this comparison is done using the relation $\mathtt{Lex}(\mathtt{mul}_{(>_{\mathtt{lvl}}, \mathcal{R}_{\blacktriangleright(P)})}, \subsetneq)$ on the pairs $(\mathbf{AvRes}^{\mathtt{po}}(P), \mathbf{Os}(P))$ and $(\mathbf{AvRes}^{\mathtt{po}}(P'), \mathbf{Os}(P'))$. First, the multisets of names $\mathbf{AvRes}^{\mathtt{po}}(P)$ and $\mathbf{AvRes}^{\mathtt{po}}(P')$ are compared using $\mathtt{mul}_{(>_{\mathtt{lvl}}, \mathcal{R}_{\blacktriangleright(P)})}$, that is, we begin by comparing the multiset of levels (according to Γ) of the names in $\mathbf{AvRes}^{\mathtt{po}}(P)$ with the multiset of levels of the names in $\mathbf{AvRes}^{\mathtt{po}}(P')$ using the multiset extension of the standard ordering over natural numbers (formally we check if $\mathtt{lvl}(\mathbf{AvRes}^{\mathtt{po}}(P)) <_{\mathrm{mul}} \mathtt{lvl}(\mathbf{AvRes}^{\mathtt{po}}(P'))$), if these multisets of numbers are equal, we compare $\mathbf{AvRes}^{\mathtt{po}}(P)$ with $\mathbf{AvRes}^{\mathtt{po}}(P')$ using the multiset extension of the effective partial order of P, $\mathcal{R}_{\blacktriangleright(P)}$. Then, if the multisets of names $\mathbf{AvRes}^{\mathtt{po}}(P)$ and $\mathbf{AvRes}^{\mathtt{po}}(P')$ are equal, we compare the numbers of outputs in P and P' which are $\mathbf{Os}(P)$ and $\mathbf{Os}(P')$.

Proposition 3.3.20 (Subject reduction)
If $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P : N$ and $P \to P'$, then

1. *$\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P' : N'$ for some N'.*
2. *$(\mathbf{AvRes}^{po}(P), \mathbf{Os}(P)) \mathit{Lex}(\mathtt{mul}_{(>_{\mathtt{lvl}}, \mathcal{R}_{\blacktriangleright(P)})}, \subsetneq)(\mathbf{AvRes}^{po}(P'), \mathbf{Os}(P'))$*

Proof.

- *Case* (**KCong**). *We use the induction hypothesis and Facts 3.3.11, 3.1.9 and 3.3.18 to conclude.*
- *Case* (**KComm**). *We have $P = \mathbf{E}[a(\widetilde{x}).P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$ and $P' = \mathbf{E}[P_1\{\widetilde{v}/\widetilde{x}\} \mid P_2]$. From $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P : N$ and Lemma 3.3.14, we derive $\Gamma, R' \vdash^+_{\mathtt{po}} a(\widetilde{x}).P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2 : M$ for some $M \subseteq N$ and some $\mathcal{R}' \subseteq \mathcal{R}_{\blacktriangleright(P)}$. Then, we derive $\Gamma, \mathcal{R}'' \vdash^+_{\mathtt{po}} P_1 : M_1$, $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} P_2 : M_2$, $\Gamma(a) = \sharp^k_{\mathtt{R}_a} \widetilde{T}$, $\Gamma(\widetilde{v}) = \widetilde{T_1}$, $\mathcal{R}'' = \mathcal{R}' \uplus \mathtt{R}_a\lfloor\widetilde{x}\rfloor$, $\mathtt{R}_a\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R}'$ and $M = M_1 \uplus M_2 \uplus \{a\}$. Fact 3.3.16 gives $\mathbf{AvRes}^{po}(P) = \mathbf{AvRes}^{po}(\mathbf{E}) \uplus \mathbf{AvRes}^{po}(P_1) \uplus \mathbf{AvRes}^{po}(P_2) \uplus \{a\}$. As $\mathcal{R}'' = \mathcal{R}' \uplus \mathtt{R}_a\lfloor\widetilde{x}\rfloor$, $\mathtt{R}_a\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R}'$, Lemma 3.3.13 gives $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} P_1\{\widetilde{v}/\widetilde{x}\} : M_1\{\widetilde{v}/\widetilde{x}\}$. Using* (**PoPar**) *we derive $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} P_1\{\widetilde{v}/\widetilde{x}\} \mid P_2 : M_1\{\widetilde{v}/\widetilde{x}\} \uplus M_2$ and Lemma 3.3.14 gives $\Gamma, P' \vdash^+_{\mathtt{po}} N' :$ for some N'. Fact 3.3.16 gives $\mathbf{AvRes}^{po}(P') = \mathbf{AvRes}^{po}(\mathbf{E}) \uplus \mathbf{AvRes}^{po}(P_1\{v/x\}) \uplus \mathbf{AvRes}^{po}(P_2)$. Fact 3.3.17 gives $\mathbf{AvRes}^{po}(P_1\{v/x\}) = \mathbf{AvRes}^{po}(P_1)\{\widetilde{v}/\widetilde{x}\}$. As $\Gamma(\widetilde{v}) = \Gamma(\widetilde{x})$, we have $\mathtt{lvl}(\mathbf{AvRes}^{po}(P_1\{v/x\})) = \mathtt{lvl}(\mathbf{AvRes}^{po}(P_1))$, thus $\mathtt{lvl}(\mathbf{AvRes}^{po}(P)) = \mathtt{lvl}(\mathbf{AvRes}^{po}(P')) \uplus \{a\}$. We conclude, as $\mathbf{AvRes}^{po}(P) >_{\mathtt{lvl}} \mathbf{AvRes}^{po}(P')$.*
- *Case* (**KTrig**). *We have $P = \mathbf{E}[!a_1^{\mathtt{free}}(\widetilde{x_1}).a_2^{\mathtt{free}}(\widetilde{x_2}).\ldots.a_n^{\mathtt{free}}(\widetilde{x_n}).P_1 \mid \overline{a_1}\langle\widetilde{v}\rangle.P_2]$ and $P' = \mathbf{E}[!a_1^{\mathtt{free}}(\widetilde{x_1}).a_2^{\mathtt{free}}(\widetilde{x_2}).\ldots.a_n^{\mathtt{free}}(\widetilde{x_n}).P_1 \mid !a_1^{\mathtt{ok}}(\widetilde{x_1}).((a_2^{\mathtt{free}}(\widetilde{x_2}).\ldots.a_n^{\mathtt{free}}(\widetilde{x_n}).P_1)\{\widetilde{v}/\widetilde{x_1}\}) \mid P_2]$. From $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} P : N$ and Lemma 3.3.14, we derive $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} !a_1^{\mathtt{free}}(\widetilde{x_1}).a_2^{\mathtt{free}}(\widetilde{x_2}).\ldots.a_n^{\mathtt{free}}(\widetilde{x_n}).P_1 \mid \overline{a_1}\langle\widetilde{v}\rangle.P_2 : M$ for some $M \supseteq N$ and $\mathcal{R}' \subseteq \mathcal{R}_{\blacktriangleright(P)}$. Then, we derive $\Gamma, !a_1^{\mathtt{free}}(x_1).a_2^{\mathtt{free}}(x_2).\ldots.a_n^{\mathtt{free}}(x_n).P_1 \vdash^+_{\mathtt{po}} \emptyset :$, $\Gamma \vdash^{\kappa,a} P_1 : M_1$, $\Gamma \vdash^{\kappa,a} P_2 : M_2$, for all j, $\Gamma(a_j) = \sharp^{k_j}_{\mathtt{R}_{a_j}} \widetilde{T_j}$, $\Gamma(\widetilde{x_j}) = \widetilde{T_j}$, $\mathcal{R}' = \mathtt{R}_{a_1}\lfloor\widetilde{x_1}\rfloor \uplus \cdots \uplus \mathtt{R}_{a_n}\lfloor\widetilde{x_n}\rfloor \uplus \mathcal{R}''$, $\{a_1, \ldots, a_n\}\mathtt{mul}_{(>_{\mathtt{lvl}}, \mathcal{R}')}M$, $\mathbf{safe}(\{a_1, \ldots, a_n\}, M, P)$, $\mathtt{R}_{a_1} \subseteq \mathcal{R}'$ and $M = M_2 \uplus \{a_1\}$. Fact 3.3.16 gives $\mathbf{AvRes}^{po}(P) = \mathbf{AvRes}^{po}(\mathbf{E}) \uplus \mathbf{AvRes}^{po}(P_2) \uplus \{a_1\}$. Lemma 3.1.8 gives $\mathbf{Os}(P) = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_2) \uplus \{k_1\}$. As $\mathtt{R}_{a_1}\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R}' \subseteq (\mathcal{R}' \uplus \mathtt{R}_{a_2}\lfloor\widetilde{x_2}\rfloor \uplus \ldots)$ and $\mathcal{R}'' = (\mathcal{R}' \uplus \mathtt{R}_{a_2}\lfloor\widetilde{x_2}\rfloor \uplus \ldots) \uplus \mathtt{R}_{a_1}\lfloor\widetilde{x_1}\rfloor$, and as the Barendregt convention gives $\widetilde{x_1} \cap \{a_1\} = \emptyset$, Lemma 3.3.13 gives $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} !a_1^{\mathtt{ok}}.((a_2^{\mathtt{free}}(x_2).\ldots.a_n^{\mathtt{free}}(x_n).P_1)\{v/x_1\}) : \emptyset$. Using* (**PoPar**) *twice we derive $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}}$ $!a_1^{\mathtt{free}}.a_2^{\mathtt{free}}(x_2).\ldots.a_n^{\mathtt{free}}(x_n).P_1$ $\mid !a_1^{\mathtt{ok}}.((a_2^{\mathtt{free}}(x_2).\ldots.a_n^{\mathtt{free}}(x_n).P_1)\{v/x_1\}) \mid P_2 : \emptyset \uplus M_2$ and Lemma 3.3.14 gives $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P' : N'$ for*

some N'. Fact 3.2.13 gives $\mathbf{AvRes}(P') = \mathbf{AvRes}(\mathbf{E}) \uplus \mathbf{AvRes}(!a_1^{\mathtt{ok}}(\widetilde{x_1}).((a_2^{\mathtt{free}}(x_2).\ldots.a_n^{\mathtt{free}}(x_n).P_1)\{v/x_1\})) \uplus \mathbf{AvRes}(P_2)$. *From Definition 3.2.12 and Fact 3.3.17, we have*

$$\mathbf{AvRes}^{po}(!a_1^{\mathtt{ok}}(\widetilde{x_1}).((a_2^{\mathtt{free}}(x_2).\ldots.a_n^{\mathtt{free}}(x_n).P_1)\{v/x_1\})) = \{a_1\}.$$

Lemma 3.1.8 and Fact 3.1.10 we get $\mathbf{Os}(P') = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_2)$. *We conclude, as we proved that* $\mathbf{AvRes}(P) = \mathbf{AvRes}(P')$ *and* $\mathbf{Os}(P') \subsetneq \mathbf{Os}(P)$.

- *Case* (**KProg**). *We have* $P = \mathbf{E}[!a_1^{\mathtt{ok}}(\widetilde{x_1}).\ldots.a_{i-1}^{\mathtt{ok}}(\widetilde{x_{i-1}}).a_i^{\mathtt{free}}(\widetilde{x_i}).a_{i+1}^{\mathtt{free}}(\widetilde{x_{i+1}}).\ldots.a_n^{\mathtt{free}}(\widetilde{x_n}).P_1 \mid \overline{a_i}\langle\widetilde{v}\rangle.P_2]$ *and* $P' = \mathbf{E}[!a_1^{\mathtt{ok}}(\widetilde{x_1}).\ldots.a_{i-1}^{\mathtt{ok}}(\widetilde{x_{i-1}}).a_i^{\mathtt{ok}}(\widetilde{x_i}).((a_{i+1}^{\mathtt{free}}(\widetilde{x_{i+1}}).\ldots.a_n^{\mathtt{free}}(\widetilde{x_n}).P_1)\{\widetilde{v}/\widetilde{x_i}\}) \mid P_2]$. *From* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P : N$ *and Lemma 3.3.14, we derive*

$$\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} !a_1^{\mathtt{ok}}(\widetilde{x_1}).\ldots.a_{i-1}^{\mathtt{ok}}(\widetilde{x_{i-1}}).a_i^{\mathtt{free}}(\widetilde{x_i}).a_{i+1}^{\mathtt{free}}(\widetilde{x_{i+1}}).\ldots.a_n^{\mathtt{free}}(\widetilde{x_n}).P_1 \mid \overline{a_i}\langle\widetilde{v}\rangle.P_2 : M$$

for some $M \supseteq N$ *and* $\mathcal{R}' \subseteq \mathcal{R}_{\blacktriangleright(P)}$. *Then, we derive the following judgements:*

$$\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} !a_1^{\mathtt{ok}}(\widetilde{x_1}).\ldots.a_{i-1}^{\mathtt{ok}}(\widetilde{x_{i-1}}).a_i^{\mathtt{free}}(\widetilde{x_i}).a_{i+1}^{\mathtt{free}}(\widetilde{x_{i+1}}).\ldots.a_n^{\mathtt{free}}(\widetilde{x_n}).P_1 : \emptyset,$$

$\Gamma, \mathcal{R}'' \vdash^+_{\mathtt{po}} P_1 : M_1$, $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} P_2 : M_2$, *knowing that for all* j, $\Gamma(a_i) = \sharp^{k_j}_{\mathtt{R}_{a_j}} \widetilde{T_j}$, $\Gamma(\widetilde{v}) = \widetilde{T_i}$, $\Gamma(\widetilde{x_j}) = \widetilde{T_j}$, *that* $\mathcal{R}' = \mathtt{R}_{a_1}\lfloor\widetilde{x_1}\rfloor \uplus \cdots \uplus \mathtt{R}_{a_n}\lfloor\widetilde{x_n}\rfloor \uplus \mathcal{R}''$, $\{a_1, \ldots, a_n\}\mathtt{mul}_{(>_{1\mathrm{v}1}, \mathcal{R}')}M$, $\mathtt{R}_{a_1} \subseteq \mathcal{R}'$ *and* $M = M_2 \uplus \{a_1\}$ *and that the condition* $\mathbf{safe}(\{a_1, \ldots, a_n\}, M, P)$, *holds. Fact 3.3.16 gives* $\mathbf{AvRes}^{po}(P) = \mathbf{AvRes}^{po}(\mathbf{E}) \uplus \{a_i\} \uplus \mathbf{AvRes}(P_2) \uplus \{a_1\} \uplus \cdots \uplus \{a_{i-1}\}$. *Lemma 3.1.8 gives* $\mathbf{Os}(P) = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_2) \uplus \{a_i\}$. *As* $\mathtt{R}_{a_i}\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R}' \subseteq (\mathcal{R}' \uplus \mathtt{R}_{a_1}\lfloor\widetilde{x_1}\rfloor \uplus \cdots \uplus \mathtt{R}_{a_n}\lfloor\widetilde{x_n}\rfloor)$ *and* $\mathcal{R}'' = (\mathcal{R}' \uplus \mathtt{R}_{a_2}\lfloor\widetilde{x_2}\rfloor \uplus \ldots) \uplus \mathtt{R}_{a_i}\lfloor\widetilde{x_i}\rfloor$, *and as the Barendregt convention gives* $\widetilde{x_1} \cap \{a_1\} = \emptyset$, *Lemma 3.3.13 gives*

$$\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} !a_1^{\mathtt{ok}}(\widetilde{x_1}).\ldots.a_{i-1}^{\mathtt{ok}}(\widetilde{x_{i-1}}).a_i^{\mathtt{ok}}(\widetilde{x_i}).((a_{i+1}^{\mathtt{free}}(\widetilde{x_{i+1}}).\ldots.a_n^{\mathtt{free}}(\widetilde{x_n}).P_1)\{\widetilde{v}/\widetilde{x_i}\}) : \emptyset.$$

Using (**PoPar**) *we derive*

$$\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} !a_1^{\mathtt{ok}}(\widetilde{x_1}).\ldots.a_{i-1}^{\mathtt{ok}}(x_{i-1}).a_i^{\mathtt{ok}}(\widetilde{x_i}).((a_{i+1}^{\mathtt{free}}(x_{i+1}).\ldots.a_n^{\mathtt{free}}(\widetilde{x_n}).P_1)\{v/\widetilde{x_i}\}) \mid P_2 : \emptyset \uplus M_2$$

and Lemma 3.2.11 gives $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P' : N'$ *for some* N'. *Fact 3.2.13 gives* $\mathbf{AvRes}(P') = \mathbf{AvRes}(\mathbf{E}) \uplus \mathbf{AvRes}(!a_1^{\mathtt{ok}}.\ldots.a_{i-1}^{\mathtt{ok}}(x_{i-1}).a_i^{\mathtt{ok}}(x_i).((a_{i+1}^{\mathtt{free}}(x_{i+1}).\ldots.a_n^{\mathtt{free}}(x_n).P_1)\{v/x_i\})) \uplus \mathbf{AvRes}(P_2)$. *From Definition 3.2.12 and Fact 3.2.14 we have*

$$\mathbf{AvRes}^{po}(!a_1^{\mathtt{ok}}.\ldots.a_{i-1}^{\mathtt{ok}}(x_{i-1}).a_i^{\mathtt{ok}}(x_i).((a_{i+1}^{\mathtt{free}}(x_{i+1}).\ldots.a_n^{\mathtt{free}}(x_n).P_1)\{v/x_i\})) = \{a_1\} \uplus \cdots \uplus \{a_{i-1}\} \uplus \{a_i\}$$

. From Lemma 3.1.8 and Fact 3.1.10 we get $\mathbf{Os}(P') = \mathbf{Os}(\mathbf{E}) \uplus \mathbf{Os}(P_2)$. *We conclude, as we proved* $\mathbf{AvRes}^{po}(P) = \mathbf{AvRes}^{po}(P')$ *and* $\mathbf{Os}(P') \subsetneq \mathbf{Os}(P)$.

- *Case* (**KFire**). *We have*

$$P = \mathbf{E}[!a_1^{\mathtt{ok}}(\widetilde{x_1}).\ldots.a_{i-1}^{\mathtt{ok}}(\widetilde{x_{i-1}}).a_n^{\mathtt{free}}(\widetilde{x_n}).P_1 \mid \overline{a_n}\langle\widetilde{v}\rangle.P_2]$$

and $P' = \mathbf{E}[P_1\{\widetilde{v}/\widetilde{x_n}\} \mid P_2]$. *From* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} P : N$ *and Lemma 3.3.14, we have*

$$\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} !a_1^{\mathtt{ok}}.\ldots.a_{i-1}^{\mathtt{ok}}(x_{i-1}).a_n^{\mathtt{free}}(x_n).P_1 \mid \overline{a_n}\langle v\rangle.P_2 : M$$

for some $M \leq N$ *and* $\mathcal{R}' \subseteq \mathcal{R}_{\blacktriangleright(P)}$. *Then, we derive the following judgements:*

$$\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} !a_1^{\mathtt{ok}}.\ldots.a_{i-1}^{\mathtt{ok}}(x_{i-1}).a_n^{\mathtt{free}}(x_n).P_1 : \emptyset,$$

$\Gamma, \mathcal{R}'' \vdash^+_{\mathtt{po}} P_1 : M_1$, $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} P_2 : M_2$ *and we know that for all* j, $\Gamma(a_j) = \sharp^{k_j}_{\mathtt{R}_{a_j}} \widetilde{T_j}$, $\Gamma(\widetilde{x_j}) = \widetilde{T_j}$, *that* $\mathcal{R}' = \mathtt{R}_{a_1}\lfloor\widetilde{x_1}\rfloor \uplus \cdots \uplus \mathtt{R}_{a_n}\lfloor\widetilde{x_n}\rfloor \uplus \mathcal{R}''$, $\{a_1, \ldots, a_n\}\mathtt{mul}_{(>_{1\mathrm{v}1}, \mathcal{R}')}M$, , $\mathtt{R}_{a_1} \subseteq \mathcal{R}'$, $M = M_2 \uplus \{a_1\}$ *and that the condition* $\mathbf{safe}(\{a_1, \ldots, a_n\}, M, P)$ *holds. Fact 3.3.16 gives* $\mathbf{AvRes}^{po}(P) = \mathbf{AvRes}^{po}(\mathbf{E}) \uplus$

$\{a_n\} \uplus \mathbf{AvRes}^{po}(P_2) \uplus \{a_1\} \uplus \cdots \uplus \{a_{n-1}\}$. *Using* (**PoPar**) *we derive* $\Gamma, \mathcal{R}' \vdash^+_{\mathtt{po}} P_1\{\widetilde{v}/\widetilde{x_n}\} \mid P_2 : M_1\{\widetilde{v}/\widetilde{x_n}\} \uplus M_2$ *and Lemma 3.3.14 gives* $\Gamma, P' \vdash^+_{\mathtt{po}} N' :$ *for some* N'. *Fact 3.3.16 gives* $\mathbf{AvRes}(P') = \mathbf{AvRes}(\mathbf{E}) \uplus \mathbf{AvRes}(P_1\{v/x_n\}) \uplus \mathbf{AvRes}(P_2)$. *As* P_1 *is guarded by a prefix in* P, *we know that every annotation in* P_1 *is* `free`, *we can apply Fact 3.3.19 to get* $\mathbf{AvRes}(P_1\{v/x_n\}) = M_1\{\widetilde{v}/\widetilde{x_n}\}$. *The condition* $\{a_1, \ldots, a_n\}\mathtt{mul}_{(>_{1\mathrm{v}1}, \mathcal{R}')}M_1$, *the fact that* $\{a_1, \ldots, a_n\} \cap \widetilde{x_n} = \emptyset$, *and Lemma 3.3.10 allow us to conclude, as* $\mathcal{R}' \subseteq \mathcal{R}_{\blacktriangleright(P)}$.

□

The termination proof is not over yet, as we have to prove that the measure we use is well-founded. More precisely, we have to prove that the partial order used in the definition of this measure is well-founded, that is, in our case, it will be enough to show that its domain is finite. This is related to the safety condition, preventing an infinite number of names from being created via restriction.

Deriving a contradiction Additional notions are necessary in order to prove the well-foundedness of the partial order. We first define the level of a reduction, which is the level of the channel on which the communication is performed, except when a replicated input sequence is fired. In this case this is the maximum decreasing level, as defined along the safety condition.

Definition 3.3.21 (Level of reduction)
If $P \to P'$, *we say that* P performs a reduction at level n into P', *written* $P \to^n P'$ *when:*

1. *either the rule* (**KComm**) *is used in the reduction derivation with* $\mathbf{E}[a(x).P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$ *and* $\Gamma(a) = \sharp^n_{\mathtt{R}_a}\widetilde{T}$.
2. *or the rule* (**KTrig**) *is used in the reduction derivation with* $\mathbf{E}[!a(\widetilde{x_1}) \ldots P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$ *and* $\Gamma(a) = \sharp^n_{\mathtt{R}_a}\widetilde{T}$.
3. *or the rule* (**KProg**) *is used in the reduction derivation with* $\mathbf{E}[!a_1(\widetilde{x_1}) \ldots P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$ *and* $\Gamma(a) = \sharp^n_{\mathtt{R}_a}\widetilde{T}$.
4. *or the rule* (**Kfire**) *is used in the reduction derivation with* $\mathbf{E}[!a_1(\widetilde{x_1}) \ldots P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$ *and the multiset comparison* $M_1\mathtt{mul}_{(>_{1\mathrm{v}1}, \mathcal{R})}M_2$ *used to type* $!a_1(\widetilde{x_1}) \ldots$ *is such that when* $M_1 = N \uplus N_1$, $M_2 = N \uplus N_2$ *with* N *maximal, then the maximum level of an element* $e_1 \in M_1$ *is* n.

In the remainder of this section, we consider, towards a contradiction, an infinite reduction sequence $(P_i)_{i\in\mathbb{N}}$ starting from a typable term $P = P_0$. First we define the partial order we will use in the soundness proof, as the union of all $\mathcal{R}_{\blacktriangleright(P_i)}$.

Definition 3.3.22
We call $\mathcal{R}_\infty$ *the partial order obtained by* $\mathcal{R}_\infty = \bigcup_i \mathcal{R}_{\blacktriangleright(P_i)}$.

Notice that $\mathcal{R}_\infty$ is a partial order, as we can easily prove that $\mathcal{R}_{\blacktriangleright(P_i)} \subseteq \mathcal{R}_{\blacktriangleright(P_j)}$ when $i < j$. It is easy to see that, for all i, $(\mathbf{AvRes}^{\mathtt{po}}(P_i), \mathbf{Os}(P_i))\mathtt{Lex}(\mathtt{mul}_{(>_{1\mathrm{v}1}, \mathcal{R}_\infty)}, \subsetneq)(\mathbf{AvRes}^{\mathtt{po}}(P_{i+1}), \mathbf{Os}(P_{i+1}))$.

We write $\mathcal{R}|_{(n)}$ to denote the restriction of $\mathcal{R}$ to the level n, which is the maximum partial order $\mathcal{R}' \subseteq \mathcal{R}$ whose domain contains only names at level n and $\mathcal{R}|_{(>n)}$ to denote the restriction of $\mathcal{R}$ to the levels $> n$, which is the maximum partial order $\mathcal{R}' \subseteq \mathcal{R}$ whose domain contains only names at level $> n$.

Similarly, if M is a multiset of names, we write $M|_{(n)}$ to denote the maximum sub-multiset of M containing only names at level n, and $M|_{(>n)}$ to denote the maximum sub-multiset at M containing only names of level $> n$.

Fact 3.3.23 (Maximum interesting level)
Suppose $\mathcal{R}_\infty$ *has an infinite domain. Then there exists* p *maximum, called the* maximum interesting level, *s.t.* $\mathcal{R}_\infty|_{(p)}$ *has an infinite domain.*

Proof. *Easy, as the number of levels involved in the domain of* $\mathcal{R}_\infty$ *is finite.* □

Fact 3.3.24 (Existence of infinite reduction)
Suppose that p is the maximum interesting level associated to an infinite computation sequence, then this derivation contains an infinite number of reductions on a level $> p$.

Proof.
Suppose it is not the case, then there is I, s.t. every reduction $P_i \to^n P_{i+1}$ for $i > I$ is s.t. $n \leq p$. By examining each reduction rule, we prove by recurrence that $\mathcal{R}_{\blacktriangleright(P_i)}|_{(p)}$ is the same for each $i > I$. The interesting case is (**KFire**)*. Here the condition* $\mathbf{safe}(\cdot,\cdot,\cdot)$ *ensures that no restriction on level $\geq p$ appears, and thus, the ordering $\mathcal{R}_\infty|_{(p)}$ cannot be extended.* □

Termination on higher levels We define a new measure here, to prove that the infinite computation cannot contain an infinite number of reductions on levels greater than the maximum interesting level. The measure first compares levels, then the numbers of variables in the two multisets, then the decreasing of the ordering, and finally, the number of outputs.

Definition 3.3.25 (Variables) *If x is a name and P a process, we say that x is a variable if x is bound by a input action in P.*

Definition 3.3.26 (Ordering on higher levels) *Given an ordering $\mathcal{R}$ clear from context. We use $\triangleright$ to denote an associated ordering on 3-uples $(M,n)_P$ composed of one natural number, one multiset of names and one process. Moreover, we only consider such pairs where $M = \mathbf{AvRes}^{po}(P)|_{(>p)}$ with p clear from context. The ordering is defined by the following: $(M_1,n_1)_{P_1} \triangleright (M_2,n_2)_{P_2}$ if:*

1. *either $M_1 >_{\mathtt{lvl}} M_2$,*
2. *or $\mathtt{lvl}(M_1) = \mathtt{lvl}(M_2)$ but there are more variables in M_1 (bound in P_1) than in M_2 (bound in P_2),*
3. *or $\mathtt{lvl}(M_1) = \mathtt{lvl}(M_2)$, the number of variables is the same in M_1 and M_2, but $M_1 \mathcal{R}_\infty^{mul}|_{(>p)} M_2$ where $\mathcal{R}_\infty^{mul}|_{(>p)}$ is the standard multiset extension of $\mathcal{R}|_{(>p)}$, the restriction of $\mathcal{R}$ to names of level p or higher.*
4. *or the multisets M_1 and M_2 are equal but $n_1 > n_2$.*

If P is clear from context, we write (M,n) for $(M,n)_P$.

For instance, if $p = 0$, $a_1 \mathcal{R} a_2$, $P_1 = a(x).\overline{x} \mid \overline{y}$, $P_2 = a(x).\overline{x} \mid b(y).\overline{y}$, then:

- $(\{x^1, y^2, a^4\}, 4)_{(P_1 \mid \overline{a})} \triangleright (\{x^1, y^2, b^3\}, 3)_{(P_2 \mid \overline{b})}$ by level comparison,
- $(\{x^1, y^2\}, 4)_{P_1} \triangleright (\{x^1, y^2\}, 3)_{P_2}$ as the multisets of levels are the same but y is bound in P_1 and thus, P_1 has more variable.
- $(\{x^1, y^2, a_1^4\}, 4)_{(P_1 \mid \overline{a_1})} \triangleright (\{x^1, y^2, a_2^4\}, 3)_{(P_1 \mid \overline{a_2})}$ as the multisets of levels are the same, the number of variables are the same, but $a_1 \mathcal{R} a_2$,
- $(\{x^1, y^2\}, 4)_{P_1} \triangleright (\{x^1, y^2\}, 3)_{P_1}$, as the two multisets are equal with respect to the three first comparisons but $4 > 3$.

Fact 3.3.27 (Well-foundedness of $\triangleright$)
If $\mathcal{R}_\infty|_{(>p)}$ is well-founded, then $\triangleright$ is well-founded.

Proof. *Directly obtained from Lemma 2.1.3.* □

The following lemma states that the measure decreases when a reduction takes place on higher levels and decreases, or stays the same, when a reduction takes place on lower levels.

Lemma 3.3.28 (Reductions and decreasing)

1. *If* $P \to^n P'$ *with* $n > p$, *then* $(\mathbf{AvRes}^{po}(P)|_{(>p)}, \mathbf{Os}(P)|_{(>p)}) \triangleright (\mathbf{AvRes}^{po}(P')|_{(>p)}, \mathbf{Os}(P)|_{(>p)})$.
2. *If* $P \to^n P'$ *with* $n < p$, *then either* $(\mathbf{AvRes}^{po}(P)|_{(>p)}, \mathbf{Os}(P)|_{(\geq p)}) = (\mathbf{AvRes}^{po}(P')|_{(>p)}, \mathbf{Os}(P)|_{(>p)})$ *or* $(\mathbf{AvRes}^{po}(P)|_{(>p)}, \mathbf{Os}(P)|_{(>p)}) \triangleright (\mathbf{AvRes}^{po}(P')|_{(>p)}, \mathbf{Os}(P)|_{(>p)})$.

Proof.

1. *By induction on the reduction derivation.*

   - (**KCong**). *Easily done as* $\mathbf{AvRes}^{po}(\cdot)|_{(>p)}$ *and* $\mathbf{Os}(\cdot)|_{(p)}$ *are stable by structural congruence.*
   - (**KComm**). *Suppose* $P \to^n P'$, $P = \mathbf{E}[a(\widetilde{x}).P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$. *We proceed the same way we did in the proof of Proposition 3.3.29 to get* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_1)|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(>p)}$ *and* $\mathbf{AvRes}^{po}(P')|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_1\{\widetilde{v}/\widetilde{x}\})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)}$ *and* $\{a\}|_{(>p)}$. *What we know is that* $\{\widetilde{v}/\widetilde{x}\}$ *leaves the level unchanged, so*

   $$\mathtt{lvl}(\mathbf{AvRes}^{po}(P)|_{(>p)}) = \mathtt{lvl}(\mathbf{AvRes}^{po}(P')|_{(>p)}).$$

   Either there is a name of level $> p$ *in* $\widetilde{x}$ *and in this case, it is easy to see that the number of variables decreases and we conclude, or there is no such name. In the latter case, we have* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(P')|_{(>p)}$ *and we conclude with* $\mathbf{Os}(P)|_{(>p)} = 1 + \mathbf{Os}(P')$, *as* $\mathtt{lvl}(a) = n$.
   - (**KTrig**). *Suppose* $P \to^n P'$, $P = \mathbf{E}[!a(\widetilde{x})^{\mathtt{free}}.\ldots.a_k(\widetilde{x_k})^{\mathtt{free}}.P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$. *We proceed the same way we did in the proof of Proposition 3.3.29 to get* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(p)}$ *and* $\mathbf{AvRes}^{po}(P')|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \cdots \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(>p)}$ *which is* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(P')|_{(>p)}$. *We conclude with* $\mathbf{Os}(P)|_{(>p)} = 1 + \mathbf{Os}(P')$, *as* $\mathtt{lvl}(a) = n$
   - (**KProg**). *Suppose* $P \to^n P'$, $P = \mathbf{E}[!a_1(\widetilde{x_1})^{\mathtt{ok}}.\ldots.a(\widetilde{x})^{\mathtt{free}}.\ldots.a_k(\widetilde{x_k})^{\mathtt{free}}.P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$. *We proceed the same way we did in the proof of Proposition 3.3.29 to get*

   $$\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(>p)} \uplus \{a_1\}|_{(>p)} \uplus \ldots$$

   and

   $$\mathbf{AvRes}^{po}(P')|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \cdots \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(>p)} \uplus \{a_1\}|_{(>p)} \uplus \ldots$$

   which is $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(P')|_{(>p)}$. *We conclude with* $\mathbf{Os}(P)|_{(>p)} = 1 + \mathbf{Os}(P')$, *as* $\mathtt{lvl}(a) = n$.
   - (**KFire**). *Suppose* $P \to^n P'$, $P = \mathbf{E}[!a_1(\widetilde{x_1})^{\mathtt{ok}}.\ldots.a(\widetilde{x})^{\mathtt{free}}.P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$. *We proceed the same way we did in the proof of Proposition 3.3.29 to get* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(p)} \uplus \{a_1\}|_{(>p)} \uplus \ldots$ *and* $\mathbf{AvRes}^{po}(P')|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \cdots \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_1)\{\widetilde{v}/\widetilde{x}\}$. *Definition 3.3.21 gives that the multiset comparison* $\{a_1\} \uplus \cdots \uplus \{a\}\mathtt{mul}_{(>_{\mathtt{lvl}},\mathcal{R})} M_1$ *used to type* $!a_1(\widetilde{x_1})\ldots$ *is s.t. when* $\{a_1\} \uplus \cdots \uplus \{a\} = N \uplus N_1$, $M_2 = N \uplus N_2$ *with* N *maximal, then the maximum level of an element of* N_1 *is* n. *We apply Fact 3.3.19 to derive* $\{a_1\} \uplus \cdots \uplus \{a\}\mathtt{mul}_{(>_{\mathtt{lvl}},\mathcal{R})}\mathbf{AvRes}^{po}(P_1\{\widetilde{v}/\widetilde{x}\})$. *As levels are unaffected by* $\{\widetilde{v}/\widetilde{x}\}$, *the fact that the reduction takes place on level* $n > p$ *implies that* $\mathbf{AvRes}^{po}(P)|_{(>p)} \neq \mathbf{AvRes}^{po}(P')|_{(>p)}$. *From the proof of Proposition 3.3.29, either* $\{a_1\} \uplus \cdots \uplus \{a_n\} >_{\mathtt{lvl}} \mathbf{AvRes}^{po}(P_1)$, *and we have* $\mathbf{AvRes}^{po}(P)|_{(>p)} >_{\mathtt{lvl}} \mathbf{AvRes}^{po}(P')|_{(>p)}$ *and we conclude, or* $\{a_1\} \uplus \cdots \uplus \{a_n\} \mathcal{R}_{mul} \mathbf{AvRes}^{po}(P_1)$ *and in this case there is no variable in* $\mathbf{AvRes}^{po}(P_1)$ *(if* $x \in \mathbf{AvRes}^{po}(P_1)$ *is a variable,* x *cannot be related to a name of* $\{a_1\} \uplus \cdots \uplus \{a\}$ *by* $\mathcal{R}$*) so we conclude.*

2. *By induction on the reduction derivation.*

- (**KCong**). *Easily done as* $\mathbf{AvRes}^{po}(\cdot)|_{(>p)}$ *and* $\mathbf{Os}(\cdot)|_{(>p)}$ *are stable by structural congruence.*
- (**KComm**). *Suppose* $P \to^n P'$, $P = \mathbf{E}[a(\widetilde{x}).P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$. *We proceed the same way we did in the proof of Proposition 3.3.29 to get*

  $$\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_1)|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(>p)}$$

  and

  $$\mathbf{AvRes}^{po}(P')|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_1\{\widetilde{v}/\widetilde{x}\})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)}.$$

  What we know is that $\{\widetilde{v}/\widetilde{x}\}$ *leaves the level unchanged, so*

  $$\mathtt{lvl}(\mathbf{AvRes}^{po}(P)|_{(>p)}) = \mathtt{lvl}(\mathbf{AvRes}^{po}(P')|_{(>p)}).$$

  Either there is a name of level $> p$ *in* $\widetilde{x}$ *and in this case, it is easy to see that the number of variables decreases and we conclude, or there is no such name. In the latter case we have* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(P')|_{(>p)}$ *and we conclude as* $\mathbf{Os}(P)|_{(>p)} = \mathbf{Os}(P')$ *as* $\mathtt{lvl}(a) = n$.
- (**KTrig**). *Suppose* $P \to^n P'$, $P = \mathbf{E}[!a(\widetilde{x})^{\mathtt{free}}.\ldots.a_k(\widetilde{x_k})^{\mathtt{free}}.P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$. *We proceed the same way we did in the proof of Proposition 3.3.29 to get* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(>p)}$ *and* $\mathbf{AvRes}^{po}(P')|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \cdots \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(>p)}$ *which is* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(P')|_{(>p)}$. *We conclude with* $\mathbf{Os}(P)|_{(>p)} = \mathbf{Os}(P')$, *as* $\mathtt{lvl}(a) = n$
- (**KProg**). *Suppose* $P \to^n P'$, $P = \mathbf{E}[!a_1(\widetilde{x_1})^{\mathtt{ok}}.\ldots.a(\widetilde{x})^{\mathtt{free}}.\ldots.a_k(\widetilde{x_k})^{\mathtt{free}}.P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$. *We proceed the same way we did in the proof of Proposition 3.3.29 to get*

  $$\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(>p)} \uplus \{a_1\}|_{(>p)} \uplus \ldots$$

  and $\mathbf{AvRes}^{po}(P')|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(p)} \uplus \{a_1\}|_{(>p)} \uplus \ldots$ *which is* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(P')|_{(>p)}$. *We conclude with* $\mathbf{Os}(P)|_{(>p)} = \mathbf{Os}(P')$, *as* $\mathtt{lvl}(a) = n$.
- (**KFire**). *Suppose* $P \to^N P'$, $P = \mathbf{E}[!a_1(\widetilde{x_1})^{\mathtt{ok}}.\ldots.a(\widetilde{x})^{\mathtt{free}}.P_1 \mid \overline{a}\langle\widetilde{v}\rangle.P_2]$. *We proceed the same way we did in the proof of Proposition 3.3.29 to get* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \{a\}|_{(>p)} \uplus \{a_1\}|_{(>p)} \uplus \ldots$ *and* $\mathbf{AvRes}^{po}(P')|_{(>p)} = \mathbf{AvRes}^{po}(\mathbf{E})|_{(>p)} \uplus \cdots \uplus \mathbf{AvRes}^{po}(P_2)|_{(>p)} \uplus \mathbf{AvRes}^{po}(P_1)\{\widetilde{v}/\widetilde{x}\}$. *Definition 3.3.21 gives that the multiset comparison* $\{a_1\} \uplus \cdots \uplus \{a\}\mathtt{mul}_{(>_{\mathtt{lvl}},\mathcal{R})}M_1$ *used to type* $!a_1(\widetilde{x_1})\ldots$ *is s.t. when* $\{a_1\} \uplus \cdots \uplus \{a\} = N \uplus N_1$, $M_2 = N \uplus N_2$ *with* N *maximal, the maximum level of an element of* N_1 *is* $n < p$. *Thus,* $\mathbf{AvRes}^{po}(P)|_{(>p)} = \mathbf{AvRes}^{po}(P')|_{(>p)}$.

□

Soundness The previous results allow us to raise a contradiction: on the higher levels the measure decreases, so there is only a finite number of reductions on higher levels. However, an infinite number of reductions on higher levels is required in order for an infinite number of reductions on the maximum interesting level to take place.

Proposition 3.3.29 (Soundness)
If $\Gamma, \mathcal{R} \vdash_{\mathtt{po}} P : M$, *then* P *terminates.*

Proof.
If $\Gamma, \mathcal{R} \vdash_{\mathtt{po}} P : M$ *and* P *diverges, then by Fact 3.3.8* $\Gamma, \mathcal{R} \vdash^+_{\mathtt{po}} \mathtt{free}(P) : M$ *and by Lemma 3.2.7* $\mathtt{free}(P)$ *diverges. We consider an infinite sequence of reductions from* $\mathtt{free}(P)$. *Thus there exists a maximum interesting level* p *as defined in Fact 3.3.23 and an infinite number of times* $P_i \to^N P_{i+1}$. *It follows from the definition of* p *that* $\mathcal{R}_\infty|_{(>p)}$ *is well-founded. We raise a contradiction, as Lemma 3.3.28 ensures that the measure* $(\mathbf{AvRes}^{po}(P)|_{(>p)}, \mathbf{Os}(P)|_{(>p)})$ *decreases an infinite number of times for the well-founded (see Fact 3.3.27) ordering* $\triangleright$.

□

3.4 Typing terminating inductive data structures

We present in this section results from [DHS08]. The previous type systems allow one to build π-processes modelling list data structures, that is structures where requests can be sent from one node to another. Therefore, it is natural to wonder if such methods can be applied to ensure termination of any well-founded inductive data-structures such as trees or directed acyclic graphs where requests can be propagated simultaneously in different branches. Previous systems validate terminating servers trading a request for another (somehow smaller for a given partial order) one; here, we present a way to validate terminating servers trading a requests for several ones.

3.4.1 A motivating example

We recall here the basic ideas behind the type system of Section 3.3 (called $\mathbf{S}^{\mathrm{ord}}$), using an example that also illustrates some of the limitations of this system on recursive structures. The example is about the implementation of a symbol table as a binary tree. Each node in the tree is a simple π-calculus process. The process T_0 below is the generator of nodes. An output $\overline{node}\langle a, l, r, s, e\rangle$ produces a node that stores a string s whose key is e, that is connected to its parent node (or to the environment, in the case of the root node) with name a, and to its children nodes with names l and r. A tree at a (that is, a tree whose root uses a for interactions with the outside) is searched for a value v via requests of the form $\overline{a}\langle \mathtt{search}, v, ans\rangle$ where ans is a return channel. When the search reaches a node, if the value is found in the node, then the corresponding key is sent back on ans; otherwise the request is concurrently propagated to both subtrees of the node. (We omit the details of a search operation that fails.)

$$
\begin{array}{rcl}
T_0 & \stackrel{\mathrm{def}}{=} & !node(a,l,r,s,e).a(mode,v,ans). \\
& & \quad \mathtt{if}\ mode = \mathtt{search}\ \mathtt{then} \\
& & \quad\quad \mathtt{if}\ v = s\ \mathtt{then}\ \ \overline{ans}\langle e\rangle \mid \overline{node}\langle a,l,r,s,e\rangle \\
& & \quad\quad \mathtt{else}\ \ \ \overline{l}\langle mode,v,ans\rangle \mid \overline{r}\langle mode,v,ans\rangle \mid \overline{node}\langle a,l,r,s,e\rangle \\
& & \quad \mathtt{else}\ \ldots
\end{array}
$$

The type system of Section 3.3 recognises a system as terminating if the continuations activated in replications have a smaller "weight" than that of the output that has been consumed to trigger the interaction. Now, consider the system composed by a tree at a and a search request $\overline{a}\langle \mathtt{search}, v, ans\rangle$. Names a, l and r play the same role in the structure, and therefore must have the same level (this is something we already explained). As the consumption of the output at a may produce outputs at l and r (the 'else' branch in T_0), the overall weight of the system increases. Due to this increase, T_0 is not typable using $\mathbf{S}^{\mathrm{ord}}$. Indeed, $\mathbf{S}^{\mathrm{ord}}$ allows the weight (the multisets of levels) of the derivatives of an interaction to be at most the same as that of the initial process, and for this it relies on a rudimentary partial order information on names; however, the weight may never increase, as is instead the case for T_0.

In the new type system that we propose below, replications in which the weight increases may be typed (indeed T_0 is typable, see Section 3.4.3). The greater expressiveness is achieved by enforcing a tight coupling between the weight and a well-founded partial order. In other words, we use here a more subtle definition for the order comparing the multiset of the names of an input sequence and the weight of its continuation. Increases in weight through reductions are possible, provided they are appropriately compensated in the partial order. This schema, while intuitively simple, is rather delicate. As an example of the possible problems (other examples will be given later), consider the system

$$
T_1 \stackrel{\mathrm{def}}{=} \overline{u} \,|\, \overline{v} \,|\, U_1 \,|\, U_2 \qquad \text{with} \qquad \left\{ \begin{array}{l} U_1 \stackrel{\mathrm{def}}{=} !p(a,b,c).a.(\overline{b} \mid \overline{c}) \\ U_2 \stackrel{\mathrm{def}}{=} !u.v.(\overline{w} \mid \overline{p}\langle w,u,v\rangle) \end{array} \right.
$$

where names w, u, v have the same level k and p has level $k' < k$ (this can be imposed, e.g., by adding extra processes in parallel). In U_1, the weight increases underneath the initial inputs at p and a; but the new outputs are smaller in the partial order, if we set a above b and c. In U_2, the weight decreases underneath the

$$(\textbf{NPoRep})\ \frac{\Gamma, \mathcal{R}' \vdash_{\texttt{po+}} P : M \qquad \Gamma(a_i) = \sharp^{k_i}_{\mathtt{R}_{a_i}} \widetilde{T_i} \qquad \Gamma(\widetilde{x_i}) = \widetilde{T_i} \qquad \mathcal{R}' = \mathtt{R}_{a_1}\lfloor\widetilde{x_1}\rfloor \uplus \cdots \uplus \mathtt{R}_{a_n}\lfloor\widetilde{x_n}\rfloor \uplus \mathcal{R} \quad \{a_1, \ldots, a_n\} >^{\mathtt{T}}_{\mathcal{R}} M \qquad \textbf{safe}(\{a_1, \ldots, a_n\}, M, P)}{\Gamma, \mathcal{R} \vdash_{\texttt{po+}} !a_1(\widetilde{x_1}).\ldots.a_n(\widetilde{x_n}).P : \emptyset}$$

Figure 3.7: Tying rule for replication for the new system with partial order

top two inputs. The system seems to meet the termination conditions (which we define rigorously below); however, it does not terminate (the outputs at u and v trigger U_2, which in turn triggers U_1 and we are back to T_1).

3.4.2 A more expressive typing rule for replication

The grammar for types is the same as the one in the previous section. Typing judgements are of the form $\Gamma, \mathcal{R} \vdash_{\texttt{po+}} P : M$, as previously. Typing rules for inactive process, restrictions, parallel compositions, non-replicated inputs and outputs are the ones found in the previous section, the replication typing rule is replaced by (**NPoRep**), found in Figure 3.7.

This rule is very similar to the old one, in particular the condition $\textbf{safe}(\cdot,\cdot,\cdot)$ remains the same. Yet, the comparison between the two multisets changes and is defined as follows (we recall that $M|_{(l)}$, where M is a multiset of names and l an integer is defined with respect to a typing context Γ by the maximum multiset included in M containing only names of level l). $\mathcal{R}_{mul}$ is the (strict) multiset extension of $\mathcal{R}$ (see Section 2).

We write $M_1 >^{\mathtt{T}}_{\mathcal{R}} M_2$ when there exists l s.t.

1. $M_1|_{(>l)} = M_2|_{(>l)}$
2. and $M_1|_{(l)} \mathcal{R}_{mul} M_2|_{(l)}$.

This means that $M_1 >^{\mathtt{T}}_{\mathcal{R}} M_2$ when there exists a level l such that the names at levels $> l$ are the same in M_1 and M_2 and that the names at level l in M_1 dominates the names of level l in M_2, either because of the partial order $\mathcal{R}$ or because of a strict inclusion. Indeed, if the names in $M_1|_{(l)}$ and $M_2|_{(l)}$ are not related by partial order, the only way to satisfy the comparison $M_1|_{(l)} \mathcal{R}_{mul} M_2|_{(l)}$ is to have $M_1|_{(l)} \supsetneq M_2|_{(l)}$.

3.4.3 Examples

We present one example that illustrates some of the technicalities of the type system, more precisely we explain how we rule out the system T_1 in Section 3.4.1 and how we type the motivating example presented above.

We have $T_1 = \overline{u} \,|\, \overline{v} \,|\, U_1 \,|\, U_2$ with $U_1 = !p(a,b,c).a.(\overline{b} \mid \overline{c})$ and $U_2 = !u.v.(\overline{w} \mid \overline{p}\langle w,u,v\rangle)$. This process diverges. We explain why it is rejected by our system, supposing, as we did in Section 3.4.1, that we must have $\mathtt{lvl}(a) > \mathtt{lvl}(p)$; e.g., p has level 1 and all other names have level 2.

For U_1, since the weight is increasing, we must resort to the partial order, and impose that the first component of uples transmitted on p dominates the two other components (so that name a dominates b and c). The typing of U_2 is then invalid, we cannot rely on the partial order to type the replication and the inclusion condition of $>^{\mathtt{T}}$ cannot be satisfied, as $M_\kappa|_2 = \{u, v\}$ does not dominate $\{w\}$ by the partial order (we have $w\mathcal{R}u$ and $w\mathcal{R}v$ from the typechecking of the output $\overline{p}\langle w,u,v\rangle$). It can be shown, more generally, that for any assignment of levels to names, T_1 cannot be typed.

Process T_0 presented in Section 3.4.1 can be type-checked, by assigning type T_a to names a, l, r, type T_{ans} to *ans*, and type T_{node} to *node*, with

$$T_a = \sharp^3(\mathtt{Tm}, \mathtt{Tval}, T_{ans})\,, \quad T_{ans} = \sharp^2(\mathtt{Tv})\,, \quad T_{node} = \sharp^1_{\{(1,2),(1,3)\}}(T_a, T_a, T_a, \mathtt{Tval}, \mathtt{Tv})\,,$$

where `Tval` is the type of the value v (strings in the example), `Tv` the type of the key associated to a value, `Tm` the type of tags indicating the method that is invoked on the tree. In the typing, the critical part is the 'else' branch in T_0; here the input on a at level 1 is traded for two outputs, on names l and r, at the same level, and we rely on the partial order derived from p to deduce the typing (a dominates both l and r). Note that at the higher level, level 3, the weight does not change, as the input at *node* is followed by an output on the same channel.

Remark 3.4.1 (Expressiviness with respect to the previous type system) *Even if the condition $>^{\mathtt{T}}_{\mathcal{R}}$ allows the typability of new terminating processes, such as T_0, there exist processes which were typable before which are no longer typable. Indeed, the definition of $>^{\mathtt{T}}_{\mathcal{R}}$ prevents simple terminating processes such as $!a.a.\bar{b}$ to be typed in an environment where a and b are given the same level and where a does not dominate b by the partial order. This contrasts with the previous systems where trading two outputs for one of the same level is considered as innocuous. Here, the condition $>^{\mathtt{T}}_{\mathcal{R}}$ forbids this and only the multiset extension of the ordering $\mathcal{R}$ is used. One one side, using this definition of $>^{\mathtt{T}}{}_{R}$ allows us to recognise as terminating the trade of one output for several outputs of the same level (when they are below for the partial order), on the other side we lost the ability to validate the trade of several outputs for fewer outputs of the same level, as this could compensate the partial-order (as in U_1).*

3.4.4 Soundness of the Type System

Theorem 3.4.2 *If $\Gamma, \mathcal{R} \vdash_{\mathtt{po+}} P : M$ then P terminates.*

Proof. *[Sketch]*

The proof of soundness is very similar to the one found in Section 3.3 and also makes use of an auxiliary calculus. Indeed, most of the previous lemmas still hold in this setting and the definition of available resources is the same. In the Subject Reduction proposition (counterpart to Proposition 3.3.20), we ensure that an effective decreasing takes place, using $>^{\mathtt{T}}_{\mathcal{R}_{\blacktriangleright(P)}}$.

The crucial point is that the comparison $>^{\mathtt{T}}_{\mathcal{R}_{\blacktriangleright(P)}}$ does not refer to multisets of levels, as the comparisons of the previous systems did. Thus, at the "maximum level that changes" (the level l in the definition of $>^{\mathtt{T}}_{\mathcal{R}}$), the number of elements on each side does not matter, only the partial order comparison does.

Then to derive a contradiction, we proceed as previously, using the completed partial order $\mathcal{R}_\infty$ corresponding to a derivation and proving that its support is finite, thanks to the definition of **safe**$(\cdot,\cdot,\cdot)$.

□

3.5 A hybrid (static/dynamic) analysis for termination

In this section we discuss a new approach to typing termination. Instead of relying solely on a type system, we present a mixed system in which the type checks are performed in two separated phases: a phase that precedes execution (as in the systems studied above), and the execution itself. Below, these two phases are referred to as *static* and *dynamic*, respectively; correspondingly we distinguish between static and dynamic typing.

The static typing, besides making the type checks, inserts into the processes *assertions* on names of the form $[a > b]$. Here, we call a process with assertions an *annotated process*. The grammar for annotated processes is the same as that of ordinary processes in Section 2, with the addition of the production $[a > b]P$ for assertions. We use $A, B, \ldots$ to range over annotated processes.

The assertions are needed in the dynamic typing, they are defined as relations between free names. At run time we check that the transitive closure of the assertions encountered during execution is well-founded. Thus the operational semantics is defined on *configurations* which are either pairs $(A, \mathcal{R})$ where A is an annotated process and $\mathcal{R}$ a partial order (as usual, represented by a set of pairs whose reflexive and transitive closure induces the partial order) or the *failure configuration* $\bot$.

$$\frac{\Gamma \vdash^{\mathtt{hy}} P : m \rightharpoonup A \qquad \Gamma(a) = \sharp^{l_a}\widetilde{T} \qquad \Gamma(v) = \widetilde{T}}{\Gamma \vdash^{\mathtt{hy}} \overline{a}\langle\widetilde{v}\rangle.P : \max(l_a, m) \rightharpoonup \overline{a}\langle\widetilde{v}\rangle.A} \qquad \frac{\Gamma \vdash^{\mathtt{hy}} P : m \rightharpoonup A \qquad \Gamma(a) = \sharp^{l_a}\widetilde{T} \qquad \Gamma(\widetilde{x}) = \widetilde{T}}{\Gamma \vdash^{\mathtt{hy}} a(\widetilde{x}).P : m \rightharpoonup a(\widetilde{x}).A}$$

$$\frac{\Gamma \vdash^{\mathtt{hy}} P_1 : m_1 \rightharpoonup A_1 \qquad \Gamma \vdash^{\mathtt{hy}} P_2 : m_2 \rightharpoonup A_2}{\Gamma \vdash^{\mathtt{hy}} P_1 \mid P_2 : \max(m_1, m_2) \rightharpoonup A_1 \mid A_2} \qquad \frac{\Gamma \vdash^{\mathtt{hy}} P : m \rightharpoonup A}{\Gamma \vdash^{\mathtt{hy}} (\boldsymbol{\nu} c)\, P : m \rightharpoonup (\boldsymbol{\nu} c)\, A} \qquad \frac{}{\Gamma \vdash^{\mathtt{hy}} \mathbf{0} : 0 \rightharpoonup \mathbf{0}}$$

$$\frac{\Gamma \vdash^{\mathtt{hy}} P : m \rightharpoonup A \qquad \Gamma(a) = \sharp^{l_a}\widetilde{T} \qquad \Gamma(\widetilde{x}) = \widetilde{T} \qquad l_a \geq m_1 \qquad l_a > \max(\mathtt{lvl}(\mathtt{Rs}(P))) \qquad A' = \mathtt{Inser}(A, a)}{\Gamma \vdash^{\mathtt{hy}} !a(\widetilde{x}).P : 0 \rightharpoonup !a(\widetilde{x}).A'}$$

Figure 3.8: The static type analysis in the mixed system

Failure in the dynamic checks occurs when the addition of a new assertion introduces a cycle; in this case the configuration $\bot$ is produced, meaning that an exception has been raised. We first define the dynamic system, and then the static system.

3.5.1 The dynamic system

The operational semantics on ordinary processes is extended to configurations as expected, and we write $\rightarrow$ for the reduction relation on configurations. The only new rule is the following:.

$$[a > b]A, \mathcal{R} \rightarrow \begin{cases} A, (\mathcal{R} \cup \{(a,b)\}) & \text{if } \mathcal{R} \cup \{(a,b)\} \text{ is a partial order} \\ \bot & \text{otherwise} \end{cases}$$

An annotated process A is *divergent* if there is an infinite sequence of reductions emanating from $(A, \emptyset)$ (where $\emptyset$ is the empty relation).

3.5.2 The static system

The static type system takes an ordinary process, performs some type checks on it, and returns an annotated process. Typing judgements are of the form $\Gamma \vdash^{\mathtt{hy}} P : m \rightharpoonup A$ where Γ is a typing context, P a process, m an integer and A an annotated process, they mean that P is typable in the context Γ with weight m and that his annotated version is A.

The rules are presented in Figure 3.8. As in Figure 3.7, the main part of the termination analysis is performed in the rule for replication. To type a replication $!a(\widetilde{x}).P$, we insert an assertion whenever we encounter an output in P that is not under a replication and whose subject has the same level as a; in this situation, levels alone are not sufficient to guarantee termination, and further checks, via the assertions, are postponed at run time.

We define $\mathtt{Rs}(P)$ as the set of all available (as usual, not guarded by a replication) restrictions in P. If A is an annotated process and a a name, then $\mathtt{Inser}(A, a)$ stands for the annotated process obtained from A by inserting an assertion $[a > b]$ in front of each available output whose subject name b has the same level as a. Intuitively, $[a > b]$ is a sanity check: a has to dominate b according to the partial order to guarantee that the process does not loop (see examples in Section 3.5.3). We write $\mathtt{lvl}(\mathtt{Rs}(P))$ for the sets of the levels of the names in $\mathtt{Rs}(P)$.

Remark 3.5.1 *In the rule for replication, only the initial input of the replication is examined. The system can be made more powerful by taking into account sequences of inputs, along the lines of the type system of Section 3.4, at the cost of a worse complexity for the inference problem (see Section 5).*

The following proposition says that a process and its annotated version perform the same reductions, unless the latter one raises an exception.

Proposition 3.5.2 (Simulation)
Suppose $\Gamma \vdash^{\mathtt{hy}} P : m \rightharpoonup A$. *If* $P \to^* P'$, *then:*

- *either* $A, \emptyset \to^* A', \mathcal{R}$ *with* $\mathbf{Er}(A') = P'$ *for some* $\mathcal{R}$,
- *or* $A, \emptyset \to^* \bot$.

Conversely, if $A, \emptyset \to^* A', \mathcal{R}$, *then* $P \to^* P'$ *for some* P' *with* $\mathbf{Er}(A') = P'$.

Proof.
Both results are easily proved by performing two inductions over the number of reductions first and over the reduction derivation then.

□

Using this result, we prove the subject reduction property:

Proposition 3.5.3 (Subject Reduction)
If $\Gamma \vdash^{\mathtt{hy}} P : m \rightharpoonup A$ *and* $(A, \mathcal{R}) \to (A', \mathcal{R}')$ *then* $\Gamma \vdash^{\mathtt{hy}} P' : m' \rightharpoonup A'$, $m \geq m'$ *and either* $P \to P'$ *or* $P = P'$.

Proof.
By induction on the reduction derivation, following the proof of Section 3.1.

□

Theorem 3.5.4 (Soundness of the hyrid type system)
If $\Gamma \vdash^{\mathtt{hy}} P : m \rightharpoonup A$, *then* A *has no diverging computation.*

Proof. *The proof of this result follows the same general strategy as the proofs of previous sections. As input sequences are not considered as a whole, the use of an auxiliary calculus is not necessary. However, the notion of configuration reduction has to be taken into account.*

Suppose there exists an infinite reduction sequence $(A_i, \mathcal{R}^i)_{i \in \mathbb{N}}$ *and that* $A_0 = A$ *and* $\mathcal{R}^0 = \mathcal{R}$. *We first use Proposition 3.5.3 to obtain an infinite pseudo-reduction sequence* $(P_i)_{i \in \mathbb{N}}$ *such that* $\forall i, \Gamma \vdash^{\mathtt{hy}} P_i : m_i \rightharpoonup A_i$ *and either* $P_i \to P_{i+1}$ *or* $P_i = P_{i+1}$.

It is easy to obtain that not only the weight m_i *does not increase along reduction (as stated in Proposition 3.5.3) but also that when the weight remains the same between* A_i *and* A_{i+1}, *either an annotation is consumed or a partial order annotation appears between the available outputs of* A_i *and the ones of* A_{i+1}.

Proposition 3.5.3 implies that there exists $I \in \mathbb{N}$ *such that* $\forall i > I, m_i = m_I$. *Consider such a reduction* $A_j, \mathcal{R}^j \to A_{j+1}, \mathcal{R}^{j+1}$. *Suppose that it does not correspond to the consumption of one annotation. As* $m_j = m_{j+1}$, *a partial order comparison exists between the multiset of the available outputs* M_j *of* A_j *and* M_{j+1} *of* A_{j+1}; *formally,* $M_j \mathcal{R}^j_{mul} M_{j+1}$. *We can define* $\mathcal{R}_\infty$ *as previously, as the union of all the relations* $\mathcal{R}_i$. *As a process contains only a finite number of annotations, there exists an infinite number of such steps* j *where the order decreases. This implies that* $\mathcal{R}_\infty$ *is not well-founded. Using a reasoning similar to the one we did in Section 3.3, we are able to use the safety condition present in the typing rule for replications to prove that* $\mathcal{R}_\infty$ *has a finite support. This implies that the relation* $\mathcal{R}_\infty$ *contains a cycle. As* $\mathcal{R}_i \subseteq \mathcal{R}_{i+1}$, *we derive the existence of* I' *such that* $\mathcal{R}_{I'}$ *contains a cycle, this contradicts the semantics rules.*

□

Remark 3.5.5 *We have to study carefully the configuration semantics in order to claim that the analysis we design is not trivial. Indeed, if we had propose a semantics where every configuration reduces to* $\bot$ *in one step, the previous property would still hold, but the system would not be interesting.*

Yet, in the solution we propose, a failure happens only when a dangerous loop in the partial order is detected. This means that if the level assignment is made as strict as possible (two names have the same level only if it must be, that is, only if they are used the same way), the system is at least as expressive as the system of Section 3.1.

3.5.3 Examples

The first example shows a divergent process that passes the static phase of the hybrid analysis and produces a failure exception at run time. Let

$$R \stackrel{\text{def}}{=} !p(a,b,c).(!a.\overline{b} \mid !b.\overline{c} \mid !c.\overline{a}) \mid \overline{p}\langle u,v,w\rangle \mid \overline{u} \ .$$

R is typable: we have a derivation for

$$\Gamma \vdash^{\text{hy}} R : m \multimap A \stackrel{\text{def}}{=} !p(a,b,c).(!a.[a>b].\overline{b} \mid !b.[b>c].\overline{c} \mid !c.[c>a].\overline{a}) \mid \overline{p}\langle u,v,w\rangle \mid \overline{u}$$

by assigning the same level to a, b, c. At run time we have the following (deterministic) sequence of reductions:

$$\begin{array}{rll}
(A,\emptyset) & \to\to & (A \mid !u.[u>v].\overline{v} \mid !v.[v>w].\overline{w} \mid !w.[w>u].\overline{u}) \mid [u>v].\overline{v},\emptyset) \\
& \to\to & (A \mid !u.[u>v].\overline{v} \mid !v.[v>w].\overline{w} \mid !w.[w>u].\overline{u}) \mid [v>w].\overline{w},\mathcal{R}_1) \\
& \to\to & (A \mid !u.[u>v].\overline{v} \mid !v.[v>w].\overline{w} \mid !w.[w>u].\overline{u}) \mid [w>u].\overline{u},\mathcal{R}_2) \\
& \to & \bot
\end{array}$$

where $\mathcal{R}_1$ is $\{(u,v)\}$ and $\mathcal{R}_2$ is $\{(u,v),(v,w)\}$. Process A eventually produces $\bot$ as the three inner replications create a cycle in the relation.

The following example present how we can type process modelling mutable inductive data structures. Let

$$Q \ = \ (Q_0 \mid \overline{p}\langle u,v\rangle \mid \overline{u}) \qquad \text{where} \qquad Q_0 \ = \ !p(a,b).(!a.\overline{b} \mid (\nu g)\ (g.!b.\overline{a} \mid g.b \mid \overline{g})) \ .$$

When the output at p is consumed we obtain the process

$$Q' \ = \ Q_0 \mid !u.\overline{v} \mid (\nu g)\ (g.!v.\overline{u} \mid g.v \mid \overline{g}) \mid \overline{u} \ .$$

If the only output on g synchronises with the left subprocess, a loop is produced by the two replications. If the right subprocess is selected, the divergence is avoided. A static type system would necessarily reject Q, due to the potential loop in the two replications ($!u.\overline{v}$ and $!v.\overline{u}$). In our hybrid system, by giving the same level to a, b, u and v, Q passes the static analysis and is annotated into $!p(a,b).(!a.[a>b]\overline{b} \mid (\nu g)\ (g.!b.[b>a]\overline{a} \mid g.b \mid \overline{g})) \mid \overline{p}\langle u,v\rangle \mid \overline{u}$. At run time, a computation matching the output $\overline{g}$ with the first input will yield $\bot$ but the computation matching this output with the second input on g will not.

Now consider $Q_1 = (\nu g)\ (\overline{g} \mid g.!a.\overline{b} \mid g.!b.\overline{a} \mid \overline{a} \mid \overline{b})$. This process is rejected by the type systems defined in the previous sections. Indeed, it is impossible for these type systems to notice that only one of the two replications can be made active (as only one output on g is available). Yet, this hybrid method can typecheck this process by giving the same weight to a and b and insert annotations $[a>b]$ and $[b>a]$ in the replications. As only one replication will be made available, no failure will be raised along execution. We now discuss the typing of recursive structures: trees with operations of *remote allocation*, that allow one to merge two trees by attaching the root of a tree to a leaf of another tree. To type the tree T_0 of Section 3.4.1, we need to take into account sequences of inputs in replications, that is, replications of the form $!a_1(x_1)\ldots a_n(x_n).P$ as we do in the type system of Section 3.2.1. (Precisely, in the subterm $!node(a,\ldots).a(\ldots)\ldots$, we need to compare the sum of the levels of names *node* and a against the weight of the continuation.) This can be easily done, as discussed in Remark 3.5.1, by strengthening the typing rule for replication in the static phase of the mixed system. Alternatively, we can keep the present typing rules and make some modifications to the programs. We discuss this solution.

Figure 3.9 presents the modified tree structure. The topmost replication, $!build(a,s_0,e_0)$, acts as a constructor, invoked for the creation of a new node; this new node carries values e_0, s_0, and interacts with the parent node via channel a. The state of this node is represented by the floating message on *state* (in which the first two components are the names for accessing the children, and are set to the special value $\mathbf{lab}(nil)$ if the node is a leaf). We make use of the choice operator + here, that has to be seen as a non-deterministic choice (choosing to reduce one of the branches of the + completely removes the code in the

$$
\begin{array}{l}
!\mathit{build}(a, s_0, e_0).\ (\nu \mathit{state})\ (\\
\quad \overline{\mathit{state}}\langle \mathit{nil}, \mathit{nil}, s_0, e_0 \rangle \\
\quad \mid\ !a(\mathit{chan}, \mathit{mode}).\mathit{state}(l, r, s, e).\mathit{chan}(v, \mathit{ans}, n). \\
\qquad \text{if } \mathit{mode} = \mathbf{lab}(\mathit{merge}) \text{ then} \\
\qquad\quad \text{if } l = \mathbf{lab}(\mathit{nil}) \text{ then } \overline{\mathit{state}}\langle n, r, s, e \rangle \\
\qquad\quad \text{else if } r = \mathbf{lab}(\mathit{nil}) \text{ then } \overline{\mathit{state}}\langle l, n, s, e \rangle \\
\qquad\quad \text{else } (\boldsymbol{\nu} \mathit{chan}')\ \ (\overline{l}\langle \mathit{chan}', \mathbf{lab}(\mathit{merge}) \rangle.\overline{\mathit{chan}'}\langle v, \mathit{ans}, n \rangle.\overline{\mathit{state}}\langle l, r, s, e \rangle \\
\qquad\qquad\qquad\qquad + \overline{r}\langle \mathit{chan}', \mathbf{lab}(\mathit{merge}) \rangle.\overline{\mathit{chan}'}\langle v, \mathit{ans}, n \rangle.\overline{\mathit{state}}\langle l, r, s, e \rangle\ ) \\
\qquad \text{else} \qquad \ldots)
\end{array}
$$

Figure 3.9: Merging tree structures

other branch). We only show the code for the $\mathbf{lab}(\mathit{merge})$ operation: the code for a search can be adapted from the example in Section 3.4.1. When $\mathbf{lab}(\mathit{merge})$ is invoked, the transmitted channel should be attached to a leaf; if there is room, this happens in the current node; otherwise the $\mathbf{lab}(\mathit{merge})$ operation is non-deterministically delegated to one of the children. (This is a simplified version of merge: the new tree is attached anywhere in the tree, without, for instance, ensuring that the tree remains well-balanced.)

The code above is accepted by the static analysis of the mixed type system, modulo the insertion of just a few annotations: the highest level is affected to names a, l, r, and an annotation $[a > l]$ (resp. $[a > r]$) is inserted before the output at l (resp. at r). The resulting annotated process does not lead to failure exceptions at run time. Notice that recursive types are needed for typing, independently from the termination analysis, but are straightforward to accommodate.

The mixed system remains, of course, incomplete — there are terminating processes whose annotated version yields $\bot$ — as the problem of the termination of a process is not decidable.

Remark 3.5.6 (Aborting or Doing nothing) *One can imagine a system without the semantic rules for $\bot$, and allowing only a reduction of an annotation when the order is compatible with it. Soundness for such a setting is proved exactly the same way. The difference is that the execution is not stopped explicitly when we reach a dangerous loop, but only implicitly.*

3.5.4 Efficiency

The static analysis of the mixed system can be made in time that is polynomial w.r.t. the size of the process being checked, by adapting the type inference algorithms of Section 5 (the modifications are mild). With such an algorithm, the static analysis introduces only the necessary assertions. More precisely, if the termination of a process can be proved by only relying on levels and weights (without referring to a partial order), then the static analysis will introduce no assertions and there will be no dynamic checks at run time.

Note that a trivial (and linear) static analysis would assign the same level to all names, and add assertions in front of all outputs prefixes. This would however mean that: all type checks are performed at run time; useful weight information is lost, so that the final termination analysis is rather rough.

Concerning the efficiency of dynamic checks, each time a new constraint is added to the $\mathcal{R}$ component of a configuration, we have to check for acyclicity of the resulting relation. This can be done via a depth-first traversal of $\mathcal{R}$, whose cost is linear in $\#\mathcal{R} + |\mathcal{R}|$, where $\#\mathcal{R}$ (resp. $|\mathcal{R}|$) stands for the size of $\mathtt{dom}(\mathcal{R})$ (resp. the number of pairs in $\mathcal{R}$). In [MSNR96], an online algorithm is presented, that allows one to perform the same task in linear amortised time in $\#\mathcal{R}$ only.

Chapter 4

Type Systems for termination in process-passing π-calculi

The following sections present proofs of termination for the higher-order concurrent languages we defined in Section 2.

4.1 In HOpi$_2$

In this section, we present an adaptation of the system of Section 3.1 for π-calculus to the calculus HOpi$_2$, previously presented in Section 2. The task is not trivial, as this type system ensures termination by relying on a control of replicated inputs, which are the sole source of divergence in π. As stated in Section 2, the replication operator is not present in the syntax of HOpi$_2$, but diverging behaviours still appear. Remember the process $Q_0 = P_0 \mid \overline{a}\langle P_0\rangle$, where $P_0 = a(X).(X \mid \overline{a}\langle X\rangle)$. This process reduces to itself in one step. When trying to identify what is unsafe in the code Q_0, we notice that the process P_0, emitted on the channel a, contains itself an emission on a. This can be seen as a "recursive" output, to be compared with, in the π-calculus setting, the presence of an output on a inside the continuation P of a replication $!a(x).P$.

Moreover, if we forbid such recursive outputs, by assigning a *level* to each channel and ensuring that no emission of processes containing outputs at level $\geq l$ are performed on channels of level l, termination is guaranteed. Indeed, at each reduction step (for instance when $\overline{a}\langle P\rangle \mid a(X).Q \to Q\{P/X\}$), the typing rule for output prefixes ensures that an output of level l is consumed (here $\overline{a}\langle P\rangle$) and the new outputs appearing in the processes (resulting from the instantiation of the occurrences of process variable X by P) have levels strictly smaller than l. This is enough to construct a well-founded measure that decreases at each step.

To sum up, the control present in the type system for π is moved from replicated inputs to outputs. In the further type systems we develop for higher-order calculus, the crux of the termination proof is always the typing rule for output.

Type System for termination in HOpi$_2$

Types for HOpi$_2$ channels are simpler than the ones for π channels, as the former only carry processes as messages. Thus the syntax for types is:

$$T ::= \sharp^k \diamond$$

As above, typing contexts Γ assign types to names (written $\Gamma(a) = \sharp^k \diamond$). However, they also assign levels to process variables (written $\Gamma(X) = n$). Typing rules for HOpi$_2$ are given by Figure 4.1.

One can check that the process Q_0 is indeed ruled out by our type system, as the presence of an output on a (say that the level of a is l) inside P_0 implies that P_0 has a weight of at least l. As P_0 appears in message position in an output on a, rule (**HOut**) will require $l > l$.

$$(\mathbf{HNil})\frac{}{\Gamma \vdash^H \mathbf{0} : 0} \qquad (\mathbf{HVar})\frac{\Gamma(X) = n}{\Gamma \vdash^H X : n} \qquad (\mathbf{HRes})\frac{\Gamma(a) = \sharp^k \diamond \qquad \Gamma \vdash^H P : n}{\Gamma \vdash^H (\boldsymbol{\nu} a)P : n}$$

$$(\mathbf{HPar})\frac{\Gamma \vdash^H P_1 : n_1 \qquad \Gamma \vdash^H P_2 : n_2}{\Gamma \vdash^H P_1 \mid P_2 : \max(n_1, n_2)} \qquad (\mathbf{HIn})\frac{\Gamma(X) = (k-1) \qquad \Gamma(a) = \sharp^k \diamond \qquad \Gamma \vdash^H P : n}{\Gamma \vdash^H a(X).P : n}$$

$$(\mathbf{HOut})\frac{\Gamma \vdash^H Q : m \qquad \Gamma \vdash^H P : n \qquad \Gamma(a) = \sharp^k \diamond \qquad m < k}{\Gamma \vdash^H \overline{a}\langle Q \rangle.P : \max(k, n)}$$

Figure 4.1: Typing rules for termination in HOpi_2

We notice that the rules (**HNil**), (**HPar**), (**HRes**) of Figure 4.1 are similar to their π counterparts.

The rules (**HVar**) and (**HIn**) ensures that process variables are taken into account when defining the global weight of a process. Giving weight n to the process X (even if it contains no outputs of any kind) is necessary, as it can be instantiated along reductions. A system not taking process variable into account, for instance one using a rule (**HVar′**) giving weight 0 to the process X would be unsound. Consider:

$$Q'_0 = b(Y).\overline{a}\langle Y \rangle \mid a(Z).(Z \mid \overline{a}\langle Z \rangle) \mid \overline{b}\langle a(X).(X \mid \overline{a}\langle X \rangle)\rangle$$

this process reduces in one communication step on channel b to Q_0. However, the process abides to the discipline of this particular system: by giving level 2 to b and level 1 to a, we notice that the condition concerning outputs stated above holds (because the process sent on b contains only an output on a, the level of a being strictly smaller than the level of b, and because the process sent on a contains no output).

Actually, one has to be aware that the process variable Y could be instantiated by anything carried on channel b, that is, processes potentially containing outputs on a. As a consequence, on one side, our rule (**HIn**) ensures that a process variable X bound by an input on a of level k is given a level equal to the maximum weight of a process that can instantiate X, namely $k-1$. On the other side, the rule (**HVar**) lets the level of a process variable contribute to the weight of a process. Thus, the above diverging process Q'_0 is ruled out by our system: if b is given level k and a level l, the output on b of a process containing an output on a gives the constraint $k > l$; being received on b, Y has level $k-1$, thus the output $\overline{a}\langle Y \rangle$ gives the constraint $l > k-1$. As we cannot satisfy $k > l > k-1$, the process cannot be type-checked.

As usual, we use a multiset measure that strictly decreases at each reduction step to prove the soundness of our type system. We use here the multiset of levels of every *available* outputs in a process, where available means, in this case, that the outputs do not occur inside a message.

For instance, consider

$$E_1 = a(X).a(X').(X \mid X \mid X') \mid \overline{a}\langle(\overline{b}\langle\mathbf{0}\rangle \mid c(Y).Y)\rangle \mid \overline{a}\langle(\overline{c}\langle\overline{b}\langle\mathbf{0}\rangle\rangle \mid b(Z).\mathbf{0})\rangle$$

E_1 is typable by giving level 3 to a, level 2 to c and level 1 to b. Indeed, inside the message sent on the first output on a, there is only one available output on b, and inside the message sent on the second one there is only one available output on c. Moreover, the latter output contains inside its message an output on b. We present here a possible reduction sequence starting from E_1:

$$E_1 \to a(X').(\overline{c}\langle\overline{b}\langle\mathbf{0}\rangle\rangle \mid b(Z).\mathbf{0} \mid \overline{c}\langle\overline{b}\langle\mathbf{0}\rangle\rangle \mid b(Z').\mathbf{0} \mid X') \mid \overline{a}\langle(\overline{b}\langle\mathbf{0}\rangle \mid c(Y).Y)\rangle$$

$$\to \overline{c}\langle\overline{b}\langle\mathbf{0}\rangle\rangle \mid b(Z).\mathbf{0} \mid \overline{c}\langle\overline{b}\langle\mathbf{0}\rangle\rangle \mid b(Z').\mathbf{0} \mid \overline{b}\langle\mathbf{0}\rangle \mid c(Y).Y$$

$$\to b(Z).\mathbf{0} \mid \overline{c}\langle\overline{b}\langle\mathbf{0}\rangle\rangle \mid b(Z').\mathbf{0} \mid \overline{b}\langle\mathbf{0}\rangle \mid \overline{b}\langle\mathbf{0}\rangle$$

$$\to\to \overline{c}\langle\overline{b}\langle\mathbf{0}\rangle\rangle$$

with the corresponding sequence of multisets of available outputs being:

$$\{3,3\} \to \{3,2,2\} \to \{2,2,1\} \to \{2,1,1\} \to \{2,1\} \to \{2\}$$

Termination Proof

In this section, we explicit the soundness proof of our type system. That is, we prove that every typable process is terminating. Even if the crux of the system is no longer the typing rule for replicated inputs but the one for outputs, the global structure remains the same: we first prove a Subject Substitution property, then we define a multiset measure, prove that this measure decreases at each reduction step and conclude by using the well-foundedness of the multiset extension of well-founded order.

First, we state the usual fact relating typability and evaluation contexts, as this result is required in order to prove Subject Substitution the way we state it.

Fact 4.1.1 (Typing and contexts)
If $\Gamma \vdash^H \mathbf{E}[P] : n$ *then:*

1. $\Gamma \vdash^H P : n'$ *for* $n' \leq n$
2. *for any,* P_1 *s.t.* $\Gamma \vdash P_1 : n_1$*, we have* $\Gamma \vdash^H \mathbf{E}[P_1] : n_{(1)}$ *for some* $n_{(1)}$*.*

Proof. *By structural induction on* $\mathbf{E}$*.* □

We also state that Subject Congruence property holds in this higher-order context, as we need it for the proof of Subject Reduction.

Lemma 4.1.2 (Subject Congruence)
If $P \equiv Q$*, then* $\Gamma \vdash^H P : n$ *iff* $\Gamma \vdash^H Q : n$*.*

Proof. *This result is established by induction on the derivation of* $P \equiv Q$ *(taking into account the symmetry property), using the fact that the* max *operator satisfies laws of associativity and commutativity.* □

We introduce now a measure on processes, defined with respect to a typing context (as to compute the measure, we need the levels of names used in prefixes inside the process). We introduce $\mathbf{M}^H(P)$, the measure associated to P, which is given as a multiset of natural numbers. As stated above, $\mathbf{M}^H(P)$ is the multiset of levels of names appearing in available output position inside P. Notice that the measure is not computed for subprocesses found inside messages, as suggested by the definition of available outputs.

Definition 4.1.3 (Measure)
When $\Gamma \vdash^H P : n$*, we inductively define* $\mathbf{M}^H(P)$*:*

$$\mathbf{M}^H(\mathbf{0}) = \mathbf{M}^H(X) = \emptyset \qquad \mathbf{M}^H(P_1 \mid P_2) = \mathbf{M}^H(P_1) \uplus \mathbf{M}^H(P_2) \qquad \mathbf{M}^H((\nu a)\ P_1) = \mathbf{M}^H(P_1)$$

$$\mathbf{M}^H(a(X).P_1) = \mathbf{M}^H(P_1) \qquad \mathbf{M}^H(\overline{a}\langle Q\rangle.P_1) = \mathbf{M}^H(P_1) \uplus \{k\} \text{ if } \Gamma(a) = \sharp^k \diamond$$

The measure is straightforwardly extended to contexts with $\mathbf{M}^H(\ ) = \emptyset$*.*

Fact 4.1.4 (Measure and Congruence)
If $P \equiv Q$ *then* $\mathbf{M}^H(P) = \mathbf{M}^H(Q)$*.*

Proof. *By induction on the derivation of* $P \equiv Q$*, using the definition of* $\mathbf{M}^H()$*, the commutativity, the associativity and the neutrality of* $\emptyset$ *for* $\uplus$*.* □

The Subject Substitution lemma shows that typability is preserved when we substitute process variables with a typable process, provided some conditions relating the levels of the process variables and the weights of the processes instantiating it are met. As written above, the level of a process variable corresponds to the maximum weight of a process instantiating it, the statement of the lemma ensures that it is indeed the case.

This lemma also explains how the measure evolves when performing a a substitution abiding such conditions. The integer c appearing in case 2., is the number of available occurrences (the notion of *availability* is easily extended to any subprocess) of the process variable X inside P. Thus the equation $\mathbf{M}^H(P\{Q/X\}) = \mathbf{M}^H(P) \uplus c.\mathbf{M}^H(Q)$ means that measure of $P\{Q/X\}$ is equal to the measure of P (remember that process variables count for $\emptyset$ inside the definition of $\mathbf{M}^H(\cdot)$) to which we add a copy of the measure of Q for each available occurrences of X inside P.

Lemma 4.1.5 (Subject Substitution)
If $\Gamma \vdash^H P : n$, $\Gamma \vdash^H Q : m'$ and $\Gamma(X) = m$ with $m' \leq m$, then there exist $n' \leq n$ and c s.t.

1. *$\Gamma \vdash^H P\{Q/X\} : n'$ and*
2. *$\mathbf{M}^H(P\{Q/X\}) = \mathbf{M}^H(P) \uplus c.\mathbf{M}^H(Q)$.*

Proof. *By induction on the typing judgement $\Gamma \vdash^H P : n$.*

- *Case* (**HNil**) *is immediate.*
- *Case* (**HPar**). *We have $P = P_1 \mid P_2$. We use the induction hypothesis, the rule* (**HPar**)*, the fact that $(P_1 \mid P_2)\{Q/X\} = (P_1[\{Q/X\}) \mid (P_2\{Q/X\})$ and Definition 4.1.3 to conclude.*
- *Case* (**HRes**). *We have $P = (\boldsymbol{\nu} a)\ P_1$. We use the induction hypothesis, rule* (**HRes**) *and Definition 4.1.3 to conclude.*
- *Case* (**HVar**). *The case where $P = Y$ and $Y \neq X$ is immediate. Suppose $P = X$. As $X\{Q/X\} = Q$, $m' \leq m$ and $\mathbf{M}^H(X) = \emptyset$, we set $c = 1$.*
- *Case* (**HIn**). *We have $P = a(Y).P_1$. We remark that if X occurs free in P, then $X \neq Y$. Thus $(a(Y).P_1)\{Q/X\} = a(Y).(P_1\{Q/X\})$, and we can rely on the induction hypothesis on P_1 to conclude using rule* (**In**) *and Definition 4.1.3.*
- *Case* (**HOut**). *We have $P = \overline{a}\langle S\rangle.P_1$. We have $\Gamma(a) = \sharp^l T$ and $\Gamma \vdash^H S : n_S$, $\Gamma \vdash^H P_1 : n_1$ with $l > n_S$ and $n = \max(l, n_1)$. By induction, we have $\Gamma \vdash^H S\{Q/X\} : n'_S \leq n_S$, $\Gamma \vdash^H P_1\{Q/X\} : n'_1 \leq n_1$, $\mathbf{M}^H(P_1\{Q/X\}) = \mathbf{M}^H(P_1) \uplus c_1.\mathbf{M}^H(Q)$. As $l > n_S \geq n'_S$, we can derive $\Gamma \vdash^H \overline{a}\langle S\{Q/X\}\rangle.P_1\{Q/X\} : \max(l, n'_1)$ and we have $\max(l, n'_1) \leq \max(l, n_1)$. Definition 4.1.3 gives $\mathbf{M}^H(P) = \{l\} \uplus \mathbf{M}^H(P_1)$ and $\mathbf{M}^H(P\{Q/X\}) = \{l\} \uplus \mathbf{M}^H(P_1\{Q/X\}) = \{l\} \uplus \mathbf{M}^H(P_1) \uplus c_1.\mathbf{M}^H(Q)$. This allows us to conclude by setting $c = c_1$.*

□

The following auxiliary lemma relates the measure of a process with its weight. Notice that the inequation $\mathbf{M}^H(P) <_{mul} \{n+1\}$ can be rephrased as "the maximum element of the measure of a process of weight n is an integer smaller than n". This allows us, in the Subject Reduction proof, to bound the measure of an instantiated process.

Lemma 4.1.6 (Measure domination)
If $\Gamma \vdash^H P : n$, then $\mathbf{M}^H(P) <_{mul} \{n+1\}$.

Proof. *By induction on the typing judgement.*

- *Case* (**HNil**). *Immediate, as $\{1\} >_{mul} \emptyset$.*
- *Case* (**HRes**). *We have $P = (\boldsymbol{\nu} a)\ P_1$. We derive $\Gamma \vdash^H P_1 : n$. We have $\mathbf{M}^H(P) = \mathbf{M}^H(P_1)$. The inductive hypothesis gives $\mathbf{M}^H(P_1) < \{n+1\}$. Thus we have $\mathbf{M}^H(P) < \{n+1\}$.*
- *Case* (**HVar**). *We have $P = X$. By definition of the measure, $\mathbf{M}^H(X) = \emptyset$, hence the result.*

- *Case* (**HPar**). *We have* $P = P_1 \mid P_2$. *We derive* $\Gamma \vdash^H P_1 : n_1, \Gamma \vdash^H P_2 : n_2$. *We have* $n = \max(n_1, n_2)$. *By the inductive hypotheses,* $\{n_1+1\} >_{mul} \mathbf{M}^H(P_1)$ *and* $\{n_2+1\} >_{mul} \mathbf{M}^H(P_2)$. *As* $\max(n_1, n_2)+1 \geq n_1 + 1$ *and* $\max(n_1, n_2) + 1 \geq n_2 + 1$, *we deduce* $\{\max(n_1, n_2) + 1\} >_{\text{mul}} \mathbf{M}^H(P_1) \uplus \mathbf{M}^H(P_2)$.

- *Case* (**HIn**). *We have* $P = a(X).P_1$. *We derive* $\Gamma, X : k - 1 \vdash^H P_1 : n$. *The induction hypothesis gives* $\{n + 1\} > \mathbf{M}^H(P_1)$, *and, by definition,* $\mathbf{M}^H(a(X).P_1) = \mathbf{M}^H(P_1)$. *We thus conclude that* $\{n + 1\} > \mathbf{M}^H(a(X).P_1)$.

- *Case* (**HOut**). *We have* $P = \overline{a}\langle Q_2\rangle.P_1$. *There exists* k *s.t.* $\Gamma(a) = \sharp^k \diamond$ *and we derive* $\Gamma \vdash^H Q_2 : n_2$, $\Gamma \vdash^H P_1 : n_1$. *We have* $\mathbf{M}^H(\overline{a}\langle P_1\rangle.Q) = \mathbf{M}^H(P_1) \uplus \{k\}$. *By induction, we have* $\{n_1+1\} >_{\text{mul}} \mathbf{M}^H(P_1)$. *We conclude that* $\{\max(k, n_1) + 1\} >_{mul} \mathbf{M}^H(\overline{a}\langle P_1\rangle.Q))$.

□

Again, we need this small fact, stating that the measure of a process $\mathbf{E}[P]$ is the measure of the context $\mathbf{E}$ added to the measure of the process P inside the hole.

Fact 4.1.7 (Measure and contexts)
If $\Gamma \vdash^H \mathbf{E}[P] : n$, *then* $\mathbf{M}^H(\mathbf{E}[P]) = \mathbf{M}^H(\mathbf{E}) \uplus \mathbf{M}^H(P)$

Proof. *Easily done by structural induction over* $\mathbf{E}$. □

The following proposition states the key property of our type system: when a typable process P reduces to P', not only is P' typable, but the measure decreases between the two processes. That is, our type system ensures that the level of the output consumed by a reduction, considered as a multiset-singleton, is greater, for the multiset extension of the standard ordering over natural numbers, than the total measure of the processes being spawned (the ones instantiating the process variables bound by the consumed input).

Proposition 4.1.8 (Subject reduction)
If $\Gamma \vdash^H P : n$ *and* $P \to P'$ *then* $\Gamma \vdash^H P : n'$ *for some* n', *and* $\mathbf{M}^H(P') <_{mul} \mathbf{M}^H(P)$.

Proof.
By induction on the derivation of $P \to P'$.

- *Case* (**HCom**). *We have* $P = \mathbf{E}[\overline{a}\langle Q\rangle.P_1 \mid a(X).P_2] \to P' = \mathbf{E}[P_1 \mid P_2\{Q/X\}]$. *From Fact 4.1.1, we get* $\Gamma \vdash^H \overline{a}\langle Q\rangle.P_1 \mid a(X).P_2 : m$. *Then, we derive* $\Gamma \vdash^H \overline{a}\langle Q\rangle.P_1 : m_1$, $\Gamma \vdash^H a(X).P_2 : m_2$, $\Gamma \vdash^H P_1 : m'_1$, $\Gamma \vdash^H Q : m_0$, $\Gamma \vdash^H P_2 : m_2$, $\Gamma(X) = l - 1$ *and* $\Gamma(a) = \sharp^l \diamond$ *with* $l > m_0$ *for some* m_0, m_1, m'_1, m_2. *Fact 4.1.7 gives* $\mathbf{M}^H(P) = \mathbf{M}^H(\mathbf{E}) \uplus \mathbf{M}^H(P_1) \uplus \mathbf{M}^H(P_2) \uplus \{l\}$. *By applying Lemma 4.1.5, we get* $\Gamma \vdash^H P_2\{Q/X\} : m'_2$ *with* $m'_2 \leq m_2$ *and* $\mathbf{M}^H(P_2\{Q/X\}) = \mathbf{M}^H(P_2) \uplus c.\mathbf{M}^H(Q)$ *for some* c. *This allows us to use Fact 4.1.1 and rule* (**Par**) *to derive* $\Gamma \vdash^H P' : n'$ *with* $n' = \max(n'_1, n'_2)$. *From Definition 4.1.3 and Fact 4.1.7, we deduce that* $\mathbf{M}^H(P') = \mathbf{M}^H(\mathbf{E}) \uplus \mathbf{M}^H(P_1) \uplus \mathbf{M}^H(P_2\{Q/X\}) = \mathbf{M}^H(\mathbf{E}) \uplus \mathbf{M}^H(P_1) \uplus \mathbf{M}^H(P_2) \uplus c.\mathbf{M}^H(Q)$. *From Lemma 4.1.6, we get* $\mathbf{M}^H(Q) <_{mul} \{m+1\}$. *This implies that* $c.\mathbf{M}^H(Q) <_{mul} \{m + 1\}$, *and we finally obtain* $c.\mathbf{M}^H(Q) <_{mul} \{l\}$. *Thus* $\mathbf{M}^H(P) >_{mul} \mathbf{M}^H(P')$.

- *Case* (**Cong**) *we use the induction hypothesis, Lemma 4.1.2 and the Fact 4.1.4 that* $\mathbf{M}^H()$ *is invariant by* $\equiv$.

□

Such a result allows us to prove the soundness of our type system, as every infinite reduction starting from a typable process contains only typable processes and the associated well-founded measure strictly decreases an infinite number of times, which is impossible.

Proposition 4.1.9 (Soundness)
If $\Gamma \vdash^H P : n$, *then* P *terminates.*

Proof. *Consider, towards a contradiction, an infinite sequence of reductions $(P_i)_{i\geq 0}$ emanating from $P = P_0$ (that is, $P_i \to P_{i+1}$ for $i \geq 0$).*

Proposition 4.1.8 allows us to construct an infinite sequence of typing judgements $(\Gamma \vdash^H P_i : n_i)_i$ and a strictly decreasing infinite sequence $(\mathbf{M}^H(P_i))_i$, which is contradictory with Theorem 2.1.3. □

Clearly, our type system fails to capture all terminating processes: there are processes that are not typable and that do not exhibit infinite computations. An example is given by $\overline{a}\langle\overline{a}\langle\mathbf{0}\rangle\rangle \mid a(X).X$, in which the recursive output on a prevents us from type-checking the process. We will see later that we can improve the expressiveness of our type system and capture more terminating process.

Typing the encoding of HOpi$_2$ into π

We now compare the expressiveness of our type system with the expressiveness induced on HOpi$_2$ by the translation into π and the existing type system of Section 3.1 for the π-calculus. We rely on (an adaptation of) the standard encoding of HOpi$_2$ into the π-calculus [San92] (see also [Tho96]).

In order to make things clearer, we encode HOpi$_2$ into a fragment of the π-calculus. The target calculus uses two kinds of channels: CCS-like channels (which are used only for synchronisation), ranged over h, and first-order channels, which are used to transmit CCS-like channels, ranged over using a, b, c. We write $[\![P]\!]$ for the π-calculus encoding of a HOpi$_2$ process P. The definition of $[\![P]\!]$ is rather standard. We recall it here (an unambiguous correspondence between HOpi$_2$ process variables – X – and their counterpart as CCS-like channels – h_X – is implicitly assumed):

Definition 4.1.10 (Encoding of HOpi$_2$ into π)
The encoding of HOpi$_2$ processes into π is inductively defined by:

$$[\![\mathbf{0}]\!] = \mathbf{0} \qquad [\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!] \qquad [\![(\boldsymbol{\nu} c)\ P]\!] = (\boldsymbol{\nu} c)\ [\![P]\!] \qquad [\![a(X).P]\!] = a(h_X).[\![P]\!] \qquad [\![X]\!] = \overline{h_X}$$

$$[\![\overline{a}\langle P\rangle.Q]\!] = (\boldsymbol{\nu} h_a)\ \overline{a}\langle h_a\rangle.(\ [\![Q]\!] \mid !h_a.[\![P]\!]\ ) \quad h_a \text{ fresh}$$

A higher-order output action $\overline{a}\langle P\rangle.Q$ is translated into the emission of a new name (h_a), which intuitively represents the address where process P can be accessed. Thus, instead of communicating a process P, an address containing a server providing copies of P is communicated. As a result, process variables can be seen as address variables. This encoding respects termination, as expressed by the following result.

Proposition 4.1.11 (Simulation through the encoding)
For any HOpi$_2$ process P, P terminates iff $[\![P]\!]$ terminates.

Proof. *Follows from Theorem 13.1.18 in [SW01].* □

In particular, the non-terminating process Q_0 of Section 2.3.1 is translated into

$$[\![Q_0]\!] = (\boldsymbol{\nu} h_a)\ \overline{a}\langle h_a\rangle.!h_a.P' \mid P' \qquad \text{where } P' = a(h_X).(\boldsymbol{\nu} h'_a)\ \overline{a}\langle h'_a\rangle.(!h'_a.\overline{h_X} \mid \overline{h_X})$$

We have two approaches to ensure termination of HOpi$_2$ processes: on the one hand, the type system we presented in this section; on the other hand, the method consisting in type-checking the translation of a HOpi$_2$ process into π.

It is no surprise, that process $[\![Q_0]\!]$ (see above) is rejected by the system of Section 3.1: first observe that the levels of h_a and h_X are necessarily equal, since they are both transmitted on channel a. This entails that subprocess $!h_a.P'$ is not typable, because of the output on h_X in P'.

There do moreover exist HOpi$_2$ processes that can be proved to terminate using the type system for HOpi$_2$, but whose encoding fails to be typable using the type system for π. A very simple example is given by $R_0 = a(X).\overline{a}\langle X\rangle$. We indeed have

$$[\![R_0]\!] = a(h_X).(\boldsymbol{\nu} h_a)\ \overline{a}\langle h_a\rangle.!h_a.\overline{h_X},$$

a process which is not typable: indeed, h_X and h_a necessarily have the same type (both are transmitted on a), which prevents subprocess $!h_a.\overline{h_X}$ from being typable.

This example suggests a way to establish a relationship between the type systems in HOpi_2 and in π. Consider for that the type system for HOpi_2 obtained by replacing rule (**HIn**) in Figure 4.1 with the following one, the other rules remaining unchanged (the typing judgement for this modified type system shall be written $\Gamma \vdash^{H'} P : n$ in the following:

$$(\textbf{HIn}')\frac{\Gamma \vdash^{H'} P : n \qquad \Gamma(a) = \sharp^k \diamond \qquad \Gamma(X) = k}{\Gamma \vdash^{H'} a(X).P : n}$$

Clearly, the modified type system is more restrictive, that is, $\Gamma \vdash^{H'} P : n$ implies $\Gamma \vdash^{H} P : n$, but not the converse (cf. process R_0 seen above).

Using this system, we can establish the following property, that allows us to draw a comparison between typability in HOpi_2 and in the π-calculus:

Proposition 4.1.12 (Typability of the encoding)
Let P be a HOpi$_2$ process. If $\Gamma \vdash^{H'} P : n$, then there exists Δ, a typing context for the π-calculus, such that $\Delta \vdash [\![P]\!] : n'$ for $n' \leq n$.

Proof.
The encoding presented above induces a translation of HOpi$_2$ typing contexts, defined as follows (we write $[\![\Gamma]\!]$ for the encoding of Γ):

- *If $\Gamma(X) = n$, then $[\![\Gamma]\!](h_X) = \sharp^n \mathbb{1}$*
- *If $\Gamma(a) = \sharp^n \diamond$, then $[\![\Gamma]\!](a) = \sharp^0 \sharp^n \mathbb{1}$.*

We reason by induction on the derivation of $\Gamma \vdash^{H'} P : n$ to prove that $\Gamma \vdash^{H'} P : n$ implies $[\![\Gamma]\!] \vdash [\![P]\!] : n$:

- *The cases corresponding to rules* (**HRes**) *and* (**Par**) *are treated easily by relying on the induction hypothesis. Case* (**Nil**) *is trivial.*
- *Case* (**HVar**). *We can apply rule* (**Out**) *to derive $[\![\Gamma]\!] \vdash [\![X]\!] : n$ since $[\![X]\!] = \overline{h_X}$.*
- *Case* (**HIn**$'$). *We have $[\![a(X).P]\!] = a(h_X).[\![P]\!]$. We know by induction that $[\![\Gamma]\!] \vdash [\![P]\!] : n'$, with $[\![\Gamma]\!](h_X) = \sharp^k \mathbb{1}$ and $n' \leq n$. We moreover know $\Gamma(a) = \sharp^k \diamond$, which gives $[\![\Gamma]\!](a) = \sharp^0 \sharp^k \diamond$. This allows us to use rule* (**In**) *to derive $[\![\Gamma]\!] \vdash [\![a(X).P]\!] : n'$.*
- *Case* (**HOut**). *Recall that $[\![\overline{a}\langle P\rangle.Q]\!] = (\boldsymbol{\nu} h_a)\ \overline{a}\langle h_a\rangle.(\,[\![Q]\!] \mid\, !h_a.[\![P]\!]\,)$, for some fresh h_a. We know by induction that $[\![\Gamma]\!] \vdash [\![P]\!] : k'$ and $[\![\Gamma]\!] \vdash [\![Q]\!] : m'$ with $k' \leq k$ and $m' \leq m$. By hypothesis, we also have $\Gamma(a) = \sharp^n \diamond$, which gives $[\![\Gamma]\!](a) = \sharp^0 \sharp^n \mathbb{1}$ and $[\![\Gamma]\!](h) = \sharp^n \mathbb{1}$. We can thus derive $[\![\Gamma]\!] \vdash !h_a.[\![P]\!] : 0$, using rule* (**Rep**), *as $k' \leq k < n$ holds. This gives, using rule* (**Par**), *$[\![\Gamma]\!] \vdash [\![Q]\!] \mid !h_a.[\![P]\!] : m'$. We can now apply rule* (**Out**) *to derive the judgement $[\![\Gamma]\!] \vdash \overline{a}\langle h_a\rangle.(\,[\![Q]\!] \mid !h_a.[\![P]\!]\,) : m'$. Finally, we can use* (**Res**) *to obtain the expected result.*

□

In case (**HIn**$'$) of the proof above, we remark that the typing hypothesis $\Gamma(X) = k$ in the original HOpi_2 derivation allows us to construct the π-calculus typing. If we were using rule (**HIn**) from Figure 4.1, we could not conclude.

The limits of our type system

Proposition 4.1.12 shows that typability of a HOpi_2 process (in the sense of the modified type system) entails typability of its encoding. By Theorem 3.1.13, going via the encoding in π therefore provides a procedure to ensure termination of HOpi_2 processes. We can observe that there do exist terms that can be typed via the encoding, but that are rejected by our type systems for HOpi_2 (by both the modified type system and the type system from the beginning of this section). This observation, together with the discussion about process R_0 above, shows that the two approaches to ensure termination of HOpi_2 processes are incomparable. Consider indeed processes:

$$R_1 = \overline{a}\langle\overline{a}\langle\mathbf{0}\rangle\rangle \mid a(X).\mathbf{0} \quad \text{and} \quad R_2 = a(X).b(Y).X \mid \overline{a}\langle\overline{a}\langle\mathbf{0}\rangle\rangle \mid \overline{b}\langle\mathbf{0}\rangle \ .$$

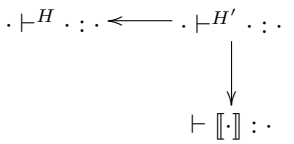
None of them is typable, because they contain "self-emissions" (an output action on channel a occurring inside a process emitted on a). However, R_1 and R_2 are terminating. Their encodings in π are

$$[\![R_1]\!] = (\nu h_a)\ \overline{a}\langle h_a\rangle.!h_a.(\nu h'_a)\ \overline{a}\langle h'_a\rangle.!h'_a.\mathbf{0} \mid a(h_X).\mathbf{0}$$

$$[\![R_2]\!] = a(h_X).b(h_Y).\overline{h_X} \mid (\nu h_a)\ \overline{a}\langle h_a\rangle.!h_a.(\nu h'_a)\ \overline{a}\langle h'_a\rangle.!h'_a.\mathbf{0} \mid (\nu h_b)\ \overline{b}\langle h_b\rangle.!h_b.\mathbf{0}$$

which are both typable using the system of Figure 3.1. A suitable assignment for R_1 is, e.g., $\Gamma(h_a) = \Gamma(h'_a) = \sharp^1\,\mathbb{1}$. Both replications are typed as they have no first-order outputs in their continuation. R_2 can be typed with the same level assignment, extended with $\Gamma(h_b) = \Gamma(h'_b) = \sharp^1\,\mathbb{1}$.

One can conclude by drawing the following diagram, where an arrow from X to Y means that method X is strictly more expressive than method Y, i.e., each HOpi_2 process recognised as terminating by Y is recognised as such by X, but there exists HOpi_2 process recognised as terminating by X which are rejected by Y.



4.2 In HOpi_ω

Types for termination in HOpi_ω

In this section, we use the type system we studied above as a base to build a type system ensuring termination of HOpi_ω processes. Things get more complicated here as β-reductions are able spawn new outputs inside processes. Indeed, where a functional value is applied to its argument, a new process is spawned (remember that in HOpi_ω, applying a function to its argument always yields a process). As functional values can be communicated, one has to control them, by giving them a level roughly corresponding to the maximum weight of a process obtained by the application of them to an argument.

As a consequence, the grammar for types for HOpi_ω contains types for values, given by:

$$T ::= \mathbb{1} \mid T \to^n \diamond$$

and types for channels of the form $Ch^n(T)$.

In manipulating types, we restrict ourselves to using only *well-formed* value types, defined as follows:

Definition 4.2.1 (Well-formed type) *We write* $\texttt{Lvl}(T) = k$ *if* $T = \mathbb{1}^0$ *and* $k = 0$ *or if* $T = T_1 \to^k \diamond$.

A value type is well formed if it is $\mathbb{1}^0$ *or* $T_1 \to^k \diamond$ *with* T_1 *well-formed and* $\texttt{Lvl}(T_1) < k$.

A channel type is well formed if it is of the form $\sharp^k T$ *and* T *is well-formed.*

Typing values

$$(\textbf{HoUnit})\frac{}{\Gamma \vdash_\omega \star : \mathbb{1}^0} \qquad (\textbf{HoAbs})\frac{\Gamma(x) = T \qquad \Gamma \vdash_\omega P : n}{\Gamma \vdash_\omega x \mapsto P : T \to^{n+1} \diamond} \qquad (\textbf{Hovar})\frac{\Gamma(x) = T}{\Gamma \vdash_\omega x : T}$$

Typing processes

$$(\textbf{HoNil})\frac{}{\Gamma \vdash_\omega \mathbf{0} : 0} \qquad (\textbf{HoRes})\frac{\Gamma(a) = Ch^k(T) \qquad \Gamma \vdash_\omega P : n}{\Gamma \vdash_\omega (\boldsymbol{\nu} a)P : n} \qquad (\textbf{HoPar})\frac{\Gamma \vdash_\omega P_1 : n_1 \qquad \Gamma \vdash_\omega P_2 : n_2}{\Gamma \vdash_\omega P_1 \mid P_2 : \max(n_1, n_2)}$$

$$(\textbf{HoApp})\frac{\Gamma \vdash_\omega v_1 : T \to^n \diamond \qquad \Gamma \vdash_\omega v_2 : T}{\Gamma \vdash_\omega v_1 \lfloor v_2 \rfloor : n} \qquad (\textbf{HoIn})\frac{\Gamma(x) = T \qquad \Gamma(a) = Ch^k(T) \qquad \Gamma \vdash_\omega P : n}{\Gamma \vdash_\omega a(x).P : n}$$

$$(\textbf{HoOut})\frac{\Gamma \vdash_\omega v : T \qquad \Gamma \vdash_\omega P : n' \qquad \Gamma(a) = Ch^n(T) \qquad \mathtt{Lvl}(T) = k \qquad n > k}{\Gamma \vdash_\omega \overline{a}\langle v \rangle.P : \max(n, n')}$$

Figure 4.2: Typing rules for termination in HOpi_ω

The definition of well-formedness states that the level assigned to an argument is always smaller that the type of the function. Thus, in order to count the weight of $v_1 \lfloor v_2 \rfloor$, one has to take into account only the level v_1, as the one of v_2 is smaller by well-formedness of types. Moreover, when reducing $(x \mapsto P)\lfloor v_2 \rfloor$ into $P\{v_2/x\}$, if x appears in function position (for instance in $x\lfloor v_3 \rfloor$), the weight of a value v_2 instantiating such an x will be taken into account.

Typing rules for HOpi_ω values and processes are given by Figure 4.2, where we implicitly impose that every value type appearing in these rules is a well-formed value type.

The rule (**HoAbs**) states that the level of a functional value is equal to 1 plus the weight of the process appearing in its definition. Rule (**HoApp**) defines the weight of an application of a function to an argument as the level of the function. As expected, the actual control takes place inside the rule (**HoOut**) where we force the level of the value being sent to be strictly smaller than the level of the channel on which it is sent.

As in Section 4.1, channel types are annotated with a level, and the weight assigned to a process is given by a natural number. The weight of a process P is bound to dominate both the maximum level of outputs contained in P (not occurring inside a message), as in Section 4.1 and, for any subprocess of the form $v_1 \lfloor v_2 \rfloor$ that occurs in P not inside a message, the maximum level associated to the function v_1.

As before, termination is proved by associating to a process a measure that decreases along reductions. We cannot focus our analysis, as above, only on the multiset of names used in output subject position in P (written $\mathbf{Os}(P)$), because β-reductions may let this multiset grow. For instance, if we take $P = (x \mapsto (\overline{a}\langle\star\rangle \mid \overline{a}\langle\star\rangle))\ \lfloor\star\rfloor$ in a context ensuring $\Gamma(a) = n$, P has no output in subject position (the two outputs on a being guarded by the abstraction on x), so that $\mathbf{Os}(P) = \emptyset$. P can however reduce to $P' = \overline{a}\langle\star\rangle \mid \overline{a}\langle\star\rangle$, with $\mathbf{Os}(P') = \{n, n\}$.

Thus we define the following multiset measure:

Definition 4.2.2 (Measure on processes in HOpi_ω)
Let P be a well-typed HOpi_ω process. We define $\mathbf{M}^{H\omega}(P) = \mathbf{Os}(P) \uplus \mathbf{Nfun}(P)$, where:

1. $\mathbf{Os}(P)$ *is the multiset of the levels of the channel names that are used in an output in P, without this output occurring in object position.*
2. $\mathbf{Nfun}(P)$ *is defined as the multiset union of all $\{k\}$, for all $v_1 \lfloor v_2 \rfloor$ occurring in P not within a message, such that v_1 is of type $T \to^k \diamond$.*

$\mathbf{M}^{H\omega}()$, with the lexicographical ordering, is well-founded.

We do not enter the details of the correctness proof for the type system for HOpi_ω, as it is subsumed by the proof of Theorem 4.2.19 in Section 4.2.

Proposition 4.2.3 (Soundness)
If $\Gamma \vdash_\omega P : n$ for some HOpi$_\omega$ process P, then P terminates.

Proof. *[Sketch] Proposition 4.2.3 is established by observing that $\mathbf{M}^{H\omega}(P)$ decreases at each step of transition:*

- *If the transition is a communication, the continuations of the processes involved in the communication contribute to the global measure the same way they did before communication, because a type preserving substitution is applied. $\mathbf{M}^{H\omega}(P)$ decreases because an output has been consumed.*
- *If the transition is a β-reduction involving a function of level k, a process of level strictly smaller than k is spawned in P. Therefore, all new messages and active function applications that contribute to the measure are of a level strictly smaller than l, and $\mathbf{M}^{H\omega}(P)$ decreases.*

□

The framework we study in this section is more powerful than those of Sections 4.1 and 4.2 for two main reasons. First, the language we work with is richer than HOpi_ω (which in turn extends HOpi_2). Second, we make a finer analysis of termination, by defining a more complex (and more expressive) type system.

The main extension to the process calculus, beyond the addition of primitive booleans and an if-then-else construct to manipulate these, is to include a primitive construct for replication in a higher-order formalism. This in principle does not add expressiveness to the calculus, because replication is encodable in HOpi_2 (using a process similar to Q_0 from Section 4.2). However, in terms of typability, having a primitive replication, and a dedicated typing rule for it, helps in dealing with examples. The type system to handle replication in presence of higher-order communications controls divergences that can arise both from self-emissions and from recursions in replications (as they appear in the setting of [DS06]).

We now turn to the description of the refinements we add to the type analysis.

Refining the calculus A first refinement we make to our termination analysis consists in attaching two pieces of information to a channel, instead of simply a level: a weight and a capacity (in the type systems seen before, the weight and the capacity are merged into a single information, namely the level). A channel a has a *weight*, which stands for the contribution of active outputs on a to the global weight of a process. For instance, in the process $U_1 = \overline{a_1}\langle U_2\rangle$, with $U_2 = \overline{b_1}\langle Q_1\rangle \mid \overline{b_2}\langle Q_2\rangle$, the global weight of U_2 is equal to the sum of the weights attached to names b_1 and b_2. We also associate a capacity to a channel a: this is an upper bound on the weight of processes that may be sent on a. U_1 is well-typed provided the capacity of a_1 is strictly greater than the global weight of U_2.

The distinction we make between the weight and capacity of a channel recalls the observations we have made above about the limitations of the type system of Section 4.1. Indeed, in the π-calculus processes $[\![R_1]\!]$ and $[\![R_2]\!]$ analysed in Section 4.1, the level of a (resp. of h_a) somehow would play the rôle of the weight (resp. of the capacity) associated to the encoding of the HOpi_2 channel a.

As a second extension to our type system, we represent the weight and the capacity attached to a channel, as well as the type attached to a process, using *multisets of natural numbers* in the way we did in Section 3.2.1. We also introduce, as previously, the possibility of treating sequences of input prefixes as a kind of 'single input action', that has the effect of decreasing the weight of the process being executed.

Let us show how this improvement copes with functional value-passing by studying an example in the formalism of HOpi_2. Consider a process of the form $P = a_1(X_1)...a_k(X_k).P'$. To type-check P, we make sure that the weight associated to the sequence of inputs is strictly greater than the weight associated to (some of the occurrences of) the process variables X_is in the continuation P'. The former quantity is equal

to $M_1^1 \uplus \cdots \uplus M_1^k$, if the weight associated to a_i is given by multiset M_1^i. To compute the latter quantity, we must take into account the multiplicity of the instances of the X_is in the process P'; this involves some technicalities, which we expose below (see Definition 4.2.5).

An Expressive Type System for Termination We now present an enriched version of HOpi_ω, that we call $\text{HOpi}^!_\omega$, for which we develop an expressive type system. The calculus $\text{HOpi}^!_\omega$ extends HOpi_ω by including primitive constructs for computation over booleans, and a replication operator. To present the grammar of $\text{HOpi}^!_\omega$, we rely on the same syntactic conventions as in the previous section, the set of values being extended with booleans `true` and `false`.

Values

$$v ::= \star \mid x \mid (x \mapsto P) \mid \texttt{true} \mid \texttt{false}$$

Processes

$$P ::= \mathbf{0} \mid (\boldsymbol{\nu} a)P \mid (P \mid P) \mid v\lfloor v \rfloor \mid a(x).P \mid \overline{a}\langle v\rangle.P \mid !a(x).P \mid \texttt{if } v \texttt{ then } P \texttt{ else } P$$

Note that, as usual, we restrict usages of the replication operator by applying it to inputs only.

Reduction is defined by giving the following rules for the reduction of the new operators. The definition of evaluation contexts does not change (as one does not want to reduce the branch of an `if then else` construct before evaluating the condition).

$$(\textbf{HKCondT})\ \frac{P \to P'}{\mathbf{E}[\texttt{if true then } P \texttt{ else } Q] \to \mathbf{E}[P']} \qquad (\textbf{HKCondF})\ \frac{Q \to Q'}{\mathbf{E}[\texttt{if false then } P \texttt{ else } Q] \to \mathbf{E}[Q']}$$

$$(\textbf{HKTrig})\ \frac{}{\mathbf{E}[\overline{a}\langle v\rangle.Q_1 \mid !a(x).Q_2] \to \mathbf{E}[Q_1 \mid Q_2\{v/x\} \mid !a(x).Q_2]}$$

$$(\textbf{HKCom})\frac{}{\mathbf{E}[\overline{a}\langle v\rangle.Q_1 \mid a(x).Q_2] \to \mathbf{E}[Q_1 \mid Q_2\{v/x\}]} \qquad (\textbf{HKBeta})\frac{}{\mathbf{E}[(x \mapsto P)\lfloor v \rfloor] \to \mathbf{E}[P\{v/x\}]}$$

Some care has to be taken when defining structural congruence. Since, as explained in Section 3.2.1, we treat sequences of inputs as a whole when type-checking processes, we are compelled to restrict the definition of structural congruence the same way: $\equiv$ is the smallest equivalence relation that satisfies the axioms given in Section 2, and that is closed under evaluation contexts (and not under prefix).

Types The types for channels in $\text{HOpi}^!_\omega$ are of the form $\sharp^{M_1,M_2}\, T$, where T ranges over types for values, defined as follows:

$$T ::= \mathbb{1}^{\emptyset} \mid \mathtt{B}^{\emptyset} \mid T \to^M \diamond$$

In order to introduce the typing rules, we need to extend the definition of well-formed types (Definition 4.2.1) to handle multisets:

Definition 4.2.4 (Well-formed value-types in $\text{HOpi}^!_\omega$)
We say that T is a well-formed value type *of $\text{HOpi}^!_\omega$ of weight M (written $\texttt{Lvl}(T) = M$), whenever either $T = \mathbb{1}^{\emptyset}$ or $T = \mathtt{B}^{\emptyset}$ and $M = \emptyset$, or T' is a well-formed value type of weight M', $T = T' \to^M \diamond$ and $M' <_{\text{mul}} M$.*

We sometimes use a shortened notation: when there is no ambiguity on the typing context, we shall write $\texttt{Lvl}(v_j) = M$ when $\Gamma(v_j) = T_j$ and $\texttt{Lvl}(T_j) = M$.

Definition 4.2.5 (M-contribution of x in P)
The M-contribution of x in P, *written* $\mathtt{Occ}(M,P,x)$, *is defined as follows:*

$$\begin{array}{l}
\mathtt{Occ}(M,\mathbf{0},x)=\emptyset\\
\mathtt{Occ}(M,v_1\lfloor v\rfloor,x)=\left\{\begin{array}{ll} M & \textit{if } v_1=x\\ \emptyset & \textit{if } v_1\neq x\end{array}\right.\\
\mathtt{Occ}(M,P_1\mid P_2,x)=\mathtt{Occ}(M,P_1,x)\uplus\mathtt{Occ}(M,P_2,x)\\
\mathtt{Occ}(M,a(x').P,x)=\left\{\begin{array}{ll} \emptyset & \textit{if } x'=x\\ \mathtt{Occ}(M,P,x) & \textit{otherwise}\end{array}\right.\\
\mathtt{Occ}(M,!a(x').P,x)=\emptyset\\
\mathtt{Occ}(M,\overline{a}\langle Q\rangle.P,x)=\mathtt{Occ}(M,(\boldsymbol{\nu}a)\ P,x)=\mathtt{Occ}(M,P,x)\\
\mathtt{Occ}(\mathtt{if}\ v\ \mathtt{then}\ P\ \mathtt{else}\ Q)=\max_{mul}(\mathtt{Occ}(M,P,x),\mathtt{Occ}(M,Q,x))\ .
\end{array}$$

$\mathtt{Occ}(M,P,x)$ is the multiset union of c copies of the multiset M, where c is the number of occurrences of x that appear neither in messages nor under a replication in P. This is reminiscent of the integer c appearing in Lemma 4.1.5. We may remark that if $M\leq_{\mathrm{mul}}N$, then $\mathtt{Occ}(M,P,x)\leq_{\mathrm{mul}}\mathtt{Occ}(N,P,x)$.

Figure 4.3 presents the rules that define the type system for $\mathrm{HOpi}^{!}_{\omega}$ — the typing judgement is written $\Gamma\vdash^{\omega,\kappa}P:N$.

The most complex rules are (**HKIn**) and (**HKRep**), where receiving processes are typed by analysing sequences of inputs. More precisely, in the former we compare the total weight associated to the channels involved in input sequences with their capacities. In the latter, two potential sources of divergence are controlled, we compare the total weight associated to the channels involved in input sequences with the sum of two entities: the capacities on one side, to prevent self-emission, and the weight of the continuation on the other side, to prevent loops due to recursive calls between replications.

It can be remarked that to handle polyadic communications, we associate the same capacity to all arguments in an input: for instance, to type-check a process of the form $a(x_1,x_2,x_3).P'$, rule (**In**) assumes that the capacity associated to a is strictly greater than the level of the types of variables x_1,x_2 and x_3 in the premise. This of course is rather rough – it would be easy to define a refinement assigning a specific capacity to each component of a tuple, at the cost of more complex types.

$\mathrm{HOpi}^{!,+}_{\omega}$, an auxiliary calculus to establish soundness. In order to prove that typable $\mathrm{HOpi}^{!}_{\omega}$ processes terminate, we rely, as above, on a measure which we define on typing derivations. As in Section 3.2.1 a measure defined as in the previous section would not be suitable, as it could grow, when the last prefix of an input sequence is consumed. Thus, we impose a variant of $\mathrm{HOpi}^{!}_{\omega}$, called $\mathrm{HOpi}^{!,+}_{\omega}$, which is a kind of *"$\mathrm{HOpi}^{!}_{\omega}$ with delayed substitutions"*. The syntax of $\mathrm{HOpi}^{!,+}_{\omega}$ is as follows:

$$\begin{array}{rcl}
P & ::= & \mathbf{0}\mid(\boldsymbol{\nu}c)P\mid P|P\mid v\lfloor v\rfloor\mid\overline{a}\langle v\rangle.P\mid\mathtt{if}\ v\ \mathtt{then}\ P\ \mathtt{else}\ P\\
& & \mid a_1(x_1)^{l_1}...a_k(x_k)^{l_k}.P\mid!a_1(x_1)^{l_1}...a_k(x_k)^{l_k}.P\ ,
\end{array}$$

where $(l_i)_{1\leq i\leq k}$ is a sequence of *annotations*. An annotation is either $\mathtt{free}$ or a $\mathrm{HOpi}^{!,+}_{\omega}$ value. We furthermore introduce a *well-formedness condition* to all $\mathrm{HOpi}^{!}_{\omega}$ processes we manipulate: we impose that in $a_1(x_1)^{l_1}...a_k(x_k)^{l_k}.P$ and $!a_1(x_1)^{l_1}...a_k(x_k)^{l_k}.P$, $l_i=\mathtt{free}$ must imply $l_{i+1}=\mathtt{free}$ for $i<k$, and that every input sequence appearing either guarded by a prefix, or inside in object position, or in annotation position, is annotated with $\mathtt{free}$.

The intuition is that, for instance, $a_1(x_1)^{v_1}.a_2(x_2)^{v_2}.a_3(x_3)^{\mathtt{free}}.P$ will evolve, after reception of value v_3 along channel a_3, into $((P\{v_1/x_1\})\{v_2/x_2\})\{v_3/x_3\}$: as long as the last prefix of a sequence of inputs has not been consumed, the substitutions involving the variables of the previous prefixes are not applied. One can remark that, as above, the last prefix $a_k(x_k)$ is always labelled with $\mathtt{free}$, because when the corresponding substitution $\{v_k/x_k\}$ is applied, the whole sequence of prefixes is consumed.

This idea is formalised by the following operation of 'triggering', that maps $\mathrm{HOpi}^{!,+}_{\omega}$ processes to their $\mathrm{HOpi}^{!}_{\omega}$ counterpart (in the following definition, we write $Q\{v/x\}\{w/y\}$ for $(Q\{v/x\})\{w/y\}$):

Typing values

$$(\textbf{HKUni})\frac{}{\Gamma \vdash^{\omega,\kappa} \star : \mathbb{1}^{\emptyset}} \qquad (\textbf{HKBool})\frac{}{\Gamma \vdash^{\omega,\kappa} \mathtt{true}, \mathtt{false} : \mathtt{B}^{\emptyset}} \qquad (\textbf{HKVar})\frac{\Gamma(x) = T}{\Gamma \vdash^{\omega,\kappa} x : T}$$

$$(\textbf{HKFun})\frac{\Gamma(x) = T \qquad \Gamma \vdash^{\omega,\kappa} P : N}{\Gamma \vdash^{\omega,\kappa} x \mapsto P : T \rightarrow^{\mathrm{succ}(N)} \diamond}$$

Typing processes

$$(\textbf{HKNil})\frac{}{\Gamma \vdash^{\omega,\kappa} \mathbf{0} : \emptyset} \qquad (\textbf{HKRes})\frac{\Gamma(a) = \sharp^{M_1,M_2} T \qquad \Gamma \vdash^{\omega,\kappa} P : N}{\Gamma \vdash^{\omega,\kappa} (\boldsymbol{\nu} a)\ P : N}$$

$$(\textbf{HKPar})\frac{\Gamma \vdash^{\omega,\kappa} P_1 : N_1 \qquad \Gamma \vdash^{\omega,\kappa} P_2 : N_2}{\Gamma \vdash^{\omega,\kappa} P_1 \mid P_2 : N_1 \uplus N_2} \qquad (\textbf{HKApp})\frac{\Gamma \vdash^{\omega,\kappa} v_1 : T \rightarrow^{M} \diamond \qquad \Gamma \vdash^{\omega,\kappa} v_2 : T}{\Gamma \vdash^{\omega,\kappa} v_1 \lfloor v_2 \rfloor : M}$$

$$(\textbf{HKIf})\frac{\Gamma \vdash^{\omega,\kappa} v : \mathtt{B}^{\emptyset} \qquad \Gamma \vdash^{\omega,\kappa} P_1 : N_1 \qquad \Gamma \vdash^{\omega,\kappa} P_2 : N_2}{\Gamma \vdash^{\omega,\kappa} \mathtt{if}\ v\ \mathtt{then}\ P\ \mathtt{else}\ Q : \max_{mul}(N_1, N_2)}$$

$$(\textbf{HKIn})\frac{\begin{array}{c}\Gamma(x_1) = T_1, \ldots, \Gamma(x_k) = T_k \qquad \Gamma \vdash^{\omega,\kappa} P : N \qquad \forall i, \Gamma(a_i) = \sharp^{M_1^i, M_2^i} T_i \\ \forall i, \mathtt{Lvl}(T_i) <_{\mathrm{mul}} M_2^i \qquad \biguplus M_1^i >_{\mathrm{mul}} \biguplus \mathtt{Occ}(M_2^i, x_i, P)\end{array}}{\Gamma \vdash^{\omega,\kappa} a_1(x_1)...a_k(x_k).P : N}$$

$$(\textbf{HKOut})\frac{\begin{array}{c}\Gamma(a) = \sharp^{M_1,M_2} T \qquad \Gamma \vdash^{\omega,\kappa} P : N \\ \Gamma \vdash^{\omega,\kappa} v : T \qquad M_2 >_{\mathrm{mul}} \mathtt{Lvl}(T)\end{array}}{\Gamma \vdash^{\omega,\kappa} \overline{a}\langle v \rangle.P : M_1 \uplus N}$$

$$(\textbf{HKRep})\frac{\begin{array}{c}\Gamma(x_1) = T_1 \ldots \Gamma(x_k) = T_k \Gamma \vdash^{\omega,\kappa} P : N \qquad \forall i, \Gamma(a_i) = \sharp^{M_1^i, M_2^i} T_i \\ \forall i, \mathtt{Lvl}(T_i) <_{\mathrm{mul}} M_2^i \qquad \biguplus M_1^i >_{\mathrm{mul}} (\biguplus \mathtt{Occ}(M_2^i, x_i, P)) \uplus N\end{array}}{\Gamma \vdash^{\omega,\kappa} !a_1(x_1)...a_k(x_k).P : \emptyset}$$

Figure 4.3: Typing Rules for HOpi$^{!}_{\omega}$

Definition 4.2.6 (From HOpi$^{!,+}_{\omega}$ to HOpi$^{!}_{\omega}$, and back) *We introduce an operator* $\mathbf{AnRem}^H()$*, mapping HOpi$^{!,+}_{\omega}$ processes (resp. values) to HOpi$^{!}_{\omega}$ processes (resp. values), defined by:*

$$\mathbf{AnRem}^H(\mathbf{0}) = \mathbf{0} \qquad \mathbf{AnRem}^H((\boldsymbol{\nu} c)\ P) = (\boldsymbol{\nu} c)\ \mathbf{AnRem}^H(P) \qquad \mathbf{AnRem}^H(\overline{a}\langle v\rangle.P) = \overline{a}\langle v\rangle.\mathbf{AnRem}^H(P)$$

$$\mathbf{AnRem}^H(v_1\lfloor v_2\rfloor) = \mathbf{AnRem}^H(v_1)\lfloor \mathbf{AnRem}^H(v_2)\rfloor \qquad \mathbf{AnRem}^H(x \mapsto P) = x \mapsto \mathbf{AnRem}^H(P)$$

$$\mathbf{AnRem}^H(x) = x \qquad \mathbf{AnRem}^H(P_1 \mid P_2) = \mathbf{AnRem}^H(P_1) \mid \mathbf{AnRem}^H(P_2)$$

$$\mathbf{AnRem}^H(\texttt{if } v \texttt{ then } P \texttt{ else } Q) = \texttt{if } v \texttt{ then } \mathbf{AnRem}^H(P) \texttt{ else } \mathbf{AnRem}^H(Q)$$

$$\begin{array}{l}
\mathbf{AnRem}^H(!a_1(x_1)^{\texttt{free}}...a_k(x_k)^{\texttt{free}}.P) \\
\quad =!a_1(x_1)...a_k(x_k).\mathbf{AnRem}^H(P) \\
\mathbf{AnRem}^H(!a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{\texttt{free}}...a_k(x_k)^{\texttt{free}}.P) \\
\quad = a_i(x_i)...a_k(x_k).(\mathbf{AnRem}^H(P)\{v_1/x_1\}\ldots\{v_{i-1}/x_{i-1}\}) \quad \textit{with } 1 < i \leq k \\
\mathbf{AnRem}^H(a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{\texttt{free}}...a_k(x_k)^{\texttt{free}}.P) \\
\quad = a_i(x_i)...a_k(x_k).(\mathbf{AnRem}^H(P)\{v_1/x_1\}\ldots\{v_{i-1}/x_{i-1}\}) \quad \textit{with } 1 \leq i \leq k
\end{array}$$

If P is a HOpi$^{!}_{\omega}$ process, we write $\texttt{free}(P)$ *for the HOpi$^{!,+}_{\omega}$ process obtained from P by decorating all input prefixes with annotation* `free`*.*

Note that $\mathbf{AnRem}^H(\texttt{free}(P)) = P$, and $\mathbf{AnRem}^H(Q)\{v/x\} = \mathbf{AnRem}^H(Q\{v/x\})$.

To define the operational semantics of HOpi$^{!,+}_{\omega}$, we keep rules (**HKBeta**), (**HKCong**), (**HKCondT**), (**HKCondF**) unchanged, and introduce the rules of Figure 4.4 (the reduction relation on HOpi$^{!,+}_{\omega}$ is written $\rightarrow^{\kappa}$). According to the explanations above, these rules enforce that substitutions are delayed until the last prefix in a sequence of input prefixes is consumed. More precisely, rules (**HKUnr**) and (**HKRep**) accumulate substitutions along sequences of prefixes, while rules (**HKEndUnr**) and (**HKEndRep**) are used to trigger the last prefix of a sequence of inputs.

Note that we treat differently replicated and non replicated sequences of input prefixes, as the condition associated to typability is different in the typing rules (**HKIn**κ) and (**HKRep**κ) (given below).

Fact 4.2.7 (Well-formedness preservation)
If $P \rightarrow^{\kappa} P'$ and P satisfies the well-formedness condition introduced above, then so does P'.

Proof. *Easily done by induction on the derivation of $P \rightarrow^{\kappa} P'$.* □

Fact 4.2.8 (Annotated calculus - Context)
If $P = \mathbf{E}[P']$ and $P = \mathbf{Rem}(Q)$, then there exists $\mathbf{E}_1$ s.t. $Q = \mathbf{E}_1[Q']$ s.t. $P' = \mathbf{Rem}(Q')$.

Proof.
Easily done by structural induction over $\mathbf{E}$.

□

Lemma 4.2.9 (Annotated calculus - Simulation)
Let $\leq^{\mathbf{HK}}$ be the relation defined on HOpi$^{!}_{\omega} \times$ HOpi$^{!,+}_{\omega}$ by: $P \leq^{\mathbf{HK}} Q$ iff $P = \mathbf{AnRem}^H(Q)$. Then $\leq^{\mathbf{HK}}$ is a simulation, that is, for any $P \leq^{\mathbf{HK}} Q$, whenever $P \rightarrow P'$, there exists Q' s.t. $Q \rightarrow^{\kappa} Q'$ and $P' \leq^{\mathbf{HK}} Q'$.

$\leq^{\mathbf{HK}}$ is actually a (strong) *bisimulation* [SW01]. We however prove only this simulation result, as it is sufficient to deduce that if $P = \mathbf{AnRem}^H(Q)$ and P diverges, then so does Q, which is what we shall need, as in Section 3.2.1.

Proof. *We reason by induction on the derivation of $P \rightarrow P'$. Cases* (**HKBeta**), (**HKCondT**) *and* (**HKCondF**) *are easily treated using Definition 4.2.6. The remaining cases are more interesting:*

$$(\textbf{HKTrRep})\ \frac{}{\overline{a_1}\langle v_1\rangle.Q \mid !a_1(x_1)^{\mathtt{free}}.a_2(x_2)^{\mathtt{free}}..a_k(x_k)^{\mathtt{free}}.P \rightarrow^{\kappa} Q \mid !a_1(x_1)^{v_1}.a_2(x_2)^{\mathtt{free}}..a_k(x_k)^{\mathtt{free}}.P \mid !a_1(x_1)^{\mathtt{free}}.a_2(x_2)^{\mathtt{free}}..a_k(x_k)^{\mathtt{free}}.P}$$

$$(\textbf{HKPrUnr})\ \frac{1 \leq i < k}{\overline{a_i}\langle v_i\rangle.Q \mid a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{\mathtt{free}}.a_{i+1}(x_{i+1})^{\mathtt{free}}...a_k(x_k)^{\mathtt{free}}.P \rightarrow^{\kappa} Q \mid a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{v_i}.a_{i+1}(x_{i+1})^{\mathtt{free}}...a_k(x_k)^{\mathtt{free}}.P}$$

$$(\textbf{HKPrRep})\ \frac{1 < i < k}{\overline{a_i}\langle v_i\rangle.Q \mid !a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{\mathtt{free}}.a_{i+1}(x_{i+1})^{\mathtt{free}}...a_k(x_k)^{\mathtt{free}}.P \rightarrow^{\kappa} Q \mid !a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{v_i}.a_{i+1}(x_{i+1})^{\mathtt{free}}...a_k(x_k)^{\mathtt{free}}.P}$$

$$(\textbf{HKEndUnr})\ \frac{}{\overline{a_k}\langle v_k\rangle.Q \mid a_1(x_1)^{v_1}...a_{k-1}(x_{k-1})^{v_{k-1}}.a_k(x_k)^{\mathtt{free}}.P \ \rightarrow^{\kappa} Q \mid P\{v_1/x_1\}\ldots\{v_{k-1}/x_{k-1}\}\{v_k/x_k\}}$$

$$(\textbf{HKEndRep})\ \frac{}{\overline{a_k}\langle v_k\rangle.Q \mid !a_1(x_1)^{v_1}...a_{k-1}(x_{k-1})^{v_{k-1}}.a_k(x_k)^{\mathtt{free}}.P \ \rightarrow^{\kappa} Q \mid P\{v_1/x_1\}\ldots\{v_{k-1}/x_{k-1}\}\{v_k/x_k\}}$$

Figure 4.4: Communication Rules for $\text{HOpi}_{\omega}^{!,+}$

- *Case* (**HKCom**). *We have* $P = \mathbf{E}[\overline{a}\langle v\rangle.P_1 \mid a(x).P_2]$ *and* $P' = \mathbf{E}[P_1 \mid P_2\{v/x\}]$. *By Definition 4.2.6 and Fact 4.2.8, we deduce that* $Q = \mathbf{E}_1[\overline{a}\langle v\rangle.Q_1 \mid Q_0]$ *where* $P_1 = \mathbf{AnRem}^H(Q_1)$. *We discuss on the form of* Q_0, *according to Definition 4.2.6:*
  - *Case* $Q_0 = !a_1(x_1)^{v_1}..a_{i-1}(x_{i-1})^{v_{i-1}}.a(x)^{\mathtt{free}}.a_{i+1}(x_{i+1})^{\mathtt{free}}..a_k(x_k)^{\mathtt{free}}.Q_2$, *with* $1 < i < k$. *We have*
    $$P_2 = a_{i+1}(x_{i+1})\ldots a_k(x_k).(\mathbf{AnRem}^H(Q_2)\{v_1/x_1\}\ldots\{v_{i-1}/x_{i-1}\}).$$
    Process Q *can perform a reduction, using rule* (**HKPrRep**), *to*
    $$Q' = \mathbf{E}_1[Q_1 \mid !a_1(x_1)^{v_1}..a_{i-1}(x_{i-1})^{v_{i-1}}.a(x)^{v}.a_{i+1}(x_{i+1})^{\mathtt{free}}...a_k(x_k)^{\mathtt{free}}.Q_2].$$
    We have $\mathbf{AnRem}^H(Q') =$
    $$\mathbf{E}[\mathbf{AnRem}^H(Q_1) \mid a_{i+1}(x_{i+1})...a_k(x_k).(\mathbf{AnRem}^H(Q_2)\{v_1/x_1\}\ldots\{v_{i-1}/x_{i-1}\}\{v/x\})] = P'$$
    (notice that we have $(a_j(x_j).S)\{v/x\} = a_j(x_j).(S\{v/x\})$*).*
  - *Case* $Q_0 = a_1(x_1)^{v_1}..a_{i-1}(x_{i-1})^{v_{i-1}}.a(x)^{\mathtt{free}}.a_{i+1}(x_{i+1})^{\mathtt{free}}..a_k(x_k)^{\mathtt{free}}.Q_2$, *with* $1 \leq i < k$. *We reason similarly using rule* (**HKPrUnr**).
  - *Case* $Q_0 = !a_1(x_1)^{v_1}..a_{k-1}(x_{k-1})^{v_{k-1}}.a(x)^{\mathtt{free}}.Q_2$. *We have*
    $$P_2 = \mathbf{AnRem}^H(Q_2)\{v_1/x_1\}\ldots\{v_{k-1}/x_{k-1}\}.$$
    Process Q *can reduce, using rule* (**HKEndRep**), *to*
    $$Q' = \mathbf{E}_1[Q_1 \mid Q_2\{v_1/x_1\}\ldots\{v_{k-1}/x_{k-1}\}\{v/x\}]$$
    and $\mathbf{AnRem}^H(Q') = P'$.

$$(\mathbf{HKIn}\kappa)\ \frac{\begin{array}{c}\Gamma \vdash^{+} v_1 : T_1 \qquad \ldots \qquad \Gamma \vdash^{+} v_i : T_i \\ \Gamma(x_1) = T_1, \ldots, \Gamma(x_k) = T_k \qquad \Gamma \vdash^{\omega,\kappa} P : N \qquad \biguplus_{1\leq j\leq k} M_1^j >_{\mathrm{mul}} \biguplus_{1\leq j\leq k} \mathtt{Occ}(M_2^j, x_j, P) \\ \forall 1 \leq j \leq k, \mathtt{Lvl}(T_j) <_{\mathrm{mul}} M_2^j \qquad \Gamma(a_j) = \sharp^{M_1^j, M_2^j} T_j\end{array}}{\Gamma \vdash^{+} a_1(x_1)^{v_1} \ldots a_i(x_i)^{v_i}.a_{i+1}(x_{i+1})^{\mathtt{free}} \ldots a_k(x_k)^{\mathtt{free}}.P : N}$$

$$(\mathbf{HKRep}\kappa)\ \frac{\begin{array}{c}\Gamma \vdash^{+} v_1 : T_1 \qquad \ldots \qquad \Gamma \vdash^{+} v_i : T_i \\ \Gamma(x_1) = T_1, \ldots, \Gamma(x_k) = T_k \qquad \Gamma \vdash^{+} P : N \biguplus_{1\leq j\leq k} M_1^j >_{\mathrm{mul}} \biguplus_{1\leq j\leq k} \mathtt{Occ}(M_2^j, x_j, P) \uplus N \\ \forall 1 \leq j \leq k, \mathtt{Lvl}(T_j) <_{\mathrm{mul}} M_2^j \qquad \Gamma(a_j) = \sharp^{M_1^j, M_2^j} T_j\end{array}}{\Gamma \vdash^{+} !a_1(x_1)^{v_1} \ldots a_i(x_i)^{v_i}.a_{i+1}(x_{i+1})^{\mathtt{free}} \ldots a_k(x_k)^{\mathtt{free}}.P : \emptyset}$$

Figure 4.5: Dedicated Typing Rules for $\mathrm{HOpi}_{\omega}^{!,+}$

- *Case* (**HKTrig**). *We have* $P = \mathbf{E}[\overline{a}\langle v\rangle.P_1 \mid !a(x).P_2]$ *and* $P' = \mathbf{E}[P_1 \mid P_2\{v/x\} \mid !a(x).P_2]$. *Using Definition 4.2.6 and Fact 4.2.8, we deduce that* $Q = \mathbf{E}_1[\overline{a}\langle v\rangle.Q_1 \mid !a(x)^{\mathtt{free}}.a_2(x_2)^{\mathtt{free}}..a_k(x_k)^{\mathtt{free}}.Q_2]$, *with* $P_1 = \mathbf{AnRem}^H(Q_1)$ *and* $P_2 = a_2(x_2)...a_k(x_k).\mathbf{AnRem}^H(Q_2)$. *Process* Q *can perform a reduction, using rule* (**HKTrRep**), *to* $Q' = \mathbf{E}_1[Q_1 \mid !a(x)^{v}.a_2(x_2)^{\mathtt{free}}..a_k(x_k)^{\mathtt{free}}.Q_2 \mid !a(x)^{\mathtt{free}}.a_2(x_2)^{\mathtt{free}}..a_k(x_k)^{\mathtt{free}}.Q_2]$ *We then have* $\mathbf{AnRem}^H(Q') = \mathbf{E}_1[\mathbf{AnRem}^H(Q_1) \mid a_2(x_2)...a_k(x_k).\mathbf{AnRem}^H(Q_2)\{v/x\}$ $\mid !a(x).a_2(x_2)..a_k(x_k).\mathbf{AnRem}^H(Q_2)] = P'$.

□

After having defined reduction in $\mathrm{HOpi}_{\omega}^{!,+}$, we now turn to typing. The basic idea is to start with a typable $\mathrm{HOpi}_{\omega}^{!}$ process, and execute it 'as a $\mathrm{HOpi}_{\omega}^{!,+}$ term'. In doing so, we keep a representation of the whole sequence of prefixes before it is totally consumed, and this allows us to reconstruct the original typing derivation along reductions. The type system for $\mathrm{HOpi}_{\omega}^{!,+}$ is thus very close to the system for $\mathrm{HOpi}_{\omega}^{!}$.

Typing judgements for $\mathrm{HOpi}_{\omega}^{!,+}$ processes (written $\Gamma \vdash^{+} P : N$) are derived using the rules of Figure 4.3, where rules (**HKIn**) and (**HKRep**) are replaced respectively with the rules presented on Figure 4.5, in order to handle annotations.

Accordingly, the contribution $\mathtt{Occ}(M, P, x)$, for P a $\mathrm{HOpi}_{\omega}^{!,+}$ term, is defined as in Definition 4.2.5 — in particular, $\mathtt{Occ}(M, a^l(y).P, x) = \mathtt{Occ}(M, P, x)$.

Remark 4.2.10 (Mapping of typing derivations) *The careful reader may have noticed that different typing derivations for a* $\mathrm{HOpi}_{\omega}^{!}$ *term* P *can be mapped to the same typing derivation in* $\mathrm{HOpi}_{\omega}^{!,+}$, *for* $\mathtt{free}(P)$. *For instance, if* $P = a_1(x_1).a_2(x_2).a_3(x_3).P'$, *we can choose to apply rule* (**HKIn**) *once, to the sequence of prefixes* $a_1(x_1).a_2(x_2).a_3(x_3)$ *(with continuation process* P'*), but we can also, alternatively, apply* (**HKIn**) *first with* $a_1(x_1).a_2(x_2)$, *the continuation process* $a_3(x_3).P'$ *being typed using a second application of* (**HKIn**). *Both these derivations are mapped to the same 'maximal' typing derivation in* $\mathrm{HOpi}_{\omega}^{!,+}$, *where rule* (**HKIn**$\kappa$) *is used only once. This has no important consequence on our reasonings, since, intuitively, a typing derivation that relies on several applications of rule* (**HKIn**) *for a given sequence of prefixes can always be replaced by the 'maximal' derivation, where* (**HKIn**) *is applied only once.*

Fact 4.2.11 (Typing equivalence)
If P *is a* $\mathrm{HOpi}_{\omega}^{!}$ *process, then* $\Gamma \vdash^{\omega,\kappa} P : N$ *if and only if* $\Gamma \vdash^{+} \mathtt{free}(P) : N$.

Proof. *Easily done by induction on the typing judgement* $\Gamma \vdash^{\omega,\kappa} P : N$. □

We are now ready to prove soundness of our type system for $\mathrm{HOpi}_{\omega}^{!,+}$. As above, we introduce for this two measures on $\mathrm{HOpi}_{\omega}^{!,+}$ processes. These measures are the counterpart of the measures presented in Definition 4.1.3.

Lemma 4.2.12 (Context Typing)
If $\Gamma \vdash^+ \mathbf{E}[P] : N$ *then:*

1. $\Gamma \vdash^+ P : N'$ *for some* $N' \leq_{\text{mul}} N$.
2. *For all* P_0 *s.t.* $\Gamma \vdash^+ P_0 : N_0$, *we can derive* $\Gamma \vdash^+ \mathbf{E}[P_0] : N_{(0)}$ *for some* $N_{(0)}$.

Proof. *By structural induction over* $\mathbf{E}$,

- *Case* $[\,]$. *Condition 1 holds trivially and condition 2 holds by setting* $N_{(0)} = N_0$.
- *Case* $(\boldsymbol{\nu} a)\ \mathbf{E}_2$. *Condition 1 holds with* $N' = N$ *and condition 2 holds by setting* $N_{(0)} = N_0$.
- *Case* $\mathbf{E} = \mathbf{E}_2 \mid P_1$. *We derive* $\Gamma \vdash^+ \mathbf{E}_2[P] : N_2$ *and* $\Gamma \vdash^+ P_1 : N_1$ *with* $N = N_2 \uplus N_1$. *The induction hypothesis gives* $\Gamma \vdash^+ P : N'$ *for some* N', *thus we get condition 1. The induction hypothesis also gives* $\Gamma \vdash^+ \mathbf{E}_2[P_0] : N_{(2)}$ *for some* $N_{(2)}$. *We set* $N_{(0)} = N_{(2)} \uplus N_1$ *and we get condition 2.*

□

Definition 4.2.13 (Measures) *If* $\Gamma \vdash^+ Q : N$, *we inductively define* $\mathbf{M}^\omega_I(Q)$ *as follows:*

$$\mathbf{M}^\omega_I(\mathbf{0}) = \emptyset \qquad \mathbf{M}^\omega_I(Q_1 \mid Q_2) = \mathbf{M}^\omega_I(Q_1) \uplus \mathbf{M}^\omega_I(Q_2) \qquad \mathbf{M}^\omega_I((\boldsymbol{\nu} a)\ Q_1) = \mathbf{M}^\omega_I(Q_1)$$

$$\mathbf{M}^\omega_I(\overline{a}\langle v\rangle.Q_1) = \mathbf{M}^\omega_I(Q_1) \uplus \{M_1\} \text{ if } \Gamma(a) = \sharp^{M_1,M_2} T \qquad \mathbf{M}^\omega_I(x\lfloor v_2 \rfloor) = \emptyset$$

$$\mathbf{M}^\omega_I(v_1\lfloor v_2 \rfloor) = \{M\} \text{ if } \Gamma \vdash^+ v_1 : T \to^M \diamond$$

$$\mathbf{M}^\omega_I(a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{\mathtt{free}}...a_k(x_k)^{\mathtt{free}}.Q_1) = \mathbf{M}^\omega_I(Q_1) \uplus \{M^1_1\} \uplus \cdots \uplus \{M^{i-1}_y 1\} \text{ if for all } j \leq k, \Gamma(a_j) = \sharp^{M^j_1,M^j_2} T_j$$

$$\mathbf{M}^\omega_I(!a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{\mathtt{free}}...a_k(x_k)^{\mathtt{free}}.Q_1) = \{M^1_1\} \uplus \cdots \uplus \{M^{i-1}_1\} \text{ if for all } j,\ \Gamma(a_j) = \sharp^{M^j_1,M^j_2} T_j$$

In order to handle delayed substitutions, we introduce another measure, noted $\mathbf{M}^\omega_{II}(Q)$, *and defined like* $\mathbf{M}^\omega_I(Q)$ *except for the following cases:*

$$\mathbf{M}^\omega_{II}(a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{\mathtt{free}}...a_k(x_k)^{\mathtt{free}}.Q_1) = \mathbf{M}^\omega_{II}(Q_1)$$

$$\mathbf{M}^\omega_{II}(!a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{\mathtt{free}}...a_k(x_k)^{l_k}.Q_1) = \emptyset$$

Both measures are easily extended to evaluation contexts by setting $\mathbf{M}^\omega_I([\,]) = \mathbf{M}^\omega_{II}([\,]) = \emptyset$.

Note that $\mathbf{M}^\omega_I()$ and $\mathbf{M}^\omega_{II}()$ coincide on processes of the form $\mathtt{free}(P)$ (for some HOpi$^!_\omega$ process P). As usual, the following facts describe how measures handle evaluation contexts and structural congruence.

Fact 4.2.14 (Context and Measures)
If $\Gamma \vdash^+ \mathbf{E}[P] : N$, *then*

1. $\mathbf{M}^\omega_I(\mathbf{E}[P]) = \mathbf{M}^\omega_I(\mathbf{E}) \uplus \mathbf{M}^\omega_I(P)$.
2. $\mathbf{M}^\omega_{II}(\mathbf{E}[P]) = \mathbf{M}^\omega_{II}(\mathbf{E}) \uplus \mathbf{M}^\omega_{II}(P)$.

Proof.

1. *By structural induction over* $\mathbf{E}$*:*
   - *Case* $[\,]$. *Then* $\mathbf{M}^\omega_I(\mathbf{E}[P]) = \mathbf{M}^\omega_I(P) = \mathbf{M}^\omega_I(P) \uplus \mathbf{M}^\omega_I([\,])$.

- *Case* $(\nu a)\ \mathbf{E}_2$. *Then* $\mathbf{M}^\omega_I(\mathbf{E}[P]) = \mathbf{M}^\omega_I((\nu a)\ \mathbf{E}_2[P])$ *which is, by Definition 4.2.13,* $\mathbf{M}^\omega_I(\mathbf{E}_2[P])$. *We use the induction hypothesis to get* $\mathbf{M}^\omega_I(\mathbf{E}_2[P]) = \mathbf{M}^\omega_I(\mathbf{E}_2) \uplus \mathbf{M}^\omega_I(P)$. *As, by Definition 4.2.13,* $\mathbf{M}^\omega_I(\mathbf{E}) = \mathbf{M}^\omega_I(\mathbf{E}_2)$, *we conclude.*
- *Case* $\mathbf{E}_2 \mid P_1$. *Then* $\mathbf{M}^\omega_I(\mathbf{E}[P]) = \mathbf{M}^\omega_I(\mathbf{E}_2[P] \mid P_1)$ *which is, by Definition 4.2.13,* $\mathbf{M}^\omega_I(\mathbf{E}_2[P]) \uplus \mathbf{M}^\omega_I(P_1)$. *We use the induction hypothesis to get* $\mathbf{M}^\omega_I(\mathbf{E}_2[P]) = \mathbf{M}^\omega_I(\mathbf{E}_2) \uplus \mathbf{M}^\omega_I(P)$. *As, by Definition 4.2.13,* $\mathbf{M}^\omega_I(\mathbf{E}) = \mathbf{M}^\omega_I(\mathbf{E}_2) \uplus \mathbf{M}^\omega_I(P_1)$, *we conclude.*

2. *The proof for* $\mathbf{M}^\omega_{II}()$ *is the same.*

□

Fact 4.2.15 (Annotated calculus - Congruence)
If $Q \equiv Q'$ *then the following propositions are equivalent:*

1. $\Gamma \vdash^+ Q : N$, $\mathbf{M}^\omega_I(Q) = M_1$ *and* $\mathbf{M}^\omega_{II}(Q) = M_2$
2. $\Gamma' \vdash^+ Q' : N'$, $\mathbf{M}^\omega_I(Q') = M_1$ *and* $\mathbf{M}^\omega_{II}(Q') = M_2$.

Proof. *By induction on the derivation of* $Q \equiv Q'$. □

Let us compare the following proof of subject substitution with the one presented in Section 3.2.1. The main difference is that when messages communicated are names, the input sequence itself can change when it is partially consumed. For instance, in the name-passing setting, $!a(x)^{\mathtt{free}}.x^{\mathtt{free}}.b^{\mathtt{free}}.P$ becomes $!a(x)^{\mathtt{ok}}.v^{\mathtt{free}}.b^{\mathtt{free}}.P$ after being triggered by $\overline{a}\langle v\rangle$. Yet, in the higher-order setting, messages are functional values and the only variables that can be instantiated are found in the continuation of the process. More formally, we always have $(!a_1(x_1)^{l_1}.\ldots.a_n(x_n)^{l_n}.P)\{v/x\} = !a_1(x_1)^{l_1}.\ldots.a_n(x_n)^{l_n}.(P\{v/x\})$. This yields a simpler subject substitution proof.

Lemma 4.2.16 (Subject substitution)
Suppose that $\Gamma \vdash^+ w : T$ *and* $\Gamma(x) = T$, *where* $T = T' \to^M \diamond$.

1. *If* $\Gamma \vdash^+ P : N$, *then* $\Gamma \vdash^+ P\{w/x\} : N$, $\mathbf{M}^\omega_I(P\{w/x\}) = \mathbf{M}^\omega_I(P) \uplus \mathtt{Occ}(M, P, x)$ *and* $\mathbf{M}^\omega_{II}(P\{w/x\}) = \mathbf{M}^\omega_{II}(P) \uplus \mathtt{Occ}(M, P, x)$.
2. *If* $\Gamma \vdash^+ v : T_0$, *then* $\Gamma \vdash^+ v\{w/x\} : T_0$.

Proof. *By induction on the typing judgement:*

- *Cases* (**HKNil**), (**HKRes**), (**HKPar**), (**HKIf**), (**HKUni**), *and* (**HKBool**) *are easily done using the induction hypotheses when needed and Definition 4.2.13.*
- *Case* (**HKApp**). *We have* $P = v_1\lfloor v_2\rfloor$. *We use the induction hypothesis and we get* $\Gamma \vdash^+ v_1\{w/x\} : T_1 \to^N \diamond$ *and* $\Gamma \vdash^+ v_2\{w/x\} : T_1$. *Using the rule* (**HKApp**) *and the fact that* $(v_1\lfloor v_2\rfloor)\{w/x\} = (v_1\{w/x\})\lfloor(v_2\{w/x\})\rfloor$, *we derive* $\Gamma \vdash^+ (v_1\lfloor v_2\rfloor)\{w/x\} : N$. *We distinguish three cases to compute the measures according to Definition 4.2.13:*
  - *if* $v_1 = x$ *then* $N = M$. *Definition 4.2.5 gives* $\mathtt{Occ}(M, x\lfloor v_2\rfloor, x) = M$ *and we have* $\mathbf{M}^\omega_I(x\lfloor v_2\rfloor) = \mathbf{M}^\omega_{II}(x\lfloor v_2\rfloor) = \emptyset$ *and* $\mathbf{M}^\omega_I(w\lfloor v_2\{w/x\}\rfloor) = \mathbf{M}^\omega_{II}(w\lfloor v_2\{w/x\}\rfloor) = M$.
  - *if* $v_1 = y$ *and* $y \neq x$, *then Definition 4.2.5 gives* $\mathtt{Occ}(M, x\lfloor v_2\rfloor, x) = M$ *and* $\mathbf{M}^\omega_I(y\lfloor v_2\{w/x\}\rfloor) = \mathbf{M}^\omega_{II}(y\lfloor v_2\{w/x\}\rfloor) = \mathbf{M}^\omega_I(y\lfloor v_2\rfloor) = \mathbf{M}^\omega_{II}(y\lfloor v_2\rfloor)$, *all these quantities being equal to* $\emptyset$.
  - *if* v_1 *is not a variable then Definition 4.2.5 gives* $\mathtt{Occ}(M, x\lfloor v_2\rfloor, x) = \emptyset$ *and Definition 4.2.13 gives* $\mathbf{M}^\omega_I(v_1\lfloor v_2\{w/x\}\rfloor) = \mathbf{M}^\omega_{II}(v_1\lfloor v_2\{w/x\}\rfloor) = \mathbf{M}^\omega_I(v_1\lfloor v_2\rfloor) = \mathbf{M}^\omega_{II}(v_1\lfloor v_2\rfloor) = N$.

- *Case* (**HKIn**κ). *We have* $P = a_1(x_1)^{v_1}...a_i(x_i)^{v_i}.a_{i+1}(x_{i+1})^{\mathtt{free}}...a_k(x_k)^{\mathtt{free}}.P_1$. *We derive, from rule* (**HKIn**κ), $\Gamma \vdash^+ P_1 : N$, $\Gamma(x_j) = T_i$ *and* $\Gamma \vdash^+ v_j : T_j$, *with* $\Gamma(a_j) = \sharp^{M_1^j, M_2^j}\ T_j$, $M_2^j >_{\mathrm{mul}} \mathtt{Lvl}(v_j)$ *and* $\biguplus M_1^j >_{\mathrm{mul}} \biguplus \mathtt{Occ}(M_2^j, x_j, P_1)$. *Since* $x \neq x_j$ *and* $x_j \notin w$, *we have* $\mathtt{Occ}(M_2^j, x_j, P_1) = \mathtt{Occ}(M_2^j, x_j, P_1\{w/x\})$ *(no occurrence of* x_j *is added or removed by replacing the occurrences of* x *by* w *in* P_1*). The induction hypothesis gives* $\Gamma \vdash^+ P_1\{w/x\} : N$, *for all* $1 \leq j \leq i$, $\Gamma \vdash^+ v_j\{w/x\} : T_j$, $\mathbf{M}_I^\omega(P_1) = \mathbf{M}_I^\omega(P_1) \uplus \mathtt{Occ}(M, P_1, x)$ *and* $\mathbf{M}_{II}^\omega(P_1) = \mathbf{M}_{II}^\omega(P_1) \uplus \mathtt{Occ}(M, P_1, x)$. *All necessary side conditions are satisfied, and we can apply* (**HKIn**κ) *and derive*

$$\Gamma \vdash^+ a_1(x_1)^{v_1\{w/x\}}...a_i(x_i)^{v_i\{w/x\}}.a_{i+1}(x_{i+1})^{\mathtt{free}}...a_k(x_k)^{\mathtt{free}}.P_1\{w/x\} : N$$

We can then use Definitions 4.2.13 and 4.2.5 to conclude that $\mathbf{M}_I^\omega(P') = \mathbf{M}_I^\omega(P) \uplus \mathtt{Occ}(M, P, x)$, *and* $\mathbf{M}_{II}^\omega(P') = \mathbf{M}_{II}^\omega(P) \uplus \mathtt{Occ}(M, P, x)$.

- *Case* (**HKRep**) *is treated like case* (**HKIn**κ).

- *Case* (**HKOut**). *We have* $P = \overline{a}\langle v\rangle.P_1$. *There exists* T' *such that* $\Gamma(a) = \sharp^{M_1, M_2}\ T'$, *and using rule* (**HKOut**) *we derive* $\Gamma \vdash^+ v : T'$, $M_2 >_{\mathrm{mul}} \mathtt{Lvl}(T')$ *and* $\Gamma \vdash^+ P_1 : N$. *The induction hypothesis gives* $\Gamma \vdash^+ P_1\{w/x\} : N$, $\Gamma \vdash^+ v\{w/x\} : T'$, $\mathbf{M}_I^\omega(P_1) = \mathbf{M}_I^\omega(P_1) \uplus \mathtt{Occ}(M, P_1, x)$ *and* $\mathbf{M}_{II}^\omega(P_1) = \mathbf{M}_{II}^\omega(P_1) \uplus \mathtt{Occ}(M, P_1, x)$. *As* $(\overline{a}\langle v\rangle.P_1)\{w/x\} = \overline{a}\langle v\{w/x\}\rangle.(P_1\{w/x\})$, *we can derive* $\Gamma \vdash^+ (\overline{a}\langle v\{w/x\}\rangle.P_1)\{w/x\} : N$ *Finally, we use Definitions 4.2.13 and 4.2.5 to conclude.*

- *Case* (**HKFun**). *We have* $v = (y \mapsto P_1)$. *There exists* T' *s.t.* $\Gamma(y) = T'$ $T_0 = T' \to^{\mathrm{succ}(M)} \diamond$ *and* $\Gamma \vdash^+ P_1 : M$. *The induction hypothesis gives* $\Gamma, y : T' \vdash^+ P_1\{w/x\} : M$. *We can derive* $\Gamma \vdash^+ y \mapsto P_1\{w/x\} : T_0$.

- *Case* (**HKVar**) *with* $v = y$. *We have* $\Gamma \vdash^+ y : T_0$ *with* $\Gamma(y) = T_0$. *We distinguish two cases:*
  - $x \neq y$. *Then* $y\{w/x\} = y$, *and the result follows.*
  - $x = y$. *Then* $T = T_0$, *and we can derive* $\Gamma \vdash^+ x\{w/x\} : T_0$.

□

This time, the measure domination lemma (Lemma 4.2.17) ensures that the measure of a process is smaller than its type. This will be used inside the proof of the Subject Reduction lemma, to bound the weight of a process being instantiated as a result of a communication.

Lemma 4.2.17 (Measure domination)
If $\Gamma \vdash^+ Q : N$, *then* $\mathbf{M}_{II}^\omega(Q) \leq_{\mathrm{mul}} N$.

Proof.
By induction on the typing derivation.

- *Cases* (**HKNil**), (**HKRes**), (**HKPar**), (**HKIf**), (**HKOut**), (**HKRep**κ) *are treated easily, using the induction hypotheses when needed, Definition 4.2.13, as well as some simple properties of multisets to do the calculations.*

- *Case* (**HKApp**). *We have* $P = v_1\lfloor v_2\rfloor$. *There exists* T_2 *such that* $\Gamma \vdash^+ v_1 : T_2 \to^N \diamond$ *and* $\Gamma \vdash^+ v_2 : T_2$. *Either* $v_1 = x$ *for some* x, *and, by Definition 4.2.13,* $\mathbf{M}_{II}^\omega(v_1\lfloor v_2\rfloor) = \emptyset$, *or* v_1 *is not a variable, and, by Definition 4.2.13,* $\mathbf{M}_{II}^\omega(v_1\lfloor v_2\rfloor) = N$*: we can conclude in both cases.*

- *Case* (**HKIn**κ). *We have* $P = a_1(x_1)^{v_1}...a_i(x_i)^{v_i}.a_{i+1}(x_{i+1})^{\mathtt{free}}..a_k(x_k)^{\mathtt{free}}.P_1$. *We derive* $\Gamma \vdash^+ P_1 : N$, $\Gamma \vdash^+ v_1 : T_1, \ldots, \Gamma \vdash^+ v_i : T_i$. *The induction hypothesis gives* $\mathbf{M}_{II}^\omega(P_1) \leq_{\mathrm{mul}} N$. *As Definition 4.2.13 gives* $\mathbf{M}_{II}^\omega(P) = N$, *we can conclude.*

□

Again, the whole termination proof relies on Subject Reduction. Our definitions for measures ensure that, for each reduction, either a decreasing for the first measure takes place, or the first measure stays the same but a decreasing appears for the second one. The first measure takes into account the prefixes already consumed inside an input sequence, whereas the second measure ignores them. As a consequence, either a reduction makes a progress inside an input sequence, and the first measure stays the same (one output ,say $\overline{a}\langle v\rangle$ is consumed but a new label `ok` appears in the input sequence, annotating an input prefix $a(x)$) but the second measure decreases (the output is consumed) or a reduction fires a whole input sequence, and the typing rules for replicated input sequences ensure that the measure of the released process is smaller than the measure of the input sequence which guarded it in the original process.

Lemma 4.2.18 (Subject reduction)
If $\Gamma \vdash^+ Q : N$ *and* $Q \to^\kappa Q'$ *then there exists* N' *s.t.* $\Gamma \vdash^+ Q' : N'$ *and*

- *either* $\mathbf{M}^\omega_I(Q) >_{\text{mul}} \mathbf{M}^\omega_I(Q')$,
- *or* $\mathbf{M}^\omega_I(Q) = \mathbf{M}^\omega_I(Q')$ *and* $\mathbf{M}^\omega_{II}(Q) >_{\text{mul}} \mathbf{M}^\omega_{II}(Q)$.

Proof. *We reason by induction on the derivation of* $Q \to^\kappa Q'$.

- *Cases* (**HKCong**), (**HKSpect**), (**HKScop**), (**HKCondT**) *and* (**HKCondF**) *are treated using the induction hypothesis, Fact 4.2.15 and and Definition 4.2.13.*
- *Case* (**HKBeta**). *We have* $Q = \mathbf{E}[(x \mapsto Q_1)\lfloor v_2\rfloor]$, $Q \to^\kappa \mathbf{E}[Q_1\{v_2/x\}]$. *Lemma 4.2.12 gives* $\Gamma \vdash^+ (x \mapsto Q_1)\lfloor v_2\rfloor : N$. *We derive* $\Gamma \vdash^+ (x \mapsto Q_1) : T \to^{\text{succ}(N_1)} \diamond$, $\Gamma \vdash^+ (x \mapsto Q_1)\lfloor v_2\rfloor : N$ *with* $\Gamma \vdash^+ Q_1 : N_1$, $\Gamma(x) = T$ *and* $\Gamma \vdash^+ v_2 : T$. *From Definition 4.2.13 and Fact 4.2.14, we deduce* $\mathbf{M}^\omega_I(Q) = \mathbf{M}^\omega_I(\mathbf{E}) \uplus \text{succ}(N_1)$. *We apply Lemma 4.2.16, yielding* $\Gamma \vdash^+ Q_1\{v_2/x\} : N_1$ *and conclude* $\Gamma \vdash^+ Q' : N_{(1)}$ *from Lemma 4.2.12. From Lemma 4.2.17, we get* $\mathbf{M}^\omega_{II}(Q_1\{v_2/x\}) <_{\text{mul}} N_1$. *As* Q_1 *appears in a message position in* Q, *because of the well-formedeness condition, every input prefix in* Q_1 *and* v_2 *is annotated with* `free`. *This allows us to deduce* $\mathbf{M}^\omega_I(Q_1\{v_2/x\}) <_{\text{mul}} N_1$. *As Fact 4.2.14 yields* $\mathbf{M}^\omega_{II}(Q') = \mathbf{M}^\omega_{II}(\mathbf{E}) \uplus \mathbf{M}^\omega_{II}(Q_1\{v_2/x\})$, *we conclude by* $\mathbf{M}^\omega_I(Q) >_{\text{mul}} \mathbf{M}^\omega_I(Q')$.
- *Case* (**HKPrUnr**). *We have*

$$Q = \mathbf{E}[\overline{a_i}\langle v_i\rangle.Q_1 \mid a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{\texttt{free}}.a_{i+1}(x_{i+1})^{\texttt{free}}...a_k(x_k)^{\texttt{free}}.Q_2]$$

and

$$Q' = \mathbf{E}[Q_1 \mid a_1(x_1)^{v_1}...a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{v_i}.a_{i+1}(x_{i+1})^{\texttt{free}}...a_k(x_k)^{\texttt{free}}.Q_2].$$

From Lemma 4.2.12, we derive $\Gamma \vdash^+ \overline{a_i}\langle v_i\rangle.Q_1 : N_1$, $\Gamma \vdash^+ Q : N$ *and*

$$\Gamma \vdash^+ a_1(x_1)^{v_1}..a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{\texttt{free}}.a_{i+1}(x_{i+1})^{\texttt{free}}..a_k(x_k)^{\texttt{free}}.Q_2 : N_2$$

with $\Gamma \vdash^+ v_i : T_i$, $\Gamma \vdash^+ Q_1 : N'_1$, $N_1 = N'_1 \uplus M^i_1$, *for all* $1 \le j \le k$, $\Gamma(a_j) = \sharp^{M^j_1, M^j_2} T_j$, $\Gamma(x_j) = T_j$, $M^j_2 >_{\text{mul}} \texttt{Lvl}(T_j)$, $\Gamma \vdash^+ v_1 : T_1, \ldots, \Gamma \vdash^+ v_{i-1} : T_{i-1}$, *and* $\Gamma \vdash^+ Q_2 : N_2$. *We can derive* $\Gamma \vdash^+ a_1(x_1)^{v_1}..a_{i-1}(x_{i-1})^{v_{i-1}}.a_i(x_i)^{v_i}.a_{i+1}(x_{i+1})^{\texttt{free}}...a_k(x_k)^{\texttt{free}}.Q_2 : N'_1$ *and, with Lemma 4.2.12,* $\Gamma \vdash^+ Q' : N'$ *with* $N' = N'_1 \uplus N_2$. *This is possible as the side conditions still hold, and* $\Gamma \vdash^+ v_i : T_i$. *By Definition 4.2.13 and Fact 4.2.14, we have* $\mathbf{M}^\omega_I(Q) = \mathbf{M}^\omega_I(\mathbf{E}) \uplus (\mathbf{M}^\omega_I(Q_1) \uplus M^i_1) \uplus (\mathbf{M}^\omega_I(Q_2) \uplus M^1_1 \uplus \cdots \uplus M^{i-1}_1)$ *and* $\mathbf{M}^\omega_I(Q') = \mathbf{M}^\omega_I(\mathbf{E}) \uplus (\mathbf{M}^\omega_I(Q_1)) \uplus (\mathbf{M}^\omega_I(Q_2) \uplus M^1_1 \uplus \cdots \uplus M^{i-1}_1 \uplus M^i_1)$ *which is* $\mathbf{M}^\omega_I(Q) = \mathbf{M}^\omega_I(Q')$. *Using the construction rules for the second measure, we deduce:* $\mathbf{M}^\omega_{II}(Q) = \mathbf{M}^\omega_{II}(\mathbf{E}) \uplus (\mathbf{M}^\omega_{II}(Q_1) \uplus M^i_1) \uplus (\mathbf{M}^\omega_{II}(Q_2))$ *and* $\mathbf{M}^\omega_{II}(Q') = \mathbf{M}^\omega_{II}(\mathbf{E}) \uplus (\mathbf{M}^\omega_{II}(Q_1)) \uplus (\mathbf{M}^\omega_I(Q_2))$ *which implies* $\mathbf{M}^\omega_{II}(Q) >_{\text{mul}} \mathbf{M}^\omega_{II}(Q')$.
- *A similar reasoning is used to treat cases* (**HKPrRep**) *and* (**HKTrig**).

- *Case* (**HKEndUnr**). *We have* $Q = \mathbf{E}[\overline{a_k}\langle v_k\rangle.Q_1 \mid a_1(x_1)^{v_1}...a_{k-1}(x_{k-1})^{v_{k-1}}.a_k^{\mathtt{free}}(x_k).Q_2]$ *and* $Q' = \mathbf{E}[Q_1 \mid Q_2\{v_1/x_1\}\dots\{v_{k-1}/x_{k-1}\}\{v_k/x_k\}]$. *Using Lemma 4.2.12, we derive* $\Gamma \vdash^+ \overline{a_k}\langle v_k\rangle.Q_1 : N_1$, $\Gamma \vdash^+ a_1(x_1)^{v_1}...a_k(x_k)^{\mathtt{free}}.Q_2 : N_2$ *and* $\Gamma \vdash^+ Q : N$ *where* $\Gamma \vdash^+ Q_1 : N_1'$, $\Gamma \vdash^+ v_1 : T_1$, $\dots$, $\Gamma \vdash^+ v_k : T_k$, $N_1 = N_1' \uplus M_1^i$, $\forall 1 \leq j \leq k, \Gamma(a_j) = \sharp^{M_1^j, M_2^j} T_j$ *and* $\Gamma(x_j) = T_j$, $M_2^j >_{\mathrm{mul}} \mathtt{Lvl}(T_j)$ *(notice that* $M_2^k >_{\mathrm{mul}} \mathtt{Lvl}(T_k)$ *is given by* $\Gamma \vdash^+ \overline{a_k}\langle v_k\rangle.Q_1 : N_1$*) and* $\Gamma \vdash^+ Q_2 : N_2$. *We use* k *times Lemma 4.2.16 to obtain* $\Gamma \vdash^+ Q_2\{v_1/x_1\}\dots\{v_k/x_k\} : N_2$ *and* $\mathbf{M}_I^\omega(Q_2\{v_1/x_1\}\dots\{v_k/x_k\}) = \mathbf{M}_I^\omega(Q_2) \uplus \mathtt{Occ}(Lvl_\Gamma(v_1), Q_2, x_1) \uplus \cdots \uplus \mathtt{Occ}(Lvl_\Gamma(v_k), Q_2, x_k)$. *We then construct* $\Gamma \vdash^+ Q_1 \mid Q_2\{v_1/x_1\}\dots\{v_k/x_k\} : N'$ *with* $N' = N_1' \uplus N_2$ *and conclude* $\Gamma \vdash^+ Q' : N_{(1)}$ *using Lemma 4.2.12. Using Definition 4.2.13 and Fact 4.2.14, we have* $\mathbf{M}_I^\omega(Q) = \mathbf{M}_I^\omega(\mathbf{E}) \uplus \mathbf{M}_I^\omega(Q_1) \uplus \mathbf{M}_I^\omega(Q_2) \uplus M_1^1 \uplus \cdots \uplus M_1^k$ *and* $\mathbf{M}_I^\omega(Q') = \mathbf{M}_I^\omega(\mathbf{E}) \uplus \mathbf{M}_I^\omega(Q_1) \uplus \mathbf{M}_I^\omega(Q_2) \uplus \mathtt{Occ}(Lvl_\Gamma(v_1), Q_2, x_1) \uplus \cdots \uplus \mathtt{Occ}(Lvl_\Gamma(v_k), Q_2, x_k)$. *The side conditions in the usage of rule* (**HKIn**κ) *give for all* $j < k$, $M_2^j >_{\mathrm{mul}} \mathtt{Lvl}(T_j) = \mathtt{Lvl}(v_j)$. *Similarly, the usage of rule* (**HKOut**κ) *gives* $M_2^k >_{\mathrm{mul}} \mathtt{Lvl}(T_k) = \mathtt{Lvl}(v_k)$ *and the rule* (**HKIn**κ) *gives* $\biguplus_{1\leq j\leq k} M_1^j >_{\mathrm{mul}} \biguplus_{1\leq j\leq k} \mathtt{Occ}(M_2^j, Q_2, x_j)$. *Thus, finally, we obtain* $\biguplus_{1\leq j\leq k} M_1^j >_{\mathrm{mul}} \biguplus_{1\leq j\leq k} \mathtt{Occ}(Lvl_\Gamma(v_j), Q_2, x_j)$ *and, using Fact 4.2.14,* $\mathbf{M}_I^\omega(Q') <_{\mathrm{mul}} \mathbf{M}_I^\omega(Q)$.
- *Case* (**HKEndRep**) *is treated similarly.*

□

Soundness of the type system is easily deduced from the decreasing of the two measures.

Theorem 4.2.19 (Soundness)
If $\Gamma \vdash^{\omega,\kappa} P : N$, *then* P *terminates.*

Proof. *Suppose, towards a contradiction, that process* P *diverges. Then, by Lemma 4.2.9, so does* $\mathtt{free}(P)$. *We thus have an infinite sequence* $(Q_i)_{i\geq 0}$ *such that* $Q_0 = \mathtt{free}(P)$ *and* $\forall i, Q_i \to^\kappa Q_{i+1}$. *Fact 4.2.11 gives* $\Gamma \vdash^+ Q_0 : N$. *We can apply Lemma 4.2.18 to each* Q_i *to obtain an infinite sequence* $\Gamma \vdash^+ Q_i : N_i$ *and an infinite sequence* $(\mathbf{M}_I^\omega(Q_i), \mathbf{M}_{II}^\omega(Q_i))$ *such that* $\forall i, \mathbf{M}_I^\omega(Q_i) >_{\mathrm{mul}} \mathbf{M}_I^\omega(Q_{i+1})$ *or* $\mathbf{M}_I^\omega(Q_i) = \mathbf{M}_I^\omega(Q_{i+1})$ *and* $\mathbf{M}_{II}^\omega(Q_i) >_{\mathrm{mul}} \mathbf{M}_{II}^\omega(Q_{i+1})$. *This contradicts the well-foundedness of* $>_{\mathrm{mul}}$. □

This completes the proof of the richer higher-order type system we presented here. We recall its features: controlling replications and application of functions, taking into account input sequences and treating differently capacity and weight for names. This allows us to reach a greater expressive power, as shown in the next section.

One can discuss the cost, in terms of technicalities, of such improvements. As in the message-passing case, dealing with input sequences implies the use of an auxiliary calculus, as the standard measure does no longer decrease over time. One has to introduce a calculus where partially consumed input sequences record, thanks to annotations, which outputs have already been consumed. As a consequence, a simulation lemma and a typing stability lemma are needed to justify the soundness of such a method. One has to be able to transform a typed process into an annotated typed process that diverges if the original process diverges. On the other side, introducing a distinction between weight and capacity does not require additional results, but only makes the types more complicated.

An expressive example: encoding separate choice

To illustrate the expressiveness of our type system for $\mathrm{HOpi}^!_\omega$, we present the encoding of the separate choice operator. Separate choice here means that operator + is applied only to inputs, or only to outputs.

The protocol in $\mathrm{HOpi}^!_\omega$ is presented in Figure 4.6; we make use of notation $\prod_{i=1}^n P_i$ to represent $P_1|\dots|P_n$, and we write $[\![P]\!]$ for the encoding of process P. The protocol is designed to let sums of output processes (the emitters) synchronise with sums of input processes (the receivers), whenever matching actions can be found. It works as follows. Any output action of an emitter may proceed. Whenever a matching input action exists, a mechanism of locks is used to ensure that at most one branch has been chosen on the emitter's side, and the same on the receiver's side. If this is not the case, the protocol backtracks, and the initial output action that has started executing is cancelled. Two channels, s and r, are used to implement two locks,

Outputs:

$$\begin{array}{l}\left[\sum_{i=1}^{n} \overline{x_i}\langle d_i\rangle.P_i\right] \;=\\ (\boldsymbol{\nu} s)\,\big(\, s(f_v).(f_v\lfloor \texttt{true}\rfloor \mid\, !s(f_v).f_v\lfloor \texttt{false}\rfloor)\\ \mid\, \big(\boldsymbol{\nu} a\big) \prod_{i=1}^{n} \overline{x_i}\langle d_i, x \mapsto \overline{s}\langle x\rangle, y \mapsto \overline{a}\langle y\rangle\rangle.a(x').\texttt{if } x' \texttt{ then } [P_i] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else } \mathbf{0}\,\big)\end{array}$$

Inputs:

$$\begin{array}{l}\left[\sum_{i=1}^{m} y_i(z).Q_i\right] \;=\\ (\boldsymbol{\nu} r)\,\big(\, \overline{r}\langle \texttt{true}\rangle\\ \quad \mid\, \prod_{i=1}^{m} (\boldsymbol{\nu} g)\,\big(\, \overline{g}\langle\star\rangle\\ \qquad \mid\, !g(t).y_i(z, f_s, f_a).r(x).\texttt{if } x\\ \qquad\quad \texttt{then } (\boldsymbol{\nu} u)\,\big(\, f_s\lfloor x \mapsto \overline{u}\langle x\rangle\rfloor \mid\, u(y).\texttt{if } y\\ \qquad\qquad\qquad \texttt{then } \overline{r}\langle \texttt{false}\rangle \mid f_a\lfloor \texttt{true}\rfloor \mid [Q_i]\\ \qquad\qquad\qquad \texttt{else } \overline{r}\langle \texttt{true}\rangle \mid f_a\lfloor \texttt{false}\rfloor \mid \overline{g}\langle\star\rangle\;\big)\\ \qquad\quad \texttt{else } \overline{r}\langle \texttt{false}\rangle \mid \overline{y_i}\langle z, f_s, f_a\rangle\,\big)\big)\end{array}$$

Figure 4.6: Separate choice in $\text{HOpi}^{!}_{\omega}$

that are tested on the receiver's side to decide whether the corresponding branch in the sum of inputs is allowed to proceed. When this is not the case, the protocol backtracks. The reader is referred to [Nes00] for a more detailed description of the protocol: ours closely follows the steps of Nestmann's original proposal. Because of backtracking, and of the inherent complexity of the processes being manipulated, the analysis of the protocol in terms of termination is non trivial. Also, when rewritten in the higher-order paradigm (more precisely, in $\text{HOpi}^{!}_{\omega}$), the protocol makes use of some patterns or combinations of operators that are delicate for termination (in particular, a pattern similar to $a(X).b(Y).X$). The proof that the original protocol does not add divergence is given in [Nes00], while [DS06] uses a type system to derive the same result.

Several details have to be changed or adapted w.r.t. the protocol in [Nes00] to program separate choice in our setting. For instance, in [Nes00], there may be a sequence of requests on channel s, the first of which receives answer `true`, and answer `false` is given to all following requests. In our protocol, this behaviour has been "hardwired" in the definition of the emitters, to clarify the encoding.

Names a and r are given the simple type $\sharp\,\mathtt{B}$ and g is given the simple type $\sharp\,\mathbb{1}$, like in the original process. Instead of sending channel a, which we cannot do since our calculus does not feature name-passing, we send a function $f_a : \mathtt{B} \to \diamond$ which allows the process that encodes a sum of inputs to output boolean values on a. The case of s is more complex, because the *input capability* on s is transmitted in the protocol of [Nes00]: the process encoding the sum of inputs performs an input on s after receiving s. The protocol of Figure 4.6 exploits an encoding of the π-calculus into $\text{HOpi}^{!}_{\omega}$, more precisely, of the *localised* π-calculus [SW01, Section 5.6]. Accordingly, a function of higher-order is transmitted in place of s in our encoding: upon reception of this function $f_s : \mathtt{B} \to \diamond \to \diamond$, it is applied to a function $f_u : \mathtt{B} \to \diamond$ (which intuitively represents the input capability), and this finally allows the process which sent f_s (the process encoding a sum of outputs) to transmit boolean values on channel $u : \sharp\,\mathtt{B}$ to the process encoding a sum of inputs. The latter protocol, in which functions are transmitted and applied, illustrates the higher-order nature of $\text{HOpi}^{!}_{\omega}$.

Typing the processes. As stated above, the protocol does not add divergence. We rely on the type system of Section 4.2 to show that our encoding of this protocol does not add non-typability; that is, if the processes Q_i and P_i are typable, the whole process is typable. Indeed, typing the processes given in Table 4.6 is possible provided the continuation processes P_i, Q_i can be typed. When this is the case, we must use, to type our protocol, levels that are strictly greater than those used to type the P_i, Q_i, which is always possible.

In what follows, we ignore this point, and assume the context $\emptyset$ is sufficient to type the P_i, Q_i with the global weight $\emptyset$. Adapting the typing to a situation where P_i, Q_i have non-$\emptyset$ weights is not conceptually difficult. For the same reasons, we ignore the level of the values d_i sent by the emitters. Instead, we just assume they have a well-formed type T of level $\emptyset$. It is easy to adapt the typing to a situation where the level of T is a given multiset M.

Proposition 4.2.20 (Typing the Encoding of Separate Choice)
Consider two sets of processes $(P_i)_{i=1,..,n}$ and $(Q_i)_{i=1,..,m}$ such that $\emptyset \vdash^+ P_i : \emptyset$, and $z : T \vdash^+ Q_i : \emptyset$ for some name z and type T, then $\left[\sum_{i=1}^n \overline{x_i}\langle d_i\rangle.P_i\right] \mid \left[\sum_{i=1}^m y_i(z).Q_i\right]$ does not exhibit a divergence.

Proof. *We establish this by applying Theorem 4.2.19. For this, we construct a typing derivation in which we assign types to the names used in Figure 4.6. The type assignment is as follows (we introduce $T_0 = \mathtt{B}^{\emptyset} \to^{\{0,0\}} \diamond$):*

$$\begin{array}{ll}
\Gamma(t) = \mathbb{1}^{\emptyset} & \Gamma(g) = \sharp^{\{4\},\{0\}}\ \mathbb{1}^{\emptyset} \\
\Gamma(x_i) = \Gamma(y_i) = \sharp^{\{4\},\{3\}}\ T, T_0 \to^{\{2,0\}} \diamond, T_0 & \Gamma(s) = \sharp^{\{2\},\{1\}}\ T_0 \\
\Gamma(r) = \sharp^{\{0\},\{0\}}\ \mathtt{B}^{\emptyset} & \Gamma(a) = \sharp^{\{0\},\{0\}}\ \mathtt{B}^{\emptyset} \\
\Gamma(u) = \sharp^{\{0\},\{0\}}\ \mathtt{B}^{\emptyset} & \Gamma(f_a) = \mathtt{B}^{\emptyset} \to^{\{0,0\}} \diamond \\
\Gamma(f_u) = \Gamma(f_v) = \mathtt{B}^{\emptyset} \to^{\{0,0\}} \diamond & \Gamma(f_s) = T_0 \to^{\{2,0\}} \diamond \\
\Gamma(z) = \Gamma(d_i) = T & \Gamma(x) = \Gamma(y) = \mathtt{B}^{\emptyset}
\end{array}$$

We do not give explicitly the construction of the whole derivation, but explain why every subprocess is typable. We introduce some notations that we use in the following calculations: we write $\mathsf{W}(a)$ for the weight of a (we overload notations and write $\mathsf{W}(P)$ for the weight of a process P), $\mathsf{C}(a)$ for the capacity of a, and $\mathsf{L}(v)$ for the level of v.

- $s(f_v).(f_v\lfloor\mathtt{true}\rfloor \mid !s(f_v).f_v\lfloor\mathtt{false}\rfloor)$. *To use rule* (**HKIn**) *here, we need to check that $\mathsf{C}(s) = \{1\}$ is greater than $\mathsf{L}(f_v) = \{0,0\}$, and also that $\mathsf{W}(s) = \{3\}$ is greater than*

  $$\mathtt{Occ}(\{2\}, (f_v\lfloor\mathtt{true}\rfloor \mid !s(f_v).f_v\lfloor\mathtt{false}\rfloor), f_v) = \{2\}$$

  (Remember that, by Definition 4.2.5, $\mathtt{Occ}(M, !a(y).P, x) = \emptyset$).

- $!s(f_v).f_v\lfloor\mathtt{false}\rfloor$. *To use rule* (**HKRep**) *here, we need to check that $\mathsf{C}(s) = \{1\}$ is greater than $\mathsf{L}(f_v) = \{0,0\}$, and that $\mathsf{W}(s) = \{3\}$ is greater than $o(\{2\}, f_v\lfloor\mathtt{false}\rfloor, f_v) \uplus \mathsf{W}(f_v\lfloor\mathtt{false}\rfloor) = \{2,2\}$.*

- $\overline{x_i}\langle d_i, x \mapsto \overline{s}\langle x\rangle, y \mapsto \overline{a}\langle y\rangle\rangle$; *this output is well-typed as $\mathsf{W}(s) = 2$, hence $\mathsf{L}(x \mapsto \overline{s}\langle x\rangle) = \{2,0\}$, which is smaller than $\{3\} = \mathsf{C}(x_i)$.*

  A similar reasoning holds for $y \mapsto \overline{a}\langle y\rangle$, which has level $\{0,0\}$. Moreover, $\mathsf{L}(d_i) = \emptyset$ is smaller than $\{3\} = \mathsf{C}(x_i)$.

- *The inputs $a(x'), u(y)$ are typed easily, because the types we assume for a and u impose level $\emptyset$ for the boolean variables x, y.*

- $f_s\lfloor x \mapsto \overline{u}\langle x\rangle\rfloor$. *The application is well-typed, because $\mathsf{W}(u) = \{0\}$, which gives a type compatible with the type we have assumed for f_s.*

- *The crux of this proof is the type-checking of the replicated subterm $!g(t).y_i(z, f_s, f_a).r(x).C$ (C is the continuation process, which can be deduced from the definition of the process in Figure 4.6).*

  In order to apply rule (**HKRep**), *we have to check that the two domination conditions hold. The condition $\mathtt{Lvl}(T_i) <_{\mathrm{mul}} M_2^i$ is fulfilled because:*

  - *$\mathsf{C}(g) = \{0\}$ is greater than $\mathsf{L}(y) = \emptyset$,*
  - *$\mathsf{C}(y_i) = \{3\}$ is greater than $\mathsf{L}(z) = \emptyset$, $\mathsf{L}(f_s) = \{2,0\}$ and $\mathsf{L}(f_a) = \{0,0\}$,*

– $\mathsf{C}(r) = \{0\}$ *is greater than* $\mathsf{L}(x) = \emptyset$.

As far as the other condition is concerned, we have to compute $\mathsf{W}(C)$ *and the contributions* $\mathtt{Occ}(\{3\}, C, f_s)$, $\mathtt{Occ}(\{3\}, C, f_a)$, $\mathtt{Occ}(\{3\}, C, z)$ *and* $\mathtt{Occ}(\{0\}, C, x)$. *Because of rule* (**HKIf**), *and by Definition 4.2.5, we have to compute these values for each branch in the nested conditional tests (we call these* C_j, *for* $j = 1, 2, 3$*), and compute the maximum.*

– *We have* $\mathsf{W}(C_1) = \mathsf{L}(f_s) \uplus \mathsf{L}(f_a) \uplus \mathsf{W}(r) = \{2, 0, 0, 0\}$ *(remember* Q_i *has weight* $\emptyset$*). If we suppose that* z *appears* c_1 *times in* Q_i*, not in object position nor inside the continuation of a replication, we have* $\mathtt{Occ}(\{3\}, z, C_1) = c_1.\{3\}$, $\mathtt{Occ}(\{3\}, C_1, f_s) = \{3\}$, $\mathtt{Occ}(\{3\}, C_1, f_a) = \{3\}$, $\mathtt{Occ}(\{0\}, C_1, x) = \{0\}$.
– *We have* $\mathsf{W}(C_2) = \mathsf{L}(f_s) \uplus \mathsf{L}(f_a) \uplus \mathsf{W}(r) \uplus \mathsf{W}(g) = \{4, 2, 0, 0, 0\}$. *We have* $\mathtt{Occ}(\{3\}, z, C_2) = \emptyset$, $\mathtt{Occ}(\{3\}, C_2, f_s) = \{3\}$, $\mathtt{Occ}(\{3\}, C_2, f_a) = \{3\}$, $\mathtt{Occ}(\{0\}, C_2, x) = \{0\}$.
– *We have* $\mathsf{W}(C_3) = \mathsf{W}(r) \uplus \mathsf{W}(y_i) = \{4, 0\}$. *We have* $\mathtt{Occ}(\{3\}, z, C_3) = \emptyset$, $\mathtt{Occ}(\{3\}, C_3, f_s) = \emptyset$, $\mathtt{Occ}(\{3\}, C_3, f_a) = \emptyset$, $\mathtt{Occ}(\{0\}, C_3, x) = \{0\}$.

Rule (**HKIf**) *allows us to compute* $\mathsf{W}(C) = \{4, 2, 0, 0, 0\}$. *Definition 4.2.5 gives* $\mathtt{Occ}(\emptyset, C, t) = \emptyset$, $\mathtt{Occ}(\{3\}, C, f_s) = \{3\}$, $\mathtt{Occ}(\{3\}, C, f_a) = \{3\}$, $\mathtt{Occ}(\{3\}, C, z) = c_1.\{3\}$ *and* $\mathtt{Occ}(\{0\}, C, x) = \{0\}$. *We can thus apply rule* (**HKRep**) *as* $\{4, 4, 0\}$, *which is the multiset sum of the weights of* g, y_i, r, *is strictly greater than* $\{4, 3, 3, 2, 0, 0, 0, 0\} \uplus c_1.\{3\}$, *the multiset sum of the global weight of* C *and the contribution of the capacities of* g, y_i, r *in* C.

□

The last item above illustrates the usefulness of the treatment of sequences of input prefixes: indeed we need to apply rule (**HKRep**) in a non-trivial way (more precisely, by treating together names g and y_i) in order to type-check this part of the process.

4.3 In PaPi

Types for termination in PaPi

Remember that in PaPi (Section 2), divergences arise both from recursion in usages of the passivation and process-passing mechanisms, and from recursive calls in the continuation of replicated (name-passing) inputs. We control the latter source of divergences by resorting to the type discipline of Section 3.1, while the former is controlled by associating levels to locations and to process-carrying channels, along the lines of the type systems we have studied in Sections 4.1 and 4.2.

However, the mere superposition of these two systems does not ensure termination, as the two mechanisms (process-passing and name-passing) can cooperate to produce divergences. This can be illustrated by the following process:

$$S_5 \quad = \quad l(X) \triangleright !a(y).X \quad | \quad l(\!|\overline{a}\langle p \rangle|\!) \quad | \quad \overline{a}\langle p \rangle \quad .$$

This process is divergent, but, unfortunately, the usages of passivation (which can be treated as a form of process-passing) and name-passing in S_5 are compliant with the principles of the aforementioned type systems. In this particular case, we must take into account the fact that X can be instantiated by a process containing an output on a channel having the same level as a. More generally, we must understand how the two type systems can interact, in order to avoid diverging behaviours.

The following grammars give the syntaxes for types for processes, locations, channels and messages (values):

$$T_P = m \qquad T_L = \mathbf{loc}^m \qquad T_C = \sharp^m T_V \qquad T_V = T_L \mid T_C \mid \diamond$$

$$(\textbf{PaNil})\frac{}{\Gamma \vdash_{\text{PaPi}} \mathbf{0} : 0} \qquad (\textbf{PaVar})\frac{\Gamma(X) = n}{\Gamma \vdash_{\text{PaPi}} X : n} \qquad (\textbf{PaRes})\frac{\Gamma(p) = T_V \Gamma \vdash_{\text{PaPi}} P : n}{\Gamma \vdash_{\text{PaPi}} (\boldsymbol{\nu} p)\ P : n}$$

$$(\textbf{PaPar})\frac{\Gamma \vdash_{\text{PaPi}} P_1 : n_1 \qquad \Gamma \vdash_{\text{PaPi}} P_2 : n_2}{\Gamma \vdash_{\text{PaPi}} P_1 \mid P_2 : \max(n_1, n_2)} \qquad (\textbf{PaPas})\frac{\Gamma(X) = k-1 \qquad \Gamma(p) = \mathbf{loc}^k \qquad \Gamma \vdash_{\text{PaPi}} P : n}{\Gamma \vdash_{\text{PaPi}} p(X) \triangleright P : n}$$

$$(\textbf{PaLoc})\frac{\Gamma(p) = \mathbf{loc}^k \qquad \Gamma \vdash_{\text{PaPi}} Q : n \qquad k > n}{\Gamma \vdash_{\text{PaPi}} p(\!|Q|\!) : k}$$

$$(\textbf{PaInP})\frac{\Gamma(X) = k-1 \qquad \Gamma(p) = Ch^k(\diamond) \qquad \Gamma \vdash_{\text{PaPi}} P : n}{\Gamma \vdash_{\text{PaPi}} p(X).P : n}$$

$$(\textbf{PaOutP})\frac{\Gamma \vdash_{\text{PaPi}} P : n \qquad \Gamma \vdash_{\text{PaPi}} Q : n' \qquad \Gamma(p) = Ch^k(\diamond) \qquad k > n'}{\Gamma \vdash_{\text{PaPi}} \overline{p}\langle Q \rangle.P : \max(k, n)}$$

$$(\textbf{PaInN})\frac{\Gamma(x) = T_V \qquad \Gamma(p) = Ch^k(T_V) \qquad \Gamma \vdash_{\text{PaPi}} P : n}{\Gamma \vdash_{\text{PaPi}} p(x).P : n}$$

$$(\textbf{PaOutN})\frac{\Gamma \vdash_{\text{PaPi}} P : n \qquad \Gamma(q) = T_V \qquad \Gamma(p) = Ch^k(T_V)}{\Gamma \vdash_{\text{PaPi}} \overline{p}\langle q \rangle.P : \max(k, n)}$$

$$(\textbf{PaRep})\frac{\Gamma(x) = T_V \qquad \Gamma(p) = Ch^k(T_V) \qquad \Gamma \vdash_{\text{PaPi}} P : n \qquad k > n}{\Gamma \vdash_{\text{PaPi}} !p(x).P : 0}$$

Figure 4.7: Typing rules for termination in PaPi

In PaPi, every entity (process, location, name-passing channel and process-passing channel) is given a level which is used to control the two sources of divergences discussed above. The level of a name-passing channel a corresponds to the maximum level allowed for the continuation P in a replicated input of the form $!a(x).P$. The level of a process-passing channel a corresponds to the maximum level of a process sent on a. Similarly, the level of a location l corresponds to the maximum level a process executing at l can have. In turn, the level of a process P corresponds to the maximum level of messages and locations that occur in P neither within a higher-order output nor under a replication. Typing rules are given on Figure 4.7.

As far as typing termination is concerned, we treat higher-order inputs (resp. outputs) like passivations (resp. located processes).

Remark 4.3.1 (Typing examples) *Process S_5 seen above cannot be typed. The typing rule for locations forces the level of location l to be strictly greater than $lvl(a)$ when typing $l(\!|\overline{a}\langle p \rangle|\!)$. The typing rule for passivation forces the level of l to be equal to $1 + lvl(X)$. Thus $lvl(X) \leq lvl(a)$ and the typing rule for replicated inputs cannot be applied to $!a(y).X$.*

For process $Coloc = l_1(X) \triangleright \big(l_2(Y) \triangleright (l_1(\!|X|Y|\!) \mid l_2(\!|\mathbf{0}|\!))\big)$ of section 2 to be typable, $lvl(l_1)$, the level assigned to l_1, should be greater than $lvl(l_2)$. In this case, we can observe that, thanks to typing, we know it is safe to take two processes running in separate locations and let them run in parallel, as Coloc does: while this might trigger new interactions (inter-locations communication is forbidden in PaPi), this is of no harm for termination.

Remark 4.3.2 (An extension of two type systems) *We can remark that the subset of the typing rules consisting of rules* (**PaNil**)*,* (**PaVar**)*,* (**PaRes**)*,*(**PaPar**)*,* (**PaInP**)*,*(**PaOutP**) *corresponds exactly to the*

type system for HOpi$_2$ introduced in Section 4.1. Hence, every HOpi$_2$ process that is typable according to the rules of Figure 4.7 is typable as a PaPi process using rules of Figure 4.7.

Moreover, the type system of Figure 4.7 subsumes the type system of Section 3.1 for the π-calculus: if a π-calculus process P is typable according to Section 3.1, then it is typable as a PaPi process.

Remark 4.3.3 (Possible extension) *It has to be noted that the type system we present can be made more expressive by exploiting ideas from Section 4.2. Indeed, we associate a* unique *level to names, and we could instead use three natural numbers to type a name: one would be its weight, and the other two would be interpreted as capacities, used to control the two sources of recursion: the weight of name passing outputs on one side, and the weight of process passing outputs and located processes on the other side. In what we have presented, these three components of the type of a name are merged into a single one. Additionally, sequences of inputs could be analysed according to the ideas of Section 4.2.*

Soundness of the Type System

The soundness proof for our type system essentially follows the same strategy as in the previous sections. Its core is the definition of a measure on processes, that takes into account the contribution of locations and first- and higher-order outputs that do not occur within a message or inside the continuation of a replication. The whole proof can be easily deduced from former proofs presented in this document. The peculiarities are that a same level system is used for both the process-passing and the message-passing parts, as said above, and that reductions can occur inside location. Thus, to ensure that the Subject Reduction property holds, one has to prove that the weight of a process inside a location does not grow after a reduction step. Thus the (**PaLoc**) rule can still be applied afterwards.

Fact 4.3.4 (Subject Congruence)
If $P \equiv P'$ then $\Gamma \vdash_{PaPi} P : n$ iff $\Gamma \vdash_{PaPi} P' : n$.

Proof. *Easily done by induction on the derivation of $P \equiv P'$, using the commutativity, the associativity and the neutrality of 0 for* max. □

We previously hinted at how the measure is defined. Outputs and locations contribute to the measure if they are inside a location, but not if they are inside a message or inside a continuation of a replicated input.

Definition 4.3.5 (Measure)
Given a PaPi process P, the measure $\mathbf{M}^{PaPi}(P)$ is defined as follows:

$$\mathbf{M}^{PaPi}(\mathbf{0}) = \mathbf{M}^{PaPi}(X) = \emptyset \qquad \mathbf{M}^{PaPi}(P_1 \mid P_2) = \mathbf{M}^{PaPi}(P_1) \uplus \mathbf{M}^{PaPi}(P_2)$$

$$\mathbf{M}^{PaPi}((\boldsymbol{\nu} p)\ P_1) = \mathbf{M}^{PaPi}(P_1) \qquad \mathbf{M}^{PaPi}(l(X) \triangleright P_1) = \mathbf{M}^{PaPi}(P_1)$$

$$\mathbf{M}^{PaPi}(l(\!|Q|\!)) = \mathbf{M}^{PaPi}(Q) \uplus \{n\} \text{ if } \Gamma(l) = \mathbf{loc}^n \qquad \mathbf{M}^{PaPi}(p(X).P_1) = \mathbf{M}^{PaPi}(P_1)$$

$$\mathbf{M}^{PaPi}(\overline{p}\langle Q\rangle.P_1) = \mathbf{M}^{PaPi}(P_1) \uplus \{k\} \text{ if } \Gamma(p) = \sharp^k \diamond \qquad \mathbf{M}^{PaPi}(\overline{p}\langle q\rangle.P_1) = \mathbf{M}^{PaPi}(P_1) \uplus \{k\} \text{ if } \Gamma(a) = \sharp^k T$$

$$\mathbf{M}^{PaPi}(p(x).P_1) = \mathbf{M}^{PaPi}(P_1) \qquad \mathbf{M}^{PaPi}(!p(x).P_1) = \emptyset$$

This measure is straightforwardly extended to evaluation contexts using $\mathbf{M}^{PaPi}([\,]) = \emptyset$.

As reduction in PaPi may involve two kinds of substitutions (one for name variables, the other for process variables) the usual Subject Substitution lemma is decomposed into two properties, which we prove below. Of course, one can notice that the subject substitution lemma for names is similar to the one proved for message-passing calculi and that the lemma for processes is similar to the one proved for process-passing calculi.

We prove the standard lemmas relating measure and typability to evaluation contexts and structural congruence. Notice that, in Lemma 4.3.6, the condition $n_0 \leq n'$ for P_0 is required when $\mathbf{E} = l(\!|\mathbf{E}_2|\!)$: indeed, we check that the process running in a location of level n cannot have a weight greater than n.

Lemma 4.3.6 (Context Typing)
If $\Gamma \vdash_{PaPi} \mathbf{E}[P] : n$ *then:*

1. $\Gamma \vdash_{PaPi} P : n'$ *for some* $n' \leq n$.
2. *For all* P_0 *s.t.* $\Gamma \vdash_{PaPi} P_0 : n_0$ *with* $n_0 \leq n'$, $\Gamma \vdash_{PaPi} \mathbf{E}[P_0] : n_{(0)}$ *with* $n_{(0)} \leq n$.

Proof.
The proof of this lemma is very similar to the one of Fact 4.1.1 and can be found in Appendix A.

□

Lemma 4.3.7 (Measure and context)
If $\Gamma \vdash_{PaPi} \mathbf{E}[P] : n$, *then* $\mathbf{M}^{PaPi}(\mathbf{E}[P]) = \mathbf{M}^{PaPi}(\mathbf{E}) \uplus \mathbf{M}^{PaPi}(P)$.

Proof. *By structural induction over* $\mathbf{E}$*:*

- *Case* $[\,]$. *Then* $\mathbf{M}^{PaPi}(\mathbf{E}[P]) = \mathbf{M}^{PaPi}(P) = \mathbf{M}^{PaPi}(P) \uplus \mathbf{M}^{PaPi}([\,])$.
- *Case* $(\nu a)\,\mathbf{E}_2$. *Then* $\mathbf{M}^{PaPi}(\mathbf{E}[P]) = \mathbf{M}^{PaPi}((\nu a)\,\mathbf{E}_2[P])$ *which is, by Definition 4.3.5* $\mathbf{M}^{PaPi}(\mathbf{E}_2[P])$. *We use the induction hypothesis, to get* $\mathbf{M}^{PaPi}(\mathbf{E}_2[P]) = \mathbf{M}^{PaPi}(\mathbf{E}_2) \uplus \mathbf{M}^{PaPi}(P)$. *As, by Definition 4.3.5,* $\mathbf{M}^{PaPi}(\mathbf{E}) = \mathbf{M}^{PaPi}(\mathbf{E}_2)$, *we conclude.*
- *Case* $\mathbf{E}_2 \mid P_1$. *Then* $\mathbf{M}^{PaPi}(\mathbf{E}[P]) = \mathbf{M}^{PaPi}(\mathbf{E}_2[P] \mid P_1)$ *which is, by Definition 4.3.5* $\mathbf{M}^{PaPi}(\mathbf{E}_2[P]) \uplus \mathbf{M}^{PaPi}(P_1)$. *We use the induction hypothesis, to get* $\mathbf{M}^{PaPi}(\mathbf{E}_2[P]) = \mathbf{M}^{PaPi}(\mathbf{E}_2) \uplus \mathbf{M}^{PaPi}(P)$. *As, by Definition 4.3.5,* $\mathbf{M}^{PaPi}(\mathbf{E}) = \mathbf{M}^{PaPi}(\mathbf{E}_2) \uplus \mathbf{M}^{PaPi}(P_1)$, *we conclude.*
- *Case* $l(\!|\mathbf{E}_2|\!)$. *Then* $\mathbf{M}^{PaPi}(\mathbf{E}[P]) = \mathbf{M}^{PaPi}(\mathbf{E}_2[P]) \uplus \{n\}$ *if* $\Gamma(l) = \mathbf{loc}^n$. *We use the induction hypothesis to get* $\mathbf{M}^{PaPi}(\mathbf{E}_2[P]) = \mathbf{M}^{PaPi}(\mathbf{E}_2) \uplus \mathbf{M}^{PaPi}(P)$. *As, by Definition 4.3.5,* $\mathbf{M}^{PaPi}(\mathbf{E}) = \mathbf{M}^{PaPi}(\mathbf{E}_2) \uplus \{n\}$, *we conclude.*

□

Fact 4.3.8 (Measure and structural congruence)
If $P \equiv Q$ *and* $\Gamma \vdash_{PaPi} P : n$, *then* $\mathbf{M}^{PaPi}(P) = \mathbf{M}^{PaPi}(Q)$.

Proof. *We proceed by induction on the derivation of* $P \cong Q$, *using the associativity, commutativity and neutrality of* $\emptyset$ *for the operator* $\uplus$. □

Here is the first subject substitution result, for substitutions of names.

Lemma 4.3.9 (Subject Substitution - Names)
If $\Gamma \vdash_{PaPi} P : n$, $\Gamma(x) = T_V$ *and* $\Gamma(q) = T_V$, *then*

1. $\Gamma \vdash_{PaPi} P\{q/x\} : n'$ *for some* $n' \leq n$ *and,*
2. $\mathbf{M}^{PaPi}(P\{q/x\}]) = \mathbf{M}^{PaPi}(P)$.

Proof.
The proof is similar to the one of Lemma 3.1.3 and can be found in Section A.

□

As announced, we prove a similar result, about substitutions of processes. Here, the process being substituted can contribute to the measure, and the factor c appearing in the expression of $\mathbf{M}^{\mathrm{PaPi}}(P\{q/x\})$ is, again, the number of available occurrences (not inside a message or the continuation of a replicated input) of the process variable X in P.

Lemma 4.3.10 (Subject Substitution - Processes)
Suppose $\Gamma \vdash_{PaPi} P : n$, $\Gamma(X) = m$ *and* $\Gamma \vdash_{PaPi} Q : m'$ *with* $m' \leq m$. *Then there exists* c *s.t.:*

1. $\Gamma \vdash_{PaPi} P\{Q/X\} : n'$ *for some* $n' \leq n$ *and;*
2. $\mathbf{M}^{PaPi}(P\{Q/X\}) = \mathbf{M}^{PaPi}(P) + c.\mathbf{M}^{PaPi}(Q)$.

Proof.
The proof is similar to the one of Lemma 4.1.5 and can be found in Section A

□

We now establish the usual upper bound property about $\mathbf{M}^{\mathrm{PaPi}}(P)$.

Lemma 4.3.11
If $\Gamma \vdash_{PaPi} P : n$ *then* $\mathbf{M}^{PaPi}(P) <_{\mathrm{mul}} \{n+1\}$.

Proof. *By induction on the typing judgement,*

- *Cases* (**PaNil**), (**PaVar**), (**PaRes**), (**PaPar**), (**PaPas**), (**PaRep**), (**PaInN**) *and* (**PaInP**) *are easily treated, using the induction hypotheses when needed and Definition 4.3.5.*
- *Case* (**PaLoc**). *Suppose* $P = l(\!|Q_1|\!)$. *We derive* $\Gamma(l) = \mathbf{loc}^n$ *and* $\Gamma \vdash_{PaPi} Q_1 : n_1$ *for some* $n_1 < n$. *The induction hypothesis gives* $\mathbf{M}^{PaPi}(Q_1) <_{\mathrm{mul}} n_1 + 1$. *By definition,* $\mathbf{M}^{PaPi}(P) = \mathbf{M}^{PaPi}(Q_1) \uplus \{n\}$. *We have* $\{n+1\} >_{\mathrm{mul}} \mathbf{M}^{PaPi}(P)$ *as* $\mathbf{M}^{PaPi}(Q_1) <_{\mathrm{mul}} \{n_1+1\} <_{\mathrm{mul}} \{n+1\}$ *and* $\{n\} <_{\mathrm{mul}} \{n+1\}$.
- *Case* (**PaOutP**). *Suppose* $P = \overline{p}\langle Q_2\rangle.P_1$. *There exists* k *s.t.* $\Gamma(p) = \sharp^k \diamond$ *and using rule* (**PaOutP**), *we derive* $\Gamma \vdash_{PaPi} Q_2 : n_2$ *and* $\Gamma \vdash_{PaPi} P_1 : n_1$, *for some* n_1, n_2 *s.t.* $n_2 < k$, *and* $n = \max(k, n_1)$. *By definition,* $\mathbf{M}^{PaPi}(P) = \mathbf{M}^{PaPi}(P_1) \uplus \{k\}$. *The induction hypothesis gives* $\{n_1+1\} >_{\mathrm{mul}} \mathbf{M}^{PaPi}(P_1)$. *Thus, as we have* $\{\max(k,n_1)+1\} >_{\mathrm{mul}} \{k\}$ *, we deduce* $\{\max(k,n_1)+1\} >_{\mathrm{mul}} \mathbf{M}^{PaPi}(P)$.
- *Case* (**PaOutN**). *Suppose* $P = \overline{p}\langle q\rangle.P_1$. *There exists* k, T_V *s.t.* $\Gamma(p) = \sharp^k T_V$ *and, using rule* (**PaOutN**), *we derive* $\Gamma \vdash_{PaPi} P_1 : n_1$ *for some* n_1 *s.t.* $n = \max(k, n_1)$. *By definition,* $\mathbf{M}^{PaPi}(P) = \mathbf{M}^{PaPi}(P_1) \uplus \{k\}$. *The induction hypothesis gives* $\{n_1+1\} >_{\mathrm{mul}} \mathbf{M}^{PaPi}(P_1)$. *Thus, as we have* $\{\max(k,n_1)+1\} >_{\mathrm{mul}} \{k\}$, *we get* $\{\max(k,n_1)+1\} >_{\mathrm{mul}} \mathbf{M}^{PaPi}(P)$.

□

Finally, we establish the main property of our type system, that relates typability, reduction and the measure.

Lemma 4.3.12 (Subject Reduction)
If $\Gamma \vdash_{PaPi} P : n$ *and* $P \to P'$, *then there exists* n' *s.t.* $n \geq n'$, $\Gamma \vdash_{PaPi} P' : n'$ *and* $\mathbf{M}^{PaPi}(P) > \mathbf{M}^{PaPi}(P')$.

Proof. *We reason by induction on the derivation of* $P \to P'$*:*

- *Case* (**PaCong**)*is treated easily using the induction hypothesis, Fact 4.3.4 and Fact 4.3.8, Definition 4.3.5, the compatibility of the multiset ordering with* $\uplus$, *and the compatibility of* $\leq$ *with* max.
- *Case* (**PaComP**). *We have in this case* $P = \mathbf{E}[\overline{p}\langle Q_1\rangle.P_3 \mid p(X).P_2]$ *and* $P' = \mathbf{E}[P_3 \mid P_2\{Q_1/X\}]$. *From Lemma 4.3.6, we get* $\Gamma \vdash_{PaPi} \overline{p}\langle Q_1\rangle.P_3 : \max(k, n_3)$ *and* $\Gamma \vdash_{PaPi} p(X).P_2 : n_2$ *for some* k, n_2, n_3 *s.t.* $\Gamma(p) = \sharp^k \diamond$, $\Gamma \vdash_{PaPi} P_1 : n_3$, $\Gamma \vdash_{PaPi} Q_1 : n_1$ *for some* $n_1 < k$, $\Gamma \vdash_{PaPi} P_2 : n_2$, $\Gamma(X) = k-1$ *with* $n = \max(k, n_3, n_2)$. *As* $k - 1 \geq n_1$, *we derive, using Lemma 4.3.10,* $\Gamma \vdash_{PaPi} P_2\{Q_1/X\} : n_2'$ *for some* $n_2' \leq n_2$ *and* $\mathbf{M}^{PaPi}(P_2\{Q_1/X\}) = \mathbf{M}^{PaPi}(P_2) + c.\mathbf{M}^{PaPi}(Q_1)$ *for some* c. *We then derive* $\Gamma \vdash_{PaPi} P_3 \mid P_2\{Q_1/X\} : \max(n_3, n_2')$. *Clearly* $\max(n_3, n_2') \leq \max(k, n_3, n_2)$. *This allows us to use Lemma 4.3.6 to conclude* $\Gamma \vdash_{PaPi} P' : n'$ *with* $n' \leq n$. *By Definition 4.3.5* $\mathbf{M}^{PaPi}(P) = \{k\} \uplus \mathbf{M}^{PaPi}(P_2) \uplus \mathbf{M}^{PaPi}(P_3)$ *and* $\mathbf{M}^{PaPi}(P') = \mathbf{M}^{PaPi}(P_2\{Q/X\}) \uplus \mathbf{M}^{PaPi}(P_3)$. *From Lemma 4.3.11, we know that* $\mathbf{M}^{PaPi}(Q_1) <_{\mathrm{mul}} \{n_1+1\}$. *As* $n_1 < k$, *we get* $c.\mathbf{M}^{PaPi}(Q_1) <_{\mathrm{mul}} \{k\}$. *This allows us to conclude* $\mathbf{M}^{PaPi}(P) <_{\mathrm{mul}} \mathbf{M}^{PaPi}(P')$.

- *The case* (**PaPass**)*, with* $P = \mathbf{E}[l(\!|Q_1|\!) \mid l(X) \triangleright P_2]$*and* $\Gamma(l) = \mathbf{loc}^k$ *for some* k*, is treated like case* (**PaComP**) *with* $P = \mathbf{E}[\overline{a}\langle Q_1\rangle.\mathbf{0} \mid a(X).P_2]$ *for some* a *s.t.* $\Gamma(a) = \sharp^k \diamond$*.*

- (**PaTrig**)*. We have* $P = \mathbf{E}[\overline{p}\langle q\rangle.P_1 \mid !a(x).P_2]$ *and* $P' = \mathbf{E}[P_1 \mid P_2\{q/x\} \mid !p(x).P_2]$*. We derive* $\Gamma \vdash_{PaPi} \overline{p}\langle q\rangle.P_1 : \max(k, n_1), \Gamma \vdash_{PaPi} !p(x).P_2 : 0$ *and* $\Gamma \vdash_{PaPi} P : n$ *for some* k*,*n_1 *s.t.* $\Gamma(p)\sharp^k T, \Gamma(q) = T$*,* $\Gamma \vdash_{PaPi} P_1 : n_1, \Gamma \vdash_{PaPi} P_2 : n_2, \Gamma(x) = T$ *for some* $n_2 < k$ *and* $n = \max(k, n_1)$*. Applying Lemma 4.3.9, allows us to construct* $\Gamma \vdash_{PaPi} P_2\{q/x\} : n_2'$ *with* $n_2' \leq n_2$ *and* $\mathbf{M}^{PaPi}(P_2\{q/x\}) = \mathbf{M}^{PaPi}(P_2)$*. We then derive* $\Gamma \vdash_{PaPi} P_1 \mid P_2\{q/x\} \mid !p(x).P_2 : \max(k, n_1, n_2)$ *As* $k > n_2$*, we get* $\max(n_1, n_2') < \max(k, n_1, n_2)$*. This allows us to use Lemma 4.3.6 to conclude* $\Gamma \vdash_{PaPi} P' : n'$*. By Definition 4.3.5,* $\mathbf{M}^{PaPi}(P) = \{k\} \uplus \mathbf{M}^{PaPi}(P_1)$ *and* $\mathbf{M}^{PaPi}(P') = \mathbf{M}^{PaPi}(P_2\{q/x\}) \uplus \mathbf{M}^{PaPi}(P_1)$*. As* $\Gamma \vdash_{PaPi} P_2 : n_2$*, we can use Lemma 4.3.11 to deduce* $\mathbf{M}^{PaPi}(P_2\{Q_1/X\}) <_{\text{mul}} \{n_2 + 1\}$*. As* $k \geq (n_2 + 1)$*, this allows us to conclude* $\mathbf{M}^{PaPi}(P) <_{\text{mul}} \mathbf{M}^{PaPi}(P')$*.*

- *The proof for case* (**PaComN**) *is deduced from the proof for case* (**PaTrig**)*.*

□

We easily derive soundness from Subject Reduction, using the same method than in the previous sections.

Theorem 4.3.13 (Soundness)
If $\Gamma \vdash_{PaPi} P : n$*, then* P *terminates.*

Proof.
We suppose by contradiction that P *diverges, which means that we have an infinite sequence* $(P_i)_{0 \leq i}$ *s.t.* $P_0 = P$ *and for each* i*,* $P_i \to P_{i+1}$*.*

By applying Lemma 4.3.12 to each P_i*, we obtain an infinite sequence* $(\Gamma \vdash_{PaPi} P_i : n_i)_i$ *s.t.* $\mathbf{M}^{PaPi}(P_{i+1}) <_{\text{mul}} \mathbf{M}^{PaPi}(P_i)$*. We obtain a contradiction with Theorem 2.1.3.* □

It would be actually easy to add to this type system the features we presented in section 4.2: taking care of input sequences, dividing levels of names and location into weight and capacity. The fact that message-passing and process-passing features are both present here does not hinder the inclusion of such refinements to our type system. Of course, the price to pay would be much longer proofs, as we would have to introduce and auxiliary annotated calculus. The ideas behind these refinements being not new in this thesis, we choose not to present them here. Yet, the expressive power of such a type system would allow us to ensure termination for a large set of programs.

Chapter 5

Inference for the weight-based type systems

In this section, we discuss the hardness of the inference problem for the type systems presented in the previous sections. We aim at building type systems whose inferences procedure is efficient, i.e. can be done in polynomial time. The time consumed by the inference procedure is counted in elementary operations and is related to the size of a process (which can be defined as either the number of prefixes inside the process or the number of symbols used, as both define the same complexity classes). To ease the readability of complexity statements, we will call $\mathbf{S}^{\uplus}$ (respectively $\mathbf{S}^{\mathrm{ord}}$) the type system developed in Section 3.2.1 (respectively 3.3).

5.1 The problem of inference for our type systems

The inference problem is, for a given process P, finding, when there exists one, a typing context Γ allowing us to type P. Formally,

Definition 5.1.1 (Inference problem)
The inference problem for a type system is, given a process P, to decide whether there exist Γ and T_P s.t. $\Gamma \vdash P : T_P$.

Remark 5.1.2 (Inference and type systems *à la Church*) *In the previous sections, we give an* à la Church *presentation of the type systems we developed. We suppose that typing contexts Γ are oracles mapping names (being bound or free) to types.* A la Church *presentation implies that each typing judgement is implicitly related to a unique typing derivation. Thus, in this setting, the inference problem can be considered as deciding if, for a given untyped process P, there exists a typing judgement $\Gamma \vdash P : T_p$ for some Γ and T_P.*

The inference problem is crucial for when it comes to implement type systems as concrete verification routines, as automated verification procedure have to take as little time as possible. Should the inference take a time exponential in the size of the process, no doubt the whole procedure will be considered inefficient in practise. As a consequence, one should aim to build type systems whose inference procedures are polynomial in the size of the process, when possible.

We give a *à la Church* presentation of these type systems, implying that a typing judgement is given a unique corresponding typing derivation, *a priori*. Therefore, if Γ and P are given, checking whether there exists T_P s.t. $\Gamma \vdash P : T_P$ is direct (and can be performed in a time polynomial in the size of the process).

Moreover, the inference procedure for the system of simple types takes polynomial time (see Proposition 5.2.1). We will explain later how this task can be performed efficiently.

As a consequence, when considering the type systems for termination we presented above, the crux of the inference problem is the assignment of levels (and partial order informations, in the case of Section 3.3)

to the types names. One can remark that, as levels are only used for comparisons between natural numbers (or multisets of natural numbers), we never need, in order to type a process, a number of levels greater than the number of names. In the case of the system of Section 3.3, the number of different partial orders that can be assigned to a single name is finite. The direct consequence of these observations is that the inference problem is always decidable for the systems we study. Indeed, if a process P contains K different names, one could try to satisfy the constraints given by the typing rules with every typing context Γ, and we can count at most K^K of them, based on what we write above, obtained by assigning to each name a natural number between 1 and K.

5.2 Hardness of inference for systems of Section 3 and 4

In this section, we study the complexity of the simple type system presented in Section 2. Every type system for termination we present above uses the simple types syntax as a basis, by decorating the simple types with annotations (levels or partial orders). Thus, the inference problem for simple types is crucial, as assigning simple types to names is a prerequisite to the assignment of more complex types.

We can notice that assigning simple types is not a difficult task. We already wrote in Section 6.1 that we have to assign simple types to names in a "top-down way" . That is, two different names necessarily have the same type if they are both carried by the same name, or if they are carried by names which have to have the same type. As a consequence, the inference algorithm we use starts with the names p which are never carried on an another channel (if there does not exist such a name, then the process P is not simply typable), and affects to the same types all names appearing in messages whose subjects have same type. Then, we perform this task recursively.

Proposition 5.2.1 (Inference for simple types)
Type inference for simple types presented in Section 2 is polynomial.

The standard type inference procedure for simple types can be found in [VH93]. The authors use an algorithm to generate constraints in time and space linear in the textual representation of the process. We give a presentation using graph, as this framework will be used in the next complexity proofs of this section.
Proof.

Consider a process P (which abides the Barendregt convention we state in Section 2). We have to explain how we get an assignment of simple types to the names of P. We use the following algorithm, building a directed graph $\mathcal{G}$ whose vertexes are sets of names (either bound or free) of P (thus edges are written (S, S')):

1. *Add a vertex $\{a\}$ in $\mathcal{G}$ for every name a present in P (this is done linearly in the size of P, by examining once each prefix and restrictions of P).*
2. *For each prefix $a(b)$, $!a(b)$ or $\overline{a}\langle b\rangle$, add an edge $(\{a\},\{b\})$ in $\mathcal{G}$ (this is done linearly in the size of P, by examination once each prefix inside P).*
3. *Repeat the following operation:*
   (a) *Check the presence of cycles in $\mathcal{G}$ (this can be done in time linear in the number of edges of $\mathcal{G}$): if there exists one, then the process is not simply-typable,*
   (b)
   - *For each vertex S which has no father in the graph (there exists at least one such S, as the graph has no cycle), perform the following (we call this procedure "treating S"):*
     i. *If S has at least one child, collapse all the children $S_1, \ldots, S_k$ of S into one single node, that is, remove the nodes $S_1, \ldots, S_k$ of $\mathcal{G}$ and create a new node $S_1 \cup S_2 \cup \cdots \cup S_k$. There is an edge $(S_1 \cup S_2 \cup \cdots \cup S_k, S')$ if and only if there was an edge (S_i, S') for some i.*
     ii. *Treat $S_1 \cup S_2 \cup \cdots \cup S_k$.*

Stop repeating 3 *when every node in the graph has at most one child.*

4. *For each node,starting with the nodes which have no child and treating their father afterwards, perform the following operations. There exists at least one node with no child, as the graph is without cycles.*

   - *Give type $\sharp(\mathbb{1})$ to this node if it has no child.*
   - *Give type $\sharp(T)$ to this node if its child has been given type T.*

This algorithm terminates, as every collapsing operation decreases the number of nodes in the graph by at least one. Moreover, is it easy to see that the algorithm is polynomial in the size of the process (the numbers of vertexes and edges in $\mathcal{G}$ are clearly linear in the number of prefixes in the process, and the operations performed are polynomial in the size of the graph).

The soundness of this algorithm is ensured by proving these statements:

- *If the algorithm is successful, then, for every prefix $\overline{a}\langle v\rangle$ in P, v is given some type T and a the type $\sharp(T)$ (the same results holds for inputs $a(x)$ and replicated inputs $!a(x)$). Suppose $\overline{a}\langle v\rangle$ appears in P, then in the step 2, an edge is created between the vertex corresponding to a and the one corresponding to v. During step 3 this edge is maintained as an edge between a vertex corresponding to a set of names containing a and a vertex corresponding to a set of names containing v. In step 4, names v is given some type T and name a the type $\sharp(T)$.*

- *If the algorithm fails, then, we have a cycle of names $a_1, \ldots, a_n$ such that for each i, a_{i+1} appears in message position in a prefix whose subject is a_i, and a_1 appears in message position in a prefix whose subject is a_n. Indeed, a failure of the algorithm can only appears in step 3.(a). It means that there exists a cycle in the graph. By definition of graph $\mathcal{G}$ in step 1 and 2, we get the latter cyclic property. As a consequence, the process is not simply-typable.*

- *If the process is not simply-typable, we derive from the typing rules an unsatisfiable set of constraints of the form $\Gamma(a) = \sharp(T) \wedge \Gamma(b) = T$. If this set of constraint is unsatisfiable, it means that it we can deduce two constraints of the form $\Gamma(a) = T$ and $\Gamma(a) = \sharp^{T'}$ with T appearing in T'. This implies the existence of a cycle in the graph, containing the vertex corresponding to name a. As a consequence, the inference algorithm fails.*

□

Remark 5.2.2 (Polyadicity) *The result presented above also holds for polyadic simple types. The proof is a bit different, as one has to remember in which position among the arguments of a channel a name is carried. Indeed, if a name a is given simple type $\sharp(\widetilde{T})$, it means that a name carried on its i-th argument shall have type T_i. Our technique can be easily adapted by considering $\mathcal{G}$ as a labelled multigraph (a directed graph where there are (possibly) several annotated edges between two vertexes) and annotating edges in $\mathcal{G}$ with numbers, recording in which argument a name is carried. For the prefixes written above, the set of edges of the resulting multigraph is $\{(\{a\}, 1, \{x, v\}), (\{a\}, 2, \{y, w\}\}$ where (S, n, S') stands for an edge annotated with n from S to S'. As a consequence, the collapsing operation unifies to a same vertex the children linked to their father by edges with the same annotation, instead of unifying all the children together. The condition for stopping the loop is now "Every vertex has at most one child linked with an edge annotated with a given number".*

We propose here a first result stating that the inference of the system of Section 3.1 can be perform in an efficient way. As we wrote before, the complexity of the whole inference procedure boils down to the time spent on finding a suitable level assignment. In this original type system for termination, the analysis is simple: the typing rule for replications generates constraints of the form $k > n$ where k is the level of the subject of the replicated input and n is the weight given by the continuation. We explain in Section 3.1 that the weight n of the continuation is exactly the maximum level of an available output $\overline{b}\langle v\rangle$ (see Definition 3.1.6) it contains. Therefore, if $\overline{b}\langle v\rangle$ is an available output in the continuation of $!a(x).P$, then l_a the level of a has to be strictly greater than l_b, the level of b, and the constraint $l_a > l_b$ has to be

satisfied. By repeating this procedure we generate a large set of constraints between integers, the existence of a solution for this set of constraints being equivalent to the typability of the whole process. Notice that the number of constraints generated in such a way is bound by the size of the continuation.

Proposition 5.2.3 (Inference for the system of Section 3.1)
Type inference for the system presented in Section 3.1 is polynomial.

Proof.
By running the algorithm presented in Proposition 5.2.1, we build a graph identifying names that have necessarily the same type. To find a suitable level assignment, we proceed in the following way: we associate to each vertex S (remember vertexes represent sets of names which have necessarily the same types) in $\mathcal{G}$ a level variable *l_S. Based on the typing rules, we can generate, in a time polynomial in the size of the process, a set $\mathcal{C}$ of constraints consisting of inequalities (of the form $l_S < l_{S'}$) on level variables, such that $\mathcal{C}$ is satisfiable if and only if P is typable, and the size of $\mathcal{C}$ is linear in the size of P by construction. A constraint $l_S < l_{S'}$ appears in $\mathcal{C}$ if there is an available output of a name in S inside the continuation of a replicated input whose subject is a name in S'. The satisfiability of $\mathcal{C}$ is equivalent to the acyclicity of the graph induced from $\mathcal{C}$, which can again be checked in polynomial time. Thus, the type inference problem for this system is polynomial.* □

System of Section 4.1 We can successfully adapt this proof to derive a similar result for the type system of Section 4.1, the corresponding set of constraints being deduced by the application of the rule (**HOut**) instead of the rule (**Rep**), in the case of Proposition 5.2.3. One can notice that the level assignments are very similar in these two settings. The whole inference procedure is actually simpler for the process-passing system as every name has type $\sharp^k \diamond$. As a consequence, one does not have to compute simple types for names as a prerequisite, and we can start the level assignment operation with a graph $\mathcal{G}$ where every node is annotated with a single name.

Proposition 5.2.4 (Type inference for HOpi$_2$)
Type inference for types presented in Section 4.1 is polynomial.

System of Section 4.2 We can also adapt this method to prove that inference for the type system of Section 4.2 is polynomial. The procedure is a bit more technically involved, as we have to give types to functional values, but the main principles are the same. We construct a graph $\mathcal{G}$ where vertexes are either sets of nodes or sets of functional values (which, in turn, can be either functional variables x, y or (explicit) functions like $x \mapsto v$). We add an edge (S, S') to $\mathcal{G}$ when the considered process contains:

- either a prefix $\overline{a}\langle v \rangle$, $a \in S$, and $v \in S'$,
- or an application $v_1 \lfloor v_2 \rfloor$, $v_1 \in S$, and $v_2 \in S'$.
- or a functional value $x \mapsto P$, $(x \mapsto P) \in S$, and $x \in S'$.

By using an algorithm similar to the one used in the proof of Proposition 5.2.3, we get a simple type assignment for functional values and channels.

We finish the inference procedure by assigning a level variable to all vertexes in $\mathcal{G}$, that is to all sets of names and all sets of functional values. Then, we use the typing rules (**HoOut**) and (**HoAbs**) to generate a set of constraints $\mathcal{C}$. This part can be done easily in a time linear in the size of the process. We conclude by solving this set of constraints, as in the proof of Proposition 5.2.3.

Proposition 5.2.5 (Type inference for HOpi$_\omega$)
Type inference for types presented in Section 4.2 is polynomial.

5.3 Hardness of inference for the system with input sequences

One crucial result regarding inference for our systems is that the type system presented in Section 3.2.1 is hard to infer. Actually, we prove in this section that its inference is NP-complete, using a reduction from 3SAT.

Such a result is not a surprise, as one can notice the differences in the constraints generated by this system, accommodating input sequences, and the one of Section 3.1. On one side, in the previous case, the comparisons between levels given by the typing rules are of the "one-to-many" kind, i.e. the level of a unique name has to be greater than the weight of a process, which is, actually, the maximum level of all available output names found inside this process. A single exploration of the process separates these constraints into one-to-one constraints of the form "the level of name a has to be strictly greater than the level of name b". On the other side, when considering input sequences, the typing rules give many-to-many comparisons: we know that a multiset of levels (the ones of the names in the input sequence) has to be greater than another multiset of levels (the ones of the available output names in the continuation) but we cannot separate this comparison into several one-to-one comparisons. For instance, when trying to type $!a.b.(\overline{c} \mid \overline{d})$, we know that the multiset $\{\mathtt{Lvl}(a), \mathtt{Lvl}(b)\}$ has to be greater than $\{\mathtt{Lvl}(c), \mathtt{Lvl}(d)\}$, but we cannot know *a priori* which is greater between $\mathtt{Lvl}(a)$ and $\mathtt{Lvl}(b)$. Moreover, we are not able to know, for instance, if $\mathtt{Lvl}(a)$ has to be strictly greater than $\mathtt{Lvl}(c)$. This uncertainty imposes a choice, and the reduction from 3SAT proves that there is no known method far better than trying each level assignment.

Theorem 5.3.1 (Hardness of inference of $\mathbf{S}^{\uplus}$)
The type inference problem for $\mathbf{S}^{\uplus}$ is NP-complete.

Proof. *As in Proposition 5.2.3, a preliminary operation can be done, in polynomial time, to construct a graph $\mathcal{G}$ unifying the names which have necessarily the same type.*

Let z be the number of names occurring in P. The problem is in NP because checking if one of the z^z different ways of distributing names into z levels yields typability can be done in polynomial time with respect to the size of the process and the number of names.

We now show that we can reduce 3SAT to the problem of finding a mapping of levels. We consider an instance $\mathcal{I}$ of 3SAT: we have n clauses $(C_i)_{i\leq n}$ of three literals each, $C_i = l_i^1, l_i^2, l_i^3$. Literals are possibly negated propositional variables taken from a set $V = \{v_1, \ldots, v_m\}$. The problem is to find a mapping from V to $\{True, False\}$ such that, in each clause, at least one literal is set to True.

All names we use to build the processes below will be CCS names. We fix a name $\mathtt{T}$. To each variable $v_k \in V$, we associate two names x_k and x'_k, and define the process

$$P_k = !\mathtt{T}.\mathtt{T}.\overline{x_k}.\overline{x'_k} \mid !x_k.x'_k.\mathtt{T}$$

We then consider a clause $C_i = \{l_i^1, l_i^2, l_i^3\}$ from $\mathcal{I}$. For $j \in \{1,2,3\}$ we let $n_i^j = x_k$ if l_i^j is v_k, and $n_i^j = x'_k$ if l_i^j is $\neg v_k$. We then define the process

$$Q_i \stackrel{\text{def}}{=} !n_i^1.n_i^2.n_i^3.\overline{\mathtt{T}}.$$

We call $\mathcal{I}'$ the problem of finding a typing derivation in $\mathbf{S}^{\uplus}$ for the process $P \stackrel{\text{def}}{=} P_1 \mid \ldots \mid P_m \mid Q_1 \mid \ldots \mid Q_n$. Note that the construction of P is polynomial in the size of $\mathcal{I}$.

We now analyse the constraints induced by the processes we have defined. The level associated to name $\mathtt{T}$ is noted t.

- *The constraint associated to $!\mathtt{T}.\mathtt{T}.\overline{x_k}.\overline{x'_k}$ is equivalent to*

$$\big(t \geq \mathtt{Lvl}(x_k) \wedge t \geq \mathtt{Lvl}(x'_k)\big) \ \wedge \ \big(t > \mathtt{Lvl}(x_k) \vee t > \mathtt{Lvl}(x'_k)\big) \ .$$

  The constraint associated to $!x_k.x'_k.\mathtt{T}$ is equivalent to

$$t \leq \mathtt{Lvl}(x_k) \ \vee \ t \leq \mathtt{Lvl}(x'_k) \ .$$

Hence, the constraint determined by P_k is equivalent to

$$\big(\mathtt{Lvl}(x_k) = t \wedge \mathtt{Lvl}(x'_k) < t\big) \quad \vee \quad \big(\mathtt{Lvl}(x'_k) = t \wedge \mathtt{Lvl}(x_k) < t\big) \ . \tag{5.1}$$

- *The constraint associated to $!n_{i_1}.n_{i_2}.n_{i_3}.\overline{\mathtt{T}}$ is equivalent to*

$$t \leq \mathtt{Lvl}(n_i^1) \ \vee \ t \leq \mathtt{Lvl}(n_i^2) \ \vee \ t \leq \mathtt{Lvl}(n_i^3) \ . \tag{5.2}$$

We now prove that the statement '$\mathcal{I}'$ has a solution' is equivalent to '$\mathcal{I}$ has a solution'. First, if $\mathcal{I}$ has a solution $S : V \to \{True, False\}$ then fix $t = 2$, and set $\mathtt{Lvl}(x_k) = 2, \mathtt{Lvl}(x'_k) = 1$ if v_k is set to True, and $\mathtt{Lvl}(x_k) = 1, \mathtt{Lvl}(x'_k) = 2$ otherwise. We check easily that condition (5.1) is satisfied; condition (5.2) also holds because S is a solution of $\mathcal{I}$. Conversely, if $\mathcal{I}'$ has a solution, then we deduce a boolean mapping for the literals in the original 3SAT problem. Since constraint (5.1) is satisfied, we can set v_k to True if $\mathtt{Lvl}(x_k) = t$, and False otherwise. We thus have that v_k is set to True iff $\mathtt{Lvl}(x_k) = t$, iff $\mathtt{Lvl}(x'_k) < t$. Hence, because constraint (5.2) is satisfied, we have that in each clause C_i, at least one of the literals is set to True, which shows that we have a solution to $\mathcal{I}$. □

This proof can be easily adapted to establish the same result for the system of section 3.3: the idea is to 'disable' the use of the partial order, e.g. by adopting a different type for $\mathtt{T}$. We thus get:

Proposition 5.3.2 (Hardness of type inference for $\mathbf{S}^{\mathrm{ord}}$) *The type inference problem for $\mathbf{S}^{\mathrm{ord}}$ is NP-complete.*

Proof. *As the type system of Section 3.3 contains the previous one, the reduction from 3SAT we propose in the proof of Theorem 5.3.1 can be adapted.* □

Investigating the cause of NP-difficulty. The crux in the proof of Theorem 5.3.1 is to use the input sequence component to introduce a form of choice: as written above, to type the process $!a.b.P$, we cannot know a priori, for c appearing in output position inside P, whether to set $\mathtt{Lvl}(a) \geq \mathtt{Lvl}(c)$ or $\mathtt{Lvl}(b) \geq \mathtt{Lvl}(c)$. Intuitively, we exploit this to encode the possibility for booleans to have two values, as well as the choice of the literal being set to True in a clause. By renouncing to the input sequences in the typing rules, we get the system of Section 3.1, which is polynomial.

However, it appears that NP-completeness is not related only to input sequences: indeed, it is possible to define a polynomial restriction of this system still accommodating input sequences. Let us call $\mathbf{S}^{\kappa}$ the type system obtained from $\mathbf{S}^{\uplus}$ by imposing *distinctness of levels*: two names can have the same level only if their types are unified when resolving the unification constraints. More precisely, the mapping from the set of names of the graph $\mathcal{G}$ to levels is injective. Note that this is more demanding: as in the previous settings, two names can have the same type without being represented by the same vertex: in the very simple process $\overline{p}\langle a\rangle \mid \overline{q}\langle a\rangle$, p and q have necessarily the same simple type $\sharp(T)$ (the simple type of a being T), but must be given different levels in $\mathbf{S}^{\kappa}$ because their types are not unified during inference (as they are not arguments of the same channel, or arguments of two channels which have necessarily the same type).

Although typing processes of the form $!a(x).b(y).c(z).P$ seems to introduce the same kind of choice as previously, it can be shown that type inference is polynomial in $\mathbf{S}^{\kappa}$. Intuitively, the reason for this is that there exists a level variable, say α, such that for every constraint $I >_{\mathrm{mul}} O$ in the set $\mathcal{C}$ determined by the process being typed, the number of occurrences of α in O is not greater than the number of occurrences of α in I. We call α a *root level variable*: it can be shown that if no such α exists, then the process is not typable.

This gives a strategy to compute a level assignment for names, and do so in polynomial time: set α to the maximum level, and consider a constraint $I >_{\mathrm{mul}} O$: if there are as many αs in I as in O, replace the constraint with the equivalent constraint where the αs are removed. Otherwise, the number of αs strictly decreases, which means we can simply get rid of this constraint. We thus obtain an equivalent, smaller problem, and we can iterate this reasoning (if there are no more constraints to satisfy, we pick a random assignment for the remaining levels).

System $\mathbf{S}^{\kappa}$ retains the lexicographical comparison and the input sequences from the system of Section 3.2.1, but is polynomial. We will propose further a new system, strictly more expressive than $\mathbf{S}^{\uplus}$ (see Proposition 5.4.3 below); since $\mathbf{S}^{\kappa}$ is a restriction of $\mathbf{S}^{\uplus}$, it is less expressive than this new system.

$$(\mathbf{NNil})\frac{}{\Gamma \vdash^+ \mathbf{0} : 0} \qquad (\mathbf{NPar})\frac{\Gamma \vdash^+ P_1 : m_1 \qquad \Gamma \vdash^+ P_2 : m_2}{\Gamma \vdash^+ P_1 \mid P_2 : m_1 + m_2}$$

$$(\mathbf{NRes})\frac{\Gamma \vdash^+ P : m \qquad \Gamma(a) = \sharp^k T}{\Gamma \vdash^+ (\nu a)\ P : m} \qquad (\mathbf{NIn})\frac{\Gamma \vdash^+ P : m \qquad \Gamma(a) = \sharp^k T \qquad \Gamma(x) = T}{\Gamma \vdash^+ a(x).P : m}$$

$$(\mathbf{NOut})\frac{\Gamma \vdash^+ P : m \qquad \Gamma(a) = \sharp^k T \qquad \Gamma(v) = T}{\Gamma \vdash^+ \overline{a}\langle v\rangle.P : m + k}$$

$$(\mathbf{NRep})\frac{\Gamma \vdash^+ P : m \qquad \forall i, \Gamma(a_i) = \sharp^{k_i} T_i \wedge \Gamma(x_i) = T_i \qquad m < \Sigma_{1 \leq i \leq n} k_i}{\Gamma \vdash^+ !a_1(x_1).\ldots.a_n(x_n).P : \emptyset}$$

Figure 5.1: Typing rules for $\mathbf{S}^+$

5.4 A polynomial system accommodating input sequences

We now study system $\mathbf{S}^+$, which is very similar to $\mathbf{S}^{\uplus}$, whose inference is studied in Theorem 5.3.1. However, one major difference is that we no longer compute the weight as a multiset of levels, but instead, as a natural number. This is already the case in Section 3.1; yet, in this section the weight of a process P stands for the algebraic sum of the levels of the available outputs in P. Using algebraic operations yields however better results, in terms of both expressiveness and complexity of inference. As usual, we give an *'a la Church* presentation of the type system.

Definition 5.4.1 (System $\mathbf{S}^+$)
System $\mathbf{S}^+$ is defined by rules of Figure 5.1

The + present in the rules (**NPar**) and (**NOut**) and the Σ present in (**NRep**) represent the standard addition over natural numbers which contrasts with the former use of multiset union $\uplus$.

Soundness of $\mathbf{S}^+$ can be established by adapting the proof for Section 3.2.1:

Proposition 5.4.2 (Soundness)
System $\mathbf{S}^+$ ensures termination.

Proof.
This can be done by checking that, everywhere in the proofs of Section 3.2.1, the multiset operators $\uplus, >_{\mathrm{mul}}$ can be replaced by $+, >$.

Definition 3.2.12 has to be changed accordingly: available resources are now computed as natural numbers and no longer as a multiset,; however, the way they are computed remains the same. As a consequence, Lemma 3.2.16 still holds. The crux of the soundness proof, Lemma 3.2.18 is proved as previously, we only need to replace, in each computation of **AvRes**(), *the multiset operators by their algebraic counterparts. The final result (corresponding to Proposition 3.2.19) holds, as the standard ordering on natural number is obviously well-founded.*

□

One can wonder why we did not start with this definition for weight (as a single natural number instead of a multiset of integers). The original system can be found in [DS06] and one can notice that it seems indeed natural to remember the contribution of each name in the weight, as multisets of names are compared in the typing rule for replication. The main reason for using multisets of names instead of plain algebraic operations is that the extensions we propose in Sections 3.3 and 3.4 cannot be easily adapted, if we do not use multisets. Indeed, the fact that two names of lower levels can collaborate to compensate a name of higher level, which

is the main reason $\mathbf{S}^+$ is more expressive than $\mathbf{S}^{\uplus}$, prevents us from including the use of a partial order in the type system without too much technicalities.

A interesting point is that $\mathbf{S}^+$ is at least as expressive as $\mathbf{S}^{\uplus}$, in the sense that every process typable using the latter is typable in the former. Proposition 5.4.3 states this property. The key of the proof is that, as the number of comparisons involved in the typing derivation is finite, we are able to give a level to each name such that the result of every comparison of multisets of levels, computed with the operators $\uplus$ and $>_{\text{mul}}$, is equivalent to the result of the same comparison, computed with the operators $+$ and $>$.

As an example, consider the comparisons of multisets of names, $(\{a\},\{b,b,c\})$ and $(\{b,b\},\{c,c,c,c,c\})$, given by typing the process $!a.(\overline{b} \mid \overline{b} \mid \overline{c}) \mid !b.b.(\overline{c}.\overline{c} \mid \overline{c}.\overline{c})$. These comparisons are satisfied, according to the previous system, by assigning levels $3,2,1$ to a,b,c (respectively). Indeed, we get $\{3\} >_{\text{mul}} \{2,2,1\}$ and $\{2,2\} >_{\text{mul}} \{1,1,1,1,1\}$. However, this level assignment no longer leads to typability if we use the rules of $\mathbf{S}^+$: we get $3 \not> 5$, $4 \not> 5$ and $5 \not> 5$. Yet, by "enlarging" the space between two consecutive levels, we get back typability. For instance if we use the level assignment $a \mapsto 3^3, b \mapsto 3^2, c \mapsto 3^1$, the comparisons become $3^3 = 27 > 3^2+3^2+3^1 = 19$ and $3^2+3^2 = 18 > 3^1+3^1+3^1+3^1+3^1 = 15$. Intuitively, there exists a number b such that the typing assignment giving level b^k in $\mathbf{S}^+$ to a name given level k in $\mathbf{S}^{\uplus}$ ensures typability.

An interesting issue is that $\mathbf{S}^+$ is actually strictly more expressive, as it allows several smaller names to cooperate in order to be compared with a single heavier name. The proof of the following lemma illustrates this property.

Proposition 5.4.3 (Expressiveness)
System $\mathbf{S}^+$ is strictly more expressive than $\mathbf{S}^{\uplus}$.

Proof.
We first show that $\mathbf{S}^+$ is at least as expressive as $\mathbf{S}^{\uplus}$. Consider a process P_0 and the associated typing judgement $\Gamma \vdash^{\kappa} P_0 : N_0$. Every replication (we suppose we have an enumeration $1 \leq r \leq R$ of the replications) in P_0 has the form $!a_1^r(x_1)\dots a_n^r(x_n).P^r$. We write I^r and O^r for the multisets of levels (with respect to Γ) associated respectively to the input sequence $a_1^r \dots a_n^r$ and we define P^r the weight of the continuation of this sequence. If M is a multiset, We write $M|_{(k)}$ for the number of occurrences of the element k in M. We define K as the maximum level assigned to a name by Γ.

As the number of replications in P is finite, there exists an integer b such that $(i) : \forall r, \forall j \in [1 \dots K]. |O^r|_{(j)} - I^r|_{(j)}| < b$, and we build Γ^+ typing context of system $\mathbf{S}^+$ for P_0 by assigning level $b^{\mathtt{Lvl}(a)}$ to name a, where $\mathtt{Lvl}(a)$ denotes the level of a according to the typing context of $\mathbf{S}^{\uplus}$ for P_0.

Let us show that this induces a correct typing for P_0 in $\mathbf{S}^+$ by proving that for each process P satisfying (i) (notice that if a process satisfies this property, then all of its subprocesses satisfy it), $\Gamma \vdash^{\kappa} P : N$ implies $\Gamma^+ \vdash^+ P : N'$, with $N' = \Sigma_k(N|_{(k)}.b^k)$. We proceed by induction on the typing judgement $\Gamma \vdash^{\kappa} P : N$.

The only interesting case is the rule (**KRep**)*, the other ones are easily dealt with, by doing computations using the operator $+$ instead of $\uplus$. Suppose we have $\Gamma \vdash^{\kappa} !a_1(x_1)\dots a_n(x_n).P_1 : \emptyset$. We denote by I the multiset of levels of $a_1,\dots,a_n$ according to Γ (formally written $\biguplus_i \mathtt{Lvl}(a_i)$). We derive $\Gamma \vdash^{\kappa} P_1 : N_1$ and $I >_{\text{mul}} N_1$. Using the induction hypothesis, we get $\Gamma^+ \vdash^+ P_1 : \Sigma_k(N_1|_{(k)}.b^k)$. According to the results about multisets stated in Section 2, this implies that there exists a level u such that $I|_{(K)} = N_1|_{(K)}, I|_{(K-1)} = N_1|_{(K-1)}, \dots, I|_{(u+1)} = N_1|_{(u+1)}$ and $I|_{(u)} > N_1|_{(u+1)}$. We compute the difference of weights between the two multisets of names according to $\mathbf{S}^+$: $\Sigma_{1\leq k\leq K}(I|_{(k)}.b^k) - \Sigma_k(N_1|_{(k)}) = \Sigma_{1\leq k\leq K}(I|_{(k)} - N_1|_{(k)})b^k \geq b^u + \Sigma_{1\leq j<u}(I|_{(k)} - N_1|_{(k)})b^k$. The latter quantity is strictly positive by definition of b, which shows that the rule* (**NRep**) *can be applied.*

Then we show that there are processes which can be typed by system $\mathbf{S}^+$ but not by $\mathbf{S}^{\uplus}$. Consider $Q_1 \stackrel{\text{def}}{=} !a.\overline{b} \mid !b.b.\overline{a}$. The process Q_1 is ill-typed according to $\mathbf{S}^{\uplus}$: the first subterm imposes $\mathtt{Lvl}(a) > \mathtt{Lvl}(b)$, and the multisets associated to the second subterm are of the form $\{\mathtt{Lvl}(b),\mathtt{Lvl}(b)\}$ and $\{\mathtt{Lvl}(a)\}$. As a consequence, there is no way to obtain $\{\mathtt{Lvl}(b),\mathtt{Lvl}(b)\} >_{\text{mul}} \{\mathtt{Lvl}(a)\}$. However, by setting $\mathtt{Lvl}(a) = 3$ and $\mathtt{Lvl}(b) = 2$, we can check that Q_1 is typable for $\mathbf{S}^+$, the replications yielding, respectively, the inequalities $3 > 2$ and $4 > 3$. □

The main feature of $\mathbf{S}^+$ is that the use of algebraic comparisons allows its inference to be polynomial.

Theorem 5.4.4 (Inference for $\mathbf{S}^+$)
Type inference for system $\mathbf{S}^+$ is polynomial.

Proof.
By inspecting the process to be typed, type inference amounts, as in the previous cases, to find a suitable level assignment; that is, in the particular case of $\mathbf{S}^+$, to find a solution to a system of inequalities of the form $\Sigma_j a_{i,j}.u_j > 0$, where the $a_{i,j}$s are (possibly negative) integers and the solution is the vector of the u_js, which are natural numbers.

This system has a solution if and only if the system consisting of the inequalities $\Sigma_j a_{i,j}.u_j \geq 1$ has one. We resort to linear programming in rationals to solve the latter problem (we can choose to minimise $\Sigma_j u_j$), which can be done in polynomial time.

Because of the shape of inequalities generated by the typing problem, there exists a rational number solution to the inequalities if and only if there exists an integer solution. More precisely, as the right member of every inequality we generate is 0, the validation of an inequality is stable by a multiplication of each u_j by the same number. Thus, if we have a solution, that is an assignment mapping each u_j to a rational number, we can multiply all these numbers by the product of all of them ($\Pi_j u_j$) and get an assignment of natural numbers, which is still a solution. □

5.5 System $\mathbf{S}^{\mathcal{R}}$: Definition and Properties

The type system $\mathbf{S}^{\text{ord}}$ is built on top of system $\mathbf{S}^{\uplus}$, and improves its expressiveness by allowing the use of partial orders. To define $\mathbf{S}^{\mathcal{R}}$, we restrict ourselves to the partial order component of $\mathbf{S}^{\text{ord}}$, and do not analyse sequences of input prefixes ($!a_1(x_1)\dots a_n(x_n)$) as in $\mathbf{S}^{\uplus}$, $\mathbf{S}^{\text{ord}}$ and $\mathbf{S}^+$: in a term of the form $!a(\widetilde{x}).P$, name a must dominate every available output in P, either because it is of higher level (as in the original system of Section 3.1), or via the partial order relation (in the case of the levels of the two names are the same). As a consequence, the measure based on the multisets of levels of available outputs can grow (and not only stay the same as in $\mathbf{S}^{\text{ord}}$) with a reduction step. Yet, in this case, as the measure decreases for the partial order, termination is ensured.

We now introduce $\mathbf{S}^{\mathcal{R}}$. Notations for partial orders and related operations are introduced in Section 3.3. We need a new safety condition, in this section, in order to prevent the restriction operators to create loops. Indeed, consider $!p(a,b).!a.(\nu c)(\overline{b} \mid \overline{p}\langle b,c\rangle)$, which is similar to the process used to justify the safety condition of $\mathbf{S}^{\text{ord}}$ (see Definition 3.3.3). This process diverges in presence of $\overline{p}\langle u,v\rangle \mid \overline{u}$. Yet it abides the conditions defined in the introduction above: we are able to let a dominates both p and b. Indeed, we could give p a type containing a level strictly smaller than the one of a and b, and a partial order stating that its first argument is strictly greater than the second one. The second replication would be typed as a is greater than p in level, and a dominates b for the partial order.

Definition 5.5.1 (Safety for $\mathbf{S}^{\mathcal{R}}$) *The condition $\mathbf{safe}^{\mathcal{R}}(k,P)$, given an integer k and a process P, and a typing context Γ (which will be clear from context every time we use this operator), holds if and only if one of the following holds:*

1. *either there is no restriction (νc) in P such that $\mathtt{Lvl}(c) = k$.*
2. *or there is no output $\overline{b}\langle v\rangle$ in P such that $\mathtt{Lvl}(c) = k$.*

As a consequence, if there is an output of level $\mathtt{Lvl}(a)$ in the continuation P, the replication is well-typed only if there is no restriction on level $\mathtt{Lvl}(a)$ (or greater) in P. Informally, our system allows creation of new names only when the decreasing only takes place on levels (and not on the partial order).

Typing judgements are similar to the ones for $\mathbf{S}^{\mathcal{R}}$. The typing rules for $\mathbf{S}^{\mathcal{R}}$ are given on Fig. 5.2.

In the rule (**SRRep**), the condition $\Gamma\mathcal{R} \vdash a :\succ^{\mathcal{R}} N$ holds if one of the following conditions holds:

1. $\forall v \in N, \mathtt{Lvl}(a) > \mathtt{Lvl}(v)$

$$(\mathbf{SRNil})\,\Gamma\mathcal{R} \vdash \mathtt{0} : \emptyset \qquad (\mathbf{SRPar})\,\frac{\Gamma\mathcal{R} \vdash P_1 : N_1 \qquad \Gamma\mathcal{R} \vdash P_2 : N_2}{\Gamma\mathcal{R} \vdash P_1|P_2 : N_1 \uplus N_2}$$

$$(\mathbf{SRIn})\,\frac{\Gamma(a) = \sharp^k_{\mathtt{R}}\Gamma(\widetilde{x}) \qquad \Gamma\mathcal{R}' \vdash P : N \qquad \mathcal{R} = \mathtt{R}\lfloor\widetilde{x}\rfloor \uplus \mathcal{R}}{\Gamma\mathcal{R} \vdash a(\widetilde{x}).P : N}$$

$$(\mathbf{SROut})\,\frac{\Gamma(a) = \sharp^k_{\mathtt{R}}\Gamma(\widetilde{v}) \qquad \Gamma\mathcal{R} \vdash P : N \qquad \mathtt{R}\lfloor\widetilde{v}\rfloor \subseteq \mathcal{R}}{\Gamma\mathcal{R} \vdash \overline{a}\langle\widetilde{v}\rangle.P : N \uplus \{a\}} \qquad (\mathbf{SRRes})\,\frac{\Gamma(c) = \sharp^n_{\mathtt{R}}\widetilde{T} \qquad \Gamma\mathcal{R} \vdash P : N}{\Gamma\mathcal{R} - \{c\} \vdash (\nu c)P : N}$$

$$(\mathbf{SRRep})\,\frac{\Gamma(a) = \sharp^k_{\mathtt{R}}\Gamma(\widetilde{x}) \qquad \Gamma\mathcal{R}' \vdash P : N \qquad \mathcal{R}' = \mathcal{R} \uplus \mathtt{R}\lfloor\widetilde{x}\rfloor \qquad \{a\} :\!\succ^{\mathcal{R}} N \qquad \mathbf{safe}^{\mathcal{R}}(k, P)}{\Gamma\mathcal{R} - \{\widetilde{x}\} \vdash\, !a(\widetilde{x}).P : \emptyset}$$

Figure 5.2: System $\mathbf{S}^{\mathcal{R}}$: Typing Rules

2. $N = N_1 \uplus \{b\}$,
   - $\forall v \in N_1, \mathtt{Lvl}(a) > \mathtt{Lvl}(v)$
   - $\mathtt{Lvl}(b) = \mathtt{Lvl}(a), a\mathcal{R}b$

We could have merged the two rules into the second one, but, for the sake of clarity, and to ease the proof, we prefer to distinguish the two cases: when the weight (defined like in $\mathbf{S}^{\mathrm{ord}}$) decreases and when it does not increase.

Notice that the partial order can be used for *at most one output* in the continuation process to typecheck a replication. Indeed, without these constraint, we could typecheck the following divergent process:

$$P_2 \quad \stackrel{\mathrm{def}}{=} \quad !p(a,b,c,d).\big(!a.\overline{c}.\overline{d} \mid !b.(\boldsymbol{\nu} e, f)\,\overline{p}\langle c,d,e,f\rangle\big) \;\mid\; \overline{p}\langle u,v,w,t\rangle.(\overline{u} \mid \overline{v}),$$

by setting $a\mathcal{R}c$ and $a\mathcal{R}d$. In P_2, the subterm replicated at b makes a recursive call to p with two new fresh names; the subterm replicated at a is typed using the partial order twice, and the outputs it triggers feed the loop.

Proposition 5.5.2 (Soundness)
System $\mathbf{S}^{\mathcal{R}}$ ensures termination.

Proof. *The main idea is to adapt the proof of Section 3.3. This can be done easily, as, in this setting, a measure decreases clearly at each reduction step. Moreover, as no input sequence is involved in the typing rules, we avoid the technicalities induced by the use of a complex measure computed on an auxiliary annotated calculus.*

The Subject Reduction result (the counterpart of the case 1. of Lemma 3.3.20) can be deduced easily by mimicking the proofs of Section 3.3, in a simpler way (as we said above, there is no annotation to take into account).

To derive soundness, we take, as a measure, the type of a process (the multiset of names N in $\Gamma\mathcal{R} \vdash P : N$) which corresponds, in this case, to the multiset of the available outputs $\mathbf{Os}(P)$, as in the termination proof of Section 3.1. We prove that, at each reduction step $P \to P'$, if $\Gamma\mathcal{R} \vdash P : N$, we have $\Gamma\mathcal{R} \vdash P : N'$ and $N :\!\succ_M^{\mathcal{R}_{\blacktriangleright(P)}} N'$, where $\mathcal{R}_{\blacktriangleright(P)}$ is the effective ordering associated to P, as defined in Definition 3.3.6 to handle the interactions between restrictions and the partial order. The comparison $M_1 :\!\succ_M^{\mathcal{R}} M_2$ is defined by: there exists K such that $M_1|_{(>K)} = M_2|_{(>K)}$, $M_1|_{(K)} = N \uplus \{a\}$, and

- *either $M_2|_{(K)} = N$*

- *or* $M_2|_{(K)} = N \uplus \{b\}$ *and* $a \mathcal{R} b$.

Clearly, if $\mathcal{R}$ *has a finite support,* $:\succ_M^{\mathcal{R}}$ *is well-founded, as it can be seen as a lexicographical composition of well-founded orderings.*

To obtain soundness, we suppose the existence of an infinite reduction sequence from a typed process and we first define $\mathcal{R}_\infty$*, as in Section 3.3, as the union of all* $\mathcal{R}_{\blacktriangleright(P_i)}$ *involved in this sequence* $(P_i)_i$*. We obtain directly from the Subject Reduction property, that if* $\Gamma\mathcal{R} \vdash P_i : N_i$ *and* $\Gamma\mathcal{R} \vdash P_{i+1} : N_{i+1}$ *then* $N_i :\succ_M^{\mathcal{R}_\infty} N_{i+1}$.

To conclude, we only need to state that the support of $\mathcal{R}_\infty$ *is finite. As condition* $\mathbf{safe}^{\mathcal{R}}(k, P)$ *in the rule* (**SRRep**) *allows the creation of new names of level* l *only when the number of output of level* l *decreases, we accommodate easily the proof of Proposition 3.3.29.* □

As written above, the inference for $\mathbf{S}^+$ is polynomial. Indeed, one can notice that even if we use a partial order allowing to compare names of same level, the lack of input sequence induces one-to-one comparisons: if an output on b is found in the continuation of a process guarded by a, then a dominates b. Yet, we have to decide if a dominates b because it has a greater level or because the use of the partial order, and this can be done efficiently, as shown by the following theorem.

Theorem 5.5.3 (Inference for $\mathbf{S}^{\mathcal{R}}$)
Type inference for system $\mathbf{S}^{\mathcal{R}}$ *is polynomial.*

The proof we present here is different from the one of [DS06], which develop an algorithm maintaining a unique set of constraints. We believe that this presentation eases the readability.

Proof.

As usual, solving the type inference problem for system $\mathbf{S}^{\mathcal{R}}$ *is finding a suitable assignment of levels and partial order to names. We start, as above, with a partition* $S_1, \ldots, S_n$ *of all names of the considered process such that all names in* S_i *must have same type.*

First we generate a set of domination constraints $\mathcal{C}$ *from the process* P*. For each replication* $!^r a(x).P$ *(the symbol* $!^r$ *is used to particularise each replication with an integer* r*), we add to* $\mathcal{C}$ *a set constraint* $(a, \mathtt{O} : r, b)$ *for each output* $\bar{b}\langle v\rangle$ *in* P *and a constraint* $(a, \mathtt{R} : r, c)$ *for each available restriction* (νc) *in* P*. The process is typable if and only if we are able to assign a level and a partial order relation to each name such that for every constraint* $(a, \mathtt{R} : r, b)$, $\mathtt{Lvl}(a) \geq \mathtt{Lvl}(b)$ *and for every* r *there is at most one constraint* $(a, \mathtt{O} : r, b)$ *such that* $\mathtt{Lvl}(a) = \mathtt{Lvl}(b)$ *and* $a\mathcal{R}b$*, and for all the other ones* $(a, \mathtt{O} : r, c)$, $\mathtt{Lvl}(a) > \mathtt{Lvl}(c)$ *and that if there is one constraint* $(a, \mathtt{O} : r, b)$ *there is no constraint* $(a, \mathtt{R} : r, c)$ *with* $\mathtt{Lvl}(a) = \mathtt{Lvl}(c)$.

Then, we check the presence of cycles between S_i*: that is, sequence of pair of names of length greater than 2 (this prevents a single name from being considered as a cycle)* $(a_{1,1}, a_{1,2}), \ldots, (a_{n,1}, a_{n,2})$ *such that, for each* i*, either* $(a_{i,1}, \mathtt{O} : r, a_{i,2})$ *for some* r*, or* $(a_{i,1}, \mathtt{R} : r, a_{i,2})$ *and there exists* S_i*,* $a_{i,2} \in S_i$ *and* $a_{i+1,1} \in S_i$*. If no such cycle can be found, we can rely on the proof of Proposition 5.2.3 to assign levels (and no partial order information at all) and derive typability.*

It is easy to prove that all names in a cycle have to be given the same type. Indeed, $a_{i,2}$ *and* $a_{i+1,1}$ *both belong to the same* S_i*, thus their types have to be the same. As* $a_{i,1}$ *and* $a_{i,2}$ *are related with a constraint (either* $(a_{i,1}, \mathtt{O} : r, a_{i,2})$ *or* $(a_{i,1}, \mathtt{O} : r, a_{i,2})$*), then* $\mathtt{Lvl}(a_{i,1}) \geq \mathtt{Lvl}(a_{i,2})$.

If such cycles exist, we use a topological sort on the graph of connected components of $\mathcal{G}$ *to assign levels (assigning the element of the same connected component to the same level). It is always possible and we check easily, after this operation, that if* $(a, \mathtt{O} : r, b)$*, then either* $\mathtt{Lvl}(a) \geq \mathtt{Lvl}(b)$ *, and if* $(a, \mathtt{R} : r, c)$*, then* $\mathtt{Lvl}(a) \geq \mathtt{Lvl}(c)$.

If there are cycles, the process is typable if and only if a partial order satisfy each comparison of type $(, \mathtt{O} :,)$ *present inside cycles, as names in a same cycle have to have the same type.*

We check that for each r*:*

- *there is at most one constraint* $(a, \mathtt{O} : r, b) \in \mathcal{C}$ *such that* $\mathtt{Lvl}(a) = \mathtt{Lvl}(b)$*, if it is not the case, the process is not typable, because the condition* $:\succ_M^{\mathcal{R}}$ *of the rule* (**SRRep**) *cannot be satisfied.*

- *if there is one* $(a, \mathtt{O} : r, b)$ *with* $\mathtt{Lvl}(a) = \mathtt{Lvl}(b)$*, we check that there is no constraint* $(a, \mathtt{R} : r, c)$ *with* $\mathtt{Lvl}(c) = \mathtt{Lvl}(a)$*. If it is the case, the process is not typable, because the safety condition of* (**SRRep**) *cannot be satisfied.*

For each constraint $(a, \mathtt{O} : r, b)$*, we set* $a\mathcal{R}b$*. Afterwards, we complete* $\mathcal{R}$ *by transitivity. At this step, the relation* $\mathcal{R}$ *can be implemented as a partial order using the typing rules if and only if the process is typable. We apply the following procedure to decide this property:*

1. *We check that* $\mathcal{R}$ *is indeed a partial order. If it is not the case, then the process is not typable: it means that we have a cyclic sequence of constraints* $(a_l, \mathtt{O} : r_l, a_{l+1})_{1 \leq l \leq p}$ *with* $a_p = a_1$*. No partial order can be used to satisfy this constraints.*

2. *We check that the partial order information can be put into types. For each pair* $a\mathcal{R}b$*,*

   - *we check that either* a *and* b *are free in* P*, or one is free and the other is bound by restriction, or both are bound by the same input.*

     If it is not the case, the process is not typable, as we cannot satisfy the partial order relations $\mathcal{R} = \mathcal{R}' \uplus \mathtt{R}_p\lfloor \widetilde{x} \rfloor$ *found in rules* (**SRIn**)*,* (**SRRep**) *and* (**SRRes**)*.*

     If it is the case, for instance if they are bound by a prefix $p(\ldots, a, \ldots, b, dots)$ *(*a *appearing in position* j_a *and* b *in position* j_b*) we add the relation* (j_a, j_b) *to the type of* p*.*

   We check that every $\mathtt{R}_p$ *associated to types defines indeed a partial order, if it is not the case, then the process is not typable.*

We compute $\mathcal{R}_0$ *as the restriction of* $\mathcal{R}$ *on names free in* P*.*

We easily now prove, by structural induction, that our type assignment induces a typing derivation for $\Gamma\mathcal{R}_0 \vdash P : N$*. Notice that we have to remember* $\mathcal{R}$ *in order to do that, as partial order information about names bound by restriction are not present in* $\mathcal{R}_0$*.*

The set of constraints $\mathcal{C}$ *is generated polynomially in the size of* P *(in can be done by checking once every prefix in* P*) and its size is also polynomial. Checking the presence of cycles, as defined above, is polynomial as we can use a topological sort of the graph of* S_i*. Computing the graph of connected components of* $\mathcal{G}$ *is also polynomial. When setting the partial order relation* $\mathcal{R}$ *by considering each cycles, we can rely only on cycles minimal for the prefix relation without loss of generality. Moreover, the set of minimal cycles can be constructed polynomially from the set of constraints. Computing the transitive closure of* $\mathcal{R}$ *can also be done in polynomial time and checking that* $\mathcal{R}$ *is a partial order also relies on topological sort. The remaining procedures treat once each pair* $a\mathcal{R}b$ *of the partial order, which is polynomial in the size on the initial process and check that all relation associated to types are partial order (again this in an instance of the topological sort algorithm).*

□

The expressiveness of $\mathbf{S}^{\mathcal{R}}$ can appear as somehow limited, compared to the one of $\mathbf{S}^{\mathrm{ord}}$ as we loose the ability to make "many-to-many comparisons". For instance the process $!p(a,b).a.(\overline{b} \mid \overline{p}\langle a, b\rangle)$, modelling a list data structure, is no longer typable.

However, we are able to use $\mathbf{S}^{\mathcal{R}}$ to recognise as terminating processes encoding lists data structures in the π-calculus. This can be done if we use processes similar to the following:

$$!create(node, key, value, succ).!node(rkey, ans).(\mathtt{if}\ rkey = key\ \mathtt{then}\ \overline{ans}\langle value\rangle\ \mathtt{else}\ \overline{succ}\langle rkey, ans\rangle).$$

Here the channel *create* can be used to create a new node in the list, whose name is *node*, pointing to its successor, whose name is *succ* and containing a key and a value. If a request is sent on *node* and the key requested is not the one stored, then this request is transmitted to *succ*. A consequence for typing is that *succ* and *node* have necessarily same type, as both can be used as first or fourth argument of *create*. $\mathbf{S}^{\mathcal{R}}$ can typecheck this process by adding in the type of *create* a partial order information stating that its first argument dominates its second one.

Actually, the expressive powers of $\mathbf{S}^{\mathcal{R}}$ and $\mathbf{S}^{\mathrm{ord}}$ are not comparable, as there exist processes which are typable in the new type system although they are rejected by the former one. Indeed, the condition used in the typing rule (**SRRep**) allows $\mathbf{S}^{\mathcal{R}}$ to type processes where the total weight, for the former definition of weight (in this case, the multiset of the levels of the available outputs), increases along reduction step. Consider $p(a,b).!a.(\overline{b} \mid \overline{c} \mid \overline{c}) \mid \overline{q}\langle u\rangle \mid \overline{q}\langle v\rangle \mid \overline{p}\langle u,v\rangle$. The two outputs on q force the types of u and b to be the same, whichever type system we are willing to use. The output and the input on p force the types of a and b to be the same. As a consequence, whatever levels we give to a,b,c (assume level k for a,b and l for c), the replication is not typable in $\mathbf{S}^{\mathrm{ord}}$ as the comparison involved is $\{k\}$ against $\{k,l,l\}$. Yet, using $\mathbf{S}^{\mathcal{R}}$ we can relate a,b by the partial order and the condition of rule (**SRRep**) is satisfied.

Chapter 6

Semantics-based methods for termination in π-calculus

6.1 Limitations of weight-based systems

In this section, we show the limitations of the weight-based type systems by studying the termination method for programs written in the formalism of the λ-calculus obtained by translating a λ-term into a π-process using the encoding presented in Definition 2.4.3.

This method of termination is considered with respect to a strategy (we explained it in Section 2.4.4) of reduction for λ in the sense that we cannot use it to prove termination in the setting of the full λ-calculus this way. Notice that there exist in [SW01] a translation from the λ-calculus to the π-calculus accommodating reductions under an abstraction (this is not a strategy, but it is not the strong λ-calculus neither).

6.1.1 Typing through the encoding

We recall here the trivially terminating process P_{II}, which is the encoding on the channel p_1 of a simple λ-calculus term: the identity applied to itself.

$$P_{II} = (\nu q_1, r_1)\ (\nu y_2)\ (\overline{q_1}\langle y_2\rangle.!y_2(x_2, q_2).\overline{q_2}\langle x_2\rangle) \mid (\nu y_3)\ (\overline{r_1}\langle y_3\rangle.!y_3(x_3, q_3).\overline{q_3}\langle x_3\rangle) \mid q_1(y_1).r_1(z_1).\overline{y_1}\langle z_1, p_1\rangle)$$

This process can actually be typechecked using the type system of Section 3.1:

$$\Gamma \vdash P_{II} : 1$$

with Γ defined by: $x_3 : \mathbb{1}$ $\quad y_3, z_1, x_2 : \sharp^1\,\Gamma(x_3) \times \sharp^0\,\Gamma(x_3)$ $\quad y_2, y_1 : \sharp^1\,\Gamma(y_3) \times \sharp^0\,\Gamma(y_3)$ $\quad q_3 : \sharp^0\,\Gamma(x_3)$

$$q_2 : \sharp^0\,\Gamma(y_3) \qquad\qquad q_1 : \sharp^0\,\Gamma(y_2)$$

Here both replications are well-typed because the replicated inputs are performed on names of level 1 (resp. y_2 and y_3) and the outputs in the continuations are of level 0 (resp. q_2 and q_3). We can generalise this to obtain a small terminating class of process (and, as a result, a terminating class of λ-terms): indeed, termination is ensured by remarking that only the dyadic channels are used as subject of replicated inputs and only monadic channels are used as subject of outputs under replication. Processes of this form are always validated by our weight-based type system as the level assignment mapping the dyadic types to 1 and the monadic ones to 0 satisfies every constraint we can raise. This corresponds, seen through the encoding, to the class λ-terms where no abstraction $\lambda x.M$ contains an application inside M. This class of λ-term can easily be proved as terminating by showing that the number of applications strictly decreases at each β-reduction step.

In the previous sections we developed type systems ensuring termination for simply-typed π. As the image by the CbV encoding of the simply-typed λ-calculus ($\lambda_{\mathbf{ST}}$ – see Section 2) is contained inside the simply-typed π, one can wonder if we can use the type systems for termination we presented above to obtain a new proof of strong normalisation for $\lambda_{\mathbf{ST}}$. One has to remember from [GTL89] that if they can exist such "arithmetic" proofs of termination for the simply-types λ, the termination proofs for the polymorphic λ (System F) cannot be expressed inside the second-order arithmetic, as the termination of F implies the coherence of this arithmetic.

Remark 6.1.1 (Statman's theorem) *One one side, in [Sta79] a theorem stating that the evaluation in the λ-calculus is not elementary recursive can be found. On the other side, the termination of a π-term type-checked with the system of Section 3.1 is bound by an exponential function (as stated in Proposition 3.1.15). As a consequence, it seems difficult for our systems to handle the expressive power of the λ-calculus. However, one has to be careful using such arguments, as the result from [Sta79] holds for the strong λ-calculus and not for an evaluation strategy.*

6.1.2 Top-down typing constraints

We correct here a mistake found in [DS06], where a (false) counter-example to the possibility of using method described above to obtain termination of $\lambda_{\mathbf{ST}}$ is given.

Consider the λ-term $M_1 = \lambda x.(\lambda y.y)\ c$, its translation, according to Definition 2.4.3 is:

$$[M_1]_p = (\nu p_1)\ \overline{p}\langle p_1\rangle.(\nu q_1, r_1)\ ((\nu p_2)\overline{q_1}\langle p_2\rangle.!p_2(y, x_1).\overline{x_1}\langle y\rangle \mid \overline{q_2}\langle c\rangle \mid \ \mid q_1(y_1).r_1(z_1).\overline{y_1}\langle z_1, p_1\rangle)$$

Then the authors of [DS06] claim that y_1 and p_2 have necessarily the same type and that, as a consequence, the term is not typable (because y_1 appears as a free output inside the continuation guarded with $!p_2(\dots)$). This claim is actually false, because even if p_2 and y_1 have simple channel-types which denote the same λ-type, the typing context can still assign these to names to two different types (especially to two different levels). Actually, one is able to type this process by giving the level 1 to p_2 and 0 to y_1. What forces two names to have the same type is the fact that these two names are both arguments of the same name (or, recursively, arguments of names which have to have same type). In the setting of $[M_1]_p$, nothing prevents us from giving them two different types. What we must look for, when trying to find a counter-example to such a method for termination of $\lambda_{\mathbf{ST}}$, is a setting where we have indeed an output on a inside a subprocess guarded with a replicated input on b, a and b being forced to have same type by *top-down constraints*, i.e. because they are used somewhere in the process in the same way.

For instance in $\overline{p}\langle a\rangle \mid q(x) \mid \overline{c}\langle q, z\rangle \mid \overline{c}\langle z, p\rangle$, the names a and x must have the same type, as a is sent on p, x is received on q and p and q must have the same type. Indeed, the latter holds because q and z are both used as the first argument of the name c, and z and p are both used as the second argument of the name c.

On the contrary, bottom-up constraints do not force two names to have the same type (i.e. two names carrying the same name can have different levels). For instance in $!q(z).\overline{p}\langle x\rangle$, both p and q carry the name x, but their types can be different: if a has type T, then we can assign the type $\sharp^1 T$ to q and the type $\sharp^0 T$ to p.

6.1.3 An actual counter-example

We found such a counter example proving that our type systems based on level-assignment and measure-decreasing are not expressive enough to recognise as terminating the translation of $\lambda_{\mathbf{ST}}$.

Proposition 6.1.2 (Limits of the weight-based type systems)
Termination of $\lambda_{\mathbf{ST}}$ cannot be obtained by using the type-systems of previous sections through the CbV-encoding.

Proof.
Consider the λ-term $M_0 = f\ (\lambda x.(f\ u)\ (u\ v))$.

M_0 is in $\lambda_{\mathbf{ST}}$:

$$\Gamma \vdash^{ST} f\ (\lambda x.(f\ u)\ (u\ v)) : \tau \to \tau$$

with $\Gamma = f : (\sigma \to \tau) \to \tau \to \tau, u : \sigma \to \tau, v : \sigma, x : \sigma$.

Indeed, $u\ v$ has type τ, $f\ u$ has type $\tau \to \tau$, thus $(f\ u)\ (u\ v)$ has type τ, $\lambda x.(f\ u)\ (u\ v)$ has type $\sigma \to \tau$ and finally $f\ (\lambda x.(f\ u)\ (u\ v))$ has type $\tau \to \tau$.

Following the rules of Definition 2.4.3, we obtain:

$$\begin{aligned}[M_0]_{a_0} = \quad & (\nu q_0, r_0)\ q_0(m_1).r_0(n_1).\overline{m_1}\langle n_1, a_0\rangle \mid \overline{q_0}\langle f\rangle \\ & \mid (\nu y)\ \overline{r_0}\langle y\rangle.!y(x, b_1).[(\nu q_1, r_1)\ q_1(m_2).r_1(m_2).\overline{m_2}\langle n_2, b_1\rangle \\ & \qquad \mid (\nu q_2, r_2)\ (q_2(m_3).r_2(m_3).\overline{m_3}\langle n_3, q_1\rangle \mid \overline{q_2}\langle f\rangle \mid \overline{r_2}\langle u\rangle) \\ & \qquad \mid ((\nu q_3, r_3)\ (q_3(m_4).r_3(n_4).\overline{m_4}\langle n_4, r_1\rangle \mid \overline{q_3}\langle u\rangle \mid \overline{r_3}\langle v\rangle))]\end{aligned}$$

As we manipulate a lot of different names, and as the process itself is difficult to read, we present, for the sake of clarity, our proof as a sequence of short claims, which can be easily proof-checked:

1. *y and n_1 are both arguments of r_0.*
2. *n_1 is the first argument of m_1.*
3. *m_1 and f are both arguments of q_0.*
4. *m_3 and f are both arguments of q_2.*
5. *n_3 and u are both arguments of r_2.*
6. *n_3 is the first argument of m_3*
7. *m_3 and m_1 have the same type (3+4)*
8. *n_1 and n_3 have the same type (2+6+7)*
9. *y and n_3 have the same type (1+8)*
10. *y and u have the same type (5+9)*
11. *u and m_4 are both arguments of q_3*
12. *y and m_4 have the same type (10+11)*

Finally, as an output on m_4 is present in the continuation of the replicated input on y, no type system presented in the previous section is able to handle this process.

□

Remark 6.1.3 (Call-by-Name) *By referring to [SW01], one can construct a counter-example similar to the one above for the CbN encoding of the λ-calculus into the π-calculus.*

Remark 6.1.4 *One can notice that the counter-example we give in the proof of Proposition 6.1.2 is not typable because two names* a priori *not related are given the same type, leading to an unsatisfiable constraint.*

6.2 Logical relations for a small functional π-calculus

The results in this section are originally presented in [San06], we briefly recall them here as they will be useful in the next section. We do not detail the similar (at least in the proof method used) results presented in [YBH04]. We explain later in which sense the two techniques differ and, at the end of this section, we sum up the limitations of these two techniques, in terms of expressiveness.

$$
\begin{array}{rcl}
P & ::= & (\nu a)\ (I_a \mid P) \text{ if } \mathrm{fn}(P) \cap \mathrm{fn}(I_a) = \emptyset \\
 & \mid & M \\
 & \mid & I_a \\
\\
I_a & ::= & !a(x).M \text{ if } a \notin \mathrm{fn}(M) \\
\\
M & ::= & (\nu a)\ (I_a \mid M) \\
 & \mid & M \mid M \\
 & \mid & \overline{a}\langle v \rangle \\
 & \mid & \mathbf{0}
\end{array}
$$

Figure 6.1: Syntax for preprocesses, resources and processes in normal-form of $\pi^{(1)}_{\mathtt{def}}$

6.2.1 Types for functional process

In this section, we present a type system ensuring termination of a small subcalculus $\pi^{(1)}_{\mathtt{def}}$ contained in the π-calculus, using logical relations. This result, found in [San06], is briefly recalled here as we make use of it in the following sections.

The crucial point of this system is that syntactical constraints prevent the definition of diverging process. We only assign simple types to names, no additional informations (such as levels or partial orders) are required in this setting.

Yet, we assign more complex types to processes. Either a process is considered as a "complete" one and is given type $\diamond$. Or a process is considered as a "service" and is given type $\mathbf{T}(a, T)$ meaning that it offers a functional service on name a of type T. As a consequence, the syntax for process types is:

$$T_P ::= \diamond \mid \mathbf{T}(a, T)$$

The syntax of $\pi^{(1)}_{\mathtt{def}}$, the language of well-formed functional processes, is obtained by closing under structural congruence the normal-forms defined by the grammars found in Figure 6.1. Thus, every process is structurally congruent to a process in normal form and we can focus our analysis on the latter. We dot not present here the definition of structural congruence in $\pi^{(1)}_{\mathtt{def}}$, as it is similar to the one defined in Section 2. Processes in normal-form (denoted by the letter M) are either the inactive process ($\mathbf{0}$), a parallel composition of two processes, an output or a *definition*, which is the parallel composition of a process and a resource on the name a, under the restriction on a. A resource on a is a replicated input on a, whose continuation is a process. Used in the proofs, preprocess are either processes, resources, or definitions $(\nu a)\ (I_a \mid P)$ where the free names of P and I_a are distinct.

We write $(\nu \widetilde{a})\ (I_{\widetilde{a}} \mid M)$ for the process $(\nu a_1)\ (I_{a_1} \mid (\nu a_2)(I_{a_2} \mid \ldots \mid M))$. We say that a preprocess P is in normal form when: it is either of the form $(\nu \widetilde{a})\ (I_{\widetilde{a}} \mid M)$ or of the form $(\nu \widetilde{a})\ (I_{\widetilde{a}} \mid I_b)$ with $b \notin \widetilde{a}$.

Definition 6.2.1 (Typing preprocesses)
Preprocesses in normal-form are typed the following way:

- $\Gamma \vdash_{\mathtt{fun}} (\nu a)\ (I_{\widetilde{a}} \mid M) : \diamond$
- $\Gamma \vdash_{\mathtt{fun}} (\nu a)\ (I_{\widetilde{a}} \mid I_b) : \mathbf{T}(b, T)$ *if b has simple type T.*

6.2.2 Termination proof

We do not give all the details of the proof, as no new contribution is added to [San06]. Yet, we give the crucial lemmas of this proof technique, in order to show how it differs from the weight-based techniques we developed previously.

As hinted above, the logical relations technique defined interpretations of process types as sets of processes. We write $P \in [\![T]\!]$ (P realises the type T) to state that P is in the interpretation of T.

Definition 6.2.2 (Logical relations)

- $P \in [\![\diamond]\!]$ *if* $\Gamma \vdash_{\mathtt{fun}} P : \diamond$ *and* P *terminates.*
- $P \in [\![\mathbf{T}(a, \sharp(\mathbb{1}))]\!]$ *if* $\Gamma \vdash_{\mathtt{fun}} P : \mathbf{T}(a, \sharp(\mathbb{1}))$ *and* $(\nu a)\ (P \mid \overline{a}) \in [\![\diamond]\!]$
- $P \in [\![\mathbf{T}(a, \sharp(T))]\!]$ *if* $\Gamma \vdash_{\mathtt{fun}} P : \mathbf{T}(a, \sharp(T))$ *and for all* I_b *such that* b *is fresh and* $I_b \in [\![\mathbf{T}(b, T_b)]\!]$, $(\nu b)\ (\overline{a}\langle b\rangle \mid I_b \mid P) \in [\![\diamond]\!]$

It is easy to prove the following fact from this definition:

Fact 6.2.3 (Termination of interpretation)
If $P \in [\![T_P]\!]$ *then* P *terminates.*

Proof.
Easily done by examining the form of P.

□

To derive soundness, we introduce the notion of relatively independent resources, which are resources which cannot share the services they offer.

Definition 6.2.4 (Relatively independent resources)
Resources $I_{a_1}, \ldots, I_{a_n}$ *are* relatively independent *if for all* i, j, a_i *does not appear free in output position in* I_{a_j}.
A process (respectively preprocess, resource) has relatively independent resources*, if for all of its subprocesses, the unguarded resources are relatively independent.*

By preventing a_i from appearing in output position in I_{a_j}, one ensures that the different resources cannot call each other. The remaining of the proof makes use of the barbed bisimilarity $\sim$, which is a standard behavioural equivalence. Details of the definition, as well as some useful properties which will be used later can be found in [San06] and are omitted here.

Lemma 6.2.5 (Simulation Lemma)
For any resources I_a *(respectively process* M*), there exists* $J_a \sim I_a$ *(respectively* $M' \sim M$*) such that* J_a *(respectively* M'*) has relatively independent resources.*

Proof. *By structural induction on* I_a *(respectively* M*), using some properties of* $\sim$ *(again details can be found in [San06]).* □

The main proof makes also use of "forwarders" which are simple replicated process $!c'(x).\overline{c}\langle x\rangle$ trading outputs on one name to outputs on another names.

Fact 6.2.6 (Properties of forwarders)

1. *If* $\Gamma(a) = \Gamma(b) = \sharp(T_x)$ *and* $\Gamma(x) = T_x$*, then* $!a(x).\overline{b}\langle x\rangle \in [\![\sharp(T_x)]\!]$.
2. $(\nu b)\ (I_b \mid \overline{b}\langle c\rangle)$ *terminates if and only if* $(\nu b, c')\ (I_b \mid !c'(x).\overline{c}\langle x\rangle \mid \overline{b}\langle c'\rangle)$ *terminates.*

Proof.

- *Either* $T_x = \mathbb{1}$ *and* $(\nu a)\ (!a.\overline{b} \mid \overline{a})$ *reduces to a process equivalent* ($\sim$) *to* $\overline{b}$*, which terminates. Or* $T_x = \sharp(T)$ *for some* T *and we have* $(\nu c)\ (I_c \mid (\nu a)\ (!a(x).\overline{b}\langle x\rangle \mid \overline{a}\langle c\rangle))$ *which reduces into* $(\nu c)\ (I_c \mid \overline{b}\langle c\rangle)$ *which is terminating (it cannot reduce further).*

- *If c' does not occur free in input position in the process R, one can prove that $(\nu c')\ (R \mid !c'(x).\overline{c}\langle x\rangle)$ diverges if and only if $R\{c/c'\}$ diverges.*

□

The following lemma is the crucial point of this proof:

Lemma 6.2.7 (Main Lemma)
If $\Gamma \vdash_{\mathtt{fun}} P : T_P$ then, for all sequences of relatively independent resources $(I_{a_i})_{1\leq i\leq n}$ s.t. for all i, $I_{a_i} \in [\![T_i]\!]$ for some T_i, we have $(\nu\widetilde{a})\ (I_{\widetilde{a}} \mid P) \in [\![T]\!]$ for some T.

Proof. *By structural induction over P, we present here the two main cases of this proof: outputs and parallel compositions:*

- *Case $\overline{a}\langle v\rangle$. We have $T_P = \diamond$ and $\Gamma \vdash_{\mathtt{fun}} (\nu\widetilde{a})\ (I_{\widetilde{a}} \mid \overline{a}\langle v\rangle) : \diamond$. We have to prove that $(\nu\widetilde{a})\ (I_{\widetilde{a}} \mid \overline{a}\langle v\rangle)$ terminates. As the resources are relatively independent, we derive that $(\nu\widetilde{a})\ (I_{\widetilde{a}} \mid P) \sim (\nu\widetilde{a}')\ (I_{\widetilde{a}'} \mid P)$ with $\widetilde{a}' = \widetilde{a} \cap \{b, c\}$. We discuss the nature of $\widetilde{a}'$:*
  - *Either $\widetilde{a}' = \emptyset$ and the process is equivalent ($\sim$) to $\overline{b}\langle c\rangle$ which terminates.*
  - *Or $\widetilde{a}' = \{b\}$ and the process is equivalent to $(\nu b)\ (I_b \mid \overline{b}\langle c\rangle)$. We use Fact 6.2.6 to state that the termination of this process is equivalent to the one of $(\nu b, c')\ (I_b \mid !c'(x).\overline{c}\langle x\rangle \mid \overline{b}\langle c'\rangle)$ and we conclude as $I_b \in [\![T_b]\!]$ and $!c'(x).\overline{c}\langle x\rangle \in [\![T_{c'}]\!]$.*
  - *Or $\widetilde{a}' = \{c\}$ and the process is equivalent to $(\nu b)\ (I_c \mid \overline{b}\langle c\rangle)$ which cannot reduce further.*
  - *Or $\widetilde{a}' = \{b, c\}$ and the process is equivalent to $(\nu b)\ (I_b \mid (\nu c)\ (I_c \mid \overline{b}\langle c\rangle))$ (we use the relative independence to move the restriction on c inside). We conclude using the definition of $I_b \in [\![T_b]\!]$.*
- *Case $M_1 \mid M_2$. We have to show that $(\nu\widetilde{a})\ (I_{\widetilde{a}} \mid M_1 \mid M_2)$ terminates. We use the properties of $\sim$ to derive that this is equivalent to the termination of $(\nu\widetilde{a})(I_{\widetilde{a}} \mid M_i) \mid (\nu\widetilde{a'})(I_{\widetilde{a'}} \mid M_i\{\widetilde{a'}/\widetilde{a}\})$. We conclude using the induction hypothesis on the M_i.*

□

Termination of $\pi^{(1)}_{\mathtt{def}}$ follows directly from this result.

Remark 6.2.8 (Typing the encoding of $\lambda_{\mathbf{ST}}$) *By carefully examining the standard CbV encoding of the λ-calculus into the π-calculus given in Section 6.1, one can notice that the π-processes corresponding to simply-typed λ-terms are contained into the functional sub-calculus which is proved terminating by the present technique. This implies that this technique is more expressive than the weight-based ones when it comes to typing functional processes. However, we shall show it later, its expressive power on standard concurrent protocols is somehow limited.*

Remark 6.2.9 (Another semantics-based technique) *The technique developed in [YBH04] differs from the one we presented above. Syntactical constraints on the inputs are less strong (for instance, non-replicated inputs exists). Instead, a type system ensures linearity: a name is either replicated, and there exists a unique replicated input on this name in the process, or linear, and there exists exactly one input and one output on this name in the process.*

The soundness proof makes use of logical relations (by defining inductively terminating semantical interpretations of types and proving afterwards that each typable term belongs to the interpretation of its type).

The article [YBH04] also proposes a proof that the standard encoding of $\lambda_{\mathbf{ST}}$ is recognised as terminating by this technique.

6.2.3 A new presentation for $\pi^{(1)}_{\texttt{def}}$

We give here a new presentation for $\pi^{(1)}_{\texttt{def}}$, called $\pi^{1}_{\texttt{def}}$, which makes the results of the following section easier to deal with.

We define the syntax of $\pi^{1}_{\texttt{def}}$ as:

$$P ::= \mathbf{0} \mid \texttt{def } f = (x).P \texttt{ in } P \mid \overline{f}\langle v\rangle \mid (P \mid P)$$

We further require, as a syntactical constraint, that $\texttt{def } f = (x).P_1 \texttt{ in } P_2$ binds f and that f does not appear in P_1.

Here, restrictions and replicated inputs are merged into a single construct. The new constructor, called *definition*, is used only for the sake of clarity and $\texttt{def } f = (x).P_1 \texttt{ in } P_2$ is meant to be actually $(\nu f)\ (!f(x).P_1 \mid P_2)$.

Semantics takes into account the new operator: reductions can occur in the second part of a definition, but not in the first one. The former represents the active part of the process whereas the latter represents a functional service available.

$$\mathbf{E} ::= [\,] \mid (\mathbf{E} \mid P) \mid \texttt{def } f = (x).P \texttt{ in } \mathbf{E}$$

The communication rule of the semantics is replaced by a (**Call**) rule.

$$(\mathbf{Cong})\frac{P \equiv Q \qquad Q \to Q' \qquad Q' \equiv P'}{P \to P'}$$

$$(\mathbf{Call})\frac{}{\mathbf{E}_1[\texttt{def } f = (x).P_1 \texttt{ in } \mathbf{E}_2[\overline{a}\langle v\rangle]] \to \mathbf{E}_1[\texttt{def } f = (x).P_1 \texttt{ in } \mathbf{E}_2[P_1\{v/x\}]]}$$

The operator `def` is only syntactical sugar, and the calculus is the same as the one defined in the previous section, yet, its presentation is more convenient to work with.

Proposition 6.2.10 (Simulation result)
If P is a $\pi^{1}_{\texttt{def}}$ process, $\texttt{LetRem}(P)$ *is obtained by replacing inductively each definition* $\texttt{def } f = (x).P_1 \texttt{ in } P_2$ *in P by $(\nu f)\ (!f(x).P_1 \mid P_2)$.*
`LetRem`*() induces a bisimulation:*

- *If $P \to P'$, then* $\texttt{LetRem}(P) \to \texttt{LetRem}(P')$.
- *If* $\texttt{LetRem}(P) \to \texttt{LetRem}(P')$, *then $P \to P'$.*

Proof. *Easily done by induction over the reduction derivation.* □

Proposition justifies the use of $\pi^{1}_{\texttt{def}}$ in the following section. By regrouping restrictions and replicated input in only one construct, the proofs of the results of the following section will be easier, as, by proceeding by induction over the structure of the processes of $\pi^{1}_{\texttt{def}}$ or over their typing derivations, one can examine the single case of the definition, instead of examining cases for restriction, parallel composition and replicated input.

6.2.4 Limitations of the semantics based techniques

These two terminating subcalculi (the one of [San06] and the one of [YBH04]) are somehow limited when it comes to express non-functional behaviours. Indeed, they are both *confluent*. Confluence is formally defined by:

$$\forall P, Q, Q', (P \to Q \wedge P \to Q') \Rightarrow \exists (P', Q \to^* P' \wedge Q' \to P')$$

It means that, when two reductions are possible, choosing one does not lead us to states which would have been unreachable if we had chosen the other. One can see the confluence property as something "not concurrent", because as a consequence, competition for a resource is a behaviour which cannot be properly modelled by a confluent program.

The calculus of [San06] contains only replicated inputs and only one input on a given name can exist inside a process. This means that a name f is associated to a unique and immutable service (the continuation P_1 in $(\nu f)\ (!f(x).P_1 \mid P_2)$) (this property is called *uniform receptiveness*). As a result, if two reductions are available, one involving an output $\overline{f_1}\langle v_1 \rangle$ and the other $\overline{f_2}\langle v_2 \rangle$, choosing to reduce one of this two outputs cannot prevent us to reduce the other one later, meaning that the calculus is confluence.

In [YBH04], the authors explicitly prove that their calculus is *convergent* (which means confluent and terminating).

As a consequence, the state of termination analysis for mobile processes so far is as follows: either one uses a weight-based methods, and some terminating functional behaviours cannot be recognised as such; or one chooses to use one of these two semantics-based methods, and some common terminating example of concurrent programs are rejected by the analysis. This justifies the introduction of a third method, making use of both results, in order to ensure termination in environments where both kinds of behaviours are combined.

Chapter 7

Termination in impure languages

In this section, we study how the two approaches for termination we presented, the one based on measure-decreasing and the one based on logical relations, can be put together. The two calculi we will present (one is another presentation of standard π, the other one is a variant of the λ-calculus featuring references) are divided into an imperative part, whose termination is proved using the first method and a functional part, whose termination is proved using the second one. As we will explain later, the task of merging the two termination proofs is not trivial; actually, one has to extend the control of weight to the functional part. Moreover, the proof technique itself is peculiar; we use a pruning of the calculus to obtain, towards a contradiction with the logical relation result, a diverging functional process from a diverging process. The proof framework is the same for both calculi, although the technical details greatly differ: for instance, the pruning has to be done more carefully in the sequential case, as the types for λ-terms (arrow types) exhibit more structure than the types for π-processes (a single integer). The results of this section are presented in [DHS10b].

7.1 In an impure π-calculus

7.1.1 A π-calculus with functional names

The π-calculus, in the presentation we give in Section 2, 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. (cf. the uniform-receptiveness discipline, [San99]). The remaining names are called *imperative*.

In the π-calculus we use in this section, functional names are introduced in a `def` construct, akin to a "letrec" (we could as well have distinguished them using types). Notice that this distinction is purely syntactic and that the calculus we will use here, from the point of view of semantics, is actually essentially the same as the one presented in section 2.

The ordinary restriction operator of the π-calculus, in contrast, is only used to introduce imperative links, which simplifies the presentation. 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 the grammar below, the only first-order value is $\star$. In examples, later, the grammar for values may be richer (integers, tuples,

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

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

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

Figure 7.1: Reduction for $\pi_{\mathtt{ST}}$

etc.); such additions are straightforward but would make the notations heavier.

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

We call $\pi_{\mathtt{ST}}$ the simply-typed version of this calculus. Typing ensures that the subject v of inputs $v(x).P$ and $!v(x).P$ is always an imperative name.

As in section 2, we suppose that every process we consider in the following abides a Barendregt convention, that is, its bound names are pairwise distinct and are distinct from its free names.

Evaluation contexts are given by the following grammar:

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

Again, one can remark easily that these evaluation contexts correspond to the ones presented in section 2.

In this setting, structural congruence between processes, also 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:

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

$$\begin{array}{l} P \,|\, \texttt{def } f = (x).Q_1 \texttt{ in } Q_2 \;\equiv \\ \quad \texttt{def } f = (x).Q_1 \texttt{ in } P|Q_2 \quad \text{ if } f \notin \mathrm{fn}(P) \end{array} .$$

(Rules for replication or for swapping consecutive restrictions or definitions are not needed in $\equiv$.) We use $\equiv$ also on evaluation contexts.

Figure 7.1 presents the rules defining the reduction relation on $\pi_{\mathtt{ST}}$ processes. To define free names and substitutions, we use the definitions we give for standard π, accommodating the presence of the `def = in` construct. We impose that, in rule (**call**), the intrusion of $P\{v/x\}$ in the context $\mathbf{E}_2$ avoids name capture.

We say that a reduction $P \to P'$ is *functional* (resp. *imperative*) when it is derived using rule (**call**) (resp. rule (**comm**) or (**trig**)).

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

Termination constraints and logical relation proofs for λ-calculi (the ones cited in Section 2) can be adapted to $\pi_{\mathtt{def}}$. In the simply-typed case, the only additional constraint is that definitions $\texttt{def } f = (x).P \texttt{ in } Q$ cannot be recursive (f is not used in the body P). We call $\pi^1_{\mathtt{def}}$ this language.

We call $\pi^1_{\mathtt{imp}}$ the terminating language composed of the set of all processes typable by the type system of Section 3.1.

7.1.2 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 $\pi^1_{\mathtt{def}}$. 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, imperative inputs respect the levels as the level of a can be higher than the level of f). Similarly, in

$$\begin{array}{l}\mathtt{def}\ f = (x).(\overline{x}|\overline{a}\langle x\rangle)\ \mathtt{in}\\ \quad \mathtt{def}\ g = a(y).\overline{f}\langle y\rangle\ \mathtt{in}\ \overline{g}\,|\,\overline{a}\langle g\rangle\end{array}$$

the definitions of f and g have a cyclic dependency, implemented via the imperative name a, that causes divergence, but the constraints in $\pi^1_{\mathtt{def}}$ and $\pi^1_{\mathtt{imp}}$ can be satisfied. Termination is guaranteed if the measure constraint of imperative names is extended to the functional ones, viewing a `def` as an input. For instance, the first process above would now be rejected because the `def` requires f to be heavier than a, whereas the replication requires the opposite. This extension would however affect the class of functional processes accepted, which would not anymore contain $\pi^1_{\mathtt{def}}$ and the process images of the simply-typed λ-calculus. (Intuitively, processes terminate "too quickly" 3.1.)

In the solution we propose, we do impose measures onto the functional names, but the constraints on them are more relaxed with respect to those for imperative names.

This way, $\pi^1_{\mathtt{def}}$ is captured. The drawback is that measure alone does not anymore guarantee termination. However, it does if the purely functional sublanguage (in our case $\pi^1_{\mathtt{def}}$) is terminating.

As usual, 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:

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

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 $\Gamma \vdash P : l$, where l is the *level* (or weight) *of* P. It is defined by the rules of Figure 7.2; on values, $\Gamma \vdash v : T$ holds if either $\Gamma(v) = T$, or $v = \star$ and $T = \mathtt{unit}$. We call $\pi^1_{\mathtt{ST}}$ the set of well-typed processes. The typing judgement is extended to evaluation contexts by adding the axiom $\Gamma \vdash [\,] : 0$.

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 (**PFIn**) and (**PFRep**) implements the measure constraint for $\pi^1_{\mathtt{imp}}$ discussed in Section 7.1.1. ensures us the control of levels on imperative reductions discussed above: the output released are strictly "lighter" than the one consumed. In rule (**PFDef**) the constraint is looser: the outputs released can be as heavy as the one 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 outputs (which would happen if a functional output underneath an imperative input a could cause the release of imperative outputs heavier than a). In the same rule (**PFDef**), the constraint $f \notin \mathrm{fn}(P_1)$ is inherited from $\pi^1_{\mathtt{def}}$.

The set of well-typed process is a calculus, as expressed by the following four lemmas:

Lemma 7.1.1 (Subject Congruence)
If $P \equiv Q$ then $\Gamma \vdash P : l$ if and only if $\Gamma \vdash Q : l$.

$$(\textbf{PFName})\frac{\Gamma(v) = T}{\Gamma \vdash v : T} \qquad (\textbf{PFUnit})\frac{}{\Gamma \vdash \star : \mathtt{unit}}$$

Typing rules for processes

$$(\textbf{PFNil})\frac{}{\Gamma \vdash \mathbf{0} : 0} \qquad (\textbf{PFRes})\ \frac{\Gamma, a : \sharp^k_{\mathtt{I}}\, T \vdash P : l}{\Gamma \vdash (\boldsymbol{\nu} a)\, P : l}$$

$$(\textbf{PFPar})\frac{\Gamma \vdash P_1 : l_1 \qquad \Gamma \vdash P_2 : l_2}{\Gamma \vdash P_1 \mid P_2 : \max(l_1, l_2)}$$

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

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

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

$$(\textbf{PFOut})\ \frac{\Gamma \vdash P : l \qquad \Gamma \vdash v : \sharp^k_{\bullet}\, T \qquad \Gamma \vdash w : T}{\Gamma \vdash \overline{v}\langle w \rangle.P : \max(k, l)}$$

Figure 7.2: Typing Rules for $\pi^1_{\mathtt{ST}}$

Proof. *By induction on the derivation of $P \equiv Q$, using the symmetry and associativity of the operator* max, *and the fact that* 0, *the weight of* $\mathbf{0}$, *is neutral.* □

The Subject Substitution property for this calculus claims that typability (and type actually) is preserved by a well-typed substitution (a substitution $\{v/x\}$ where x and v have same type). The proof is standard, the case (**PFDef**) is done as if it was a parallel composition (as we require the name being substituted to be free in P, this name cannot be the defined name).

Lemma 7.1.2 (Subject Substitution)
If $\Gamma(x) = \Gamma(v)$, $x \in \mathrm{fn}(P)$ *and* $\Gamma \vdash P : l$ *then* $\Gamma \vdash P\{v/x\} : l$.

Proof.
The proof is similar to the ones of the previous subject substitution lemmas (for instance Lemma 3.2.10). Details can be found in Appendix A.

□

As we choose, as previously, to use a context-based semantics, we have to state how typability accommodates evaluation contexts. The definition of evaluation contexts ensures that the weight of the inside process is smaller than the weight of the whole process. Moreover, we can replace the inside process by any typable process which weights no more.

Lemma 7.1.3 (Context typing) *If* $\Gamma \vdash \mathbf{E}[P] : l$, *then*

1. $\Gamma \vdash P : l'$ *for some* $l' \leq l$.
2. *For all* P_0 *s.t.* $\Gamma \vdash P_0 : l_0$ *with* $l_0 \leq l'$, *then* $\Gamma \vdash \mathbf{E}[P_0] : l_{(0)}$ *with* $l_{(0)} \leq l$.

Proof. *By structural induction on* $\mathbf{E}$.

- *Case* $[\,]$ *is trivial and case* $(\boldsymbol{\nu} a)\ \mathbf{E}_1[]$ *is easy.*
- *Case* $\mathbf{E} = \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_1$. *We have* $\Gamma \vdash \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_1[P] : l$ *from which we derive* $\Gamma \vdash \mathbf{E}_1[P] : l$. *We use the induction hypothesis to conclude.*
- *Case* $\mathbf{E} = \mathbf{E}_1 \mid P_1$. *We have* $\Gamma \vdash \mathbf{E}_1[P] \mid P_1 : l$ *from which we derive* $\Gamma \vdash \mathbf{E}_1[P] : l'$ *and* $\Gamma \vdash P_1 : l_1$ *with* $l = \max(l', l_1)$. *The induction hypothesis gives* $\Gamma \vdash P : l''$ *with* $l'' \leq l'$ *as* $l' \leq l$ *and* $\Gamma \vdash \mathbf{E}_1[P_0] : l_{(1)}$ *with* $l_{(1)} \leq l'$. *We set* $l_{(0)} = \max(l_{(1)}, l_1)$ *and we conclude.*

□

The Subject Reduction proof relies on the Subject Substitution result, ensuring that substitutions induced by reductions are well-typed.

Proposition 7.1.4 (Subject Reduction) *If* $\Gamma \vdash P : l$ *and* $P \to P'$ *then* $\Gamma \vdash P' : l'$ *for some* $l' \leq l$.

Proof. *By induction on the derivation of* $P \to P'$*:*

- *Case* (**PFcall**). *We have*
$$P = \mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[\overline{f}\langle v\rangle.P_2]]$$
and
$$P' = \mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[P_1\{v/x\} \mid P_2]].$$
From $\Gamma \vdash P : l$ *and Lemma 7.1.3, we get* $\Gamma \vdash \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \overline{f}\langle v\rangle.P_2 : l''$ *from which we derive* $\Gamma(f) = \sharp^k_{\mathtt{F}}\, T$, $\Gamma(x) = \Gamma(v) = T$, $\Gamma \vdash P_1 : l_1$, $k \geq l_1$ *and* $\Gamma \vdash P_2 : l_2$. *By Lemma 7.1.2,* $\Gamma \vdash P_1\{v/x\} : l_1$*; we use rule* (**PFPar**) *to get* $\Gamma \vdash P_1\{v/x\} : \max(l_1, l_2)$. *As* $k \geq l_1$, $\max(k, l_2) \geq \max(l_1, l_2)$. *We conclude using Lemma 7.1.3.*
- *Case* (**PFtrig**). *We have* $P = \mathbf{E}[\overline{a}\langle v\rangle.P_2 \mid !a(x).P_1]$ *and* $P' = \mathbf{E}[P_2 \mid P_1\{v/x\} \mid !a(x).P_1]$. *From* $\Gamma \vdash P : l$ *we use Lemma 7.1.3 to deduce* $\Gamma \vdash \overline{a}\langle v\rangle.P_2 \mid !a(x).P_1 : l'$ *from which we get* $\Gamma \vdash P_2 : l_2$, $\Gamma \vdash P_1 : l_1$, $\Gamma(a) = \sharp^k_{\mathtt{I}}\, T$ *and* $\Gamma(v) = \Gamma(x) = T$, $k > l_1$ *and* $l' = \max(k, l_2)$. *By Lemma 7.1.2,* $\Gamma \vdash P_1\{v/x\} : l_1$ *and we use rule* (**PFPar**) *to get* $\Gamma \vdash P_2 \mid P_1\{v/x\} \mid !a(x).P_1 : \max(l_1, l_2)$. *As* $k > l_1$, $l' \geq \max(l_1, l_2)$. *We use Lemma 7.1.3 to conclude.*
- *Case* (**PFcomm**) *is similar.*
- *Case* (**PFcong**) *is treated using Lemma 7.1.1.*

□

The termination proof for $\pi^1_{\mathtt{imp}}$ uses the property that at every reduction the *multiset weight* 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.

Remark 7.1.5 *In the type system of Section 3.1, we have a strict decreasing, along every reduction step, of the* multiset weight *of* P*, which is defined as the multiset of natural numbers that are the levels of names appearing in output in* P*, where only the occurrences of unguarded outputs are taken into account. Intuitively, along the lines of rule* (**PFRep**)*, an output at level* k *is traded for possibly several outputs at levels strictly smaller than* k*. This does not hold in the present setting, as the firing of a* `def` *construct may involve the trading of an output at level* k *for one or several outputs at the same level (rule* (**PFDef**)*).*

For instance consider $\mathtt{def}\ f = (\overline{g} \mid \overline{g})\ \mathtt{in}\ \mathtt{def}\ g = \overline{a}\langle f\rangle\ \mathtt{in}\ \overline{f} \mid \overline{a}\langle g\rangle$*. In this process, the two outputs on a force the names* g *and* f *to have the same type, and thus, the same level. This process is typable using the typing rules we presented in this section. However, by looking at the first definition, we notice that we can trade an output on* f *for two outputs on* g*. Thus the measure, should we define it as usual, would increase*

strictly after a first reduction on f. Moreover, this increasing cannot be compared to the one we dealt with in section 3.2.1. In this section, the increasing of weight takes place on one particular reduction (when the last prefix of an input sequence is consumed) but is actually compensated by other reductions. Thus, we are able to group together the consequences on measure of several reductions (corresponding to the consumption of a same input sequence) to claim that this measure actually decreases over time (we do it formally by annotating the calculus and delaying the time when we compute the measure).

This is not the case in the present setting, where a functional reduction can let the weight increase and is not directly related with other reductions, preventing us from compute actual decreasing from several reductions. The termination of this system relies on the functional termination, which cannot be expressed directly as a weight, in the sense that we present it, as hinted in Section 6.1. In the example presented above, the typing and syntactical constraints prevent the definition of f to contain name f and the definition of g to contain names f and g. These constraints build an implicit hierarchy among functional names, preventing loops from arising.

7.1.3 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 7.1.9) and a related simulation result (Lemma 7.1.16), we transform the given divergence into one for a functional process in $\pi^1_{\mathtt{def}}$. This is yields contradiction, as $\pi^1_{\mathtt{def}}$ is terminating (see 6). 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 "semantic equality" after commutation). This however does not hold on $P_0 \stackrel{\text{def}}{=} !a.\overline{f} \,|\, \overline{a} \to P_1 \stackrel{\text{def}}{=} !a.\overline{f} \,|\, \overline{f}$, as the pruning of P_0 would be $\mathbf{0}$ whereas that of P_1 would be $\overline{f}$.

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 7.1.22 – the level of a reduction being the level of the name along which the reduction occurs). Further, at the cost of possibly removing an initial segment of the infinite 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 ones (`def` and outputs) on functional names 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. We thus derive an infinite computation in $\pi^1_{\mathtt{def}}$, which is impossible as $\pi^1_{\mathtt{def}}$ is terminating.

The simulation lemma 7.1.16 ensures that a functional reduction at level p in the original process is preserved by 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). Since there must be infinitely many such reductions at p in the original infinite computation (this is proved using the measure arguments for imperative names), we can conclude that pruning produces an infinite computation in $\pi^1_{\mathtt{def}}$.

Output-asynchronous π To ease the remaining proofs, we focus on processes with asynchronous outputs only: that is, output prefixes which have no continuation. The following lemmas show that we do not loose generality by doing so.

Definition 7.1.6 (Asynchronous output – Translation) *A process P has asynchronous outputs if the continuation of every output in P is $\mathbf{0}$.*

Our translation into the set of output-asynchronous processes is defined on processes and contexts:

$$\mathrm{As}(\overline{v}\langle w\rangle.P) = (\overline{v}\langle w\rangle \mid \mathrm{As}(P)) \qquad \mathrm{As}(!a(x).P) = !a(x).\mathrm{As}(P) \qquad \mathrm{As}(a(x).P) = a(x).\mathrm{As}(P)$$

$$\mathrm{As}(\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2) = \mathtt{def}\ f = (x).\mathrm{As}(P_1)\ \mathtt{in}\ \mathrm{As}(P_2)$$

$$\mathrm{As}([\,]) = [\,] \qquad \mathrm{As}(\mathbf{E} \mid P) = \mathrm{As}(\mathbf{E}) \mid \mathrm{As}(P) \qquad \mathrm{As}((\boldsymbol{\nu} a)\ \mathbf{E}) = (\boldsymbol{\nu} a)\ \mathrm{As}(P)$$

$$\mathrm{As}(\texttt{def}\ f = (x).P\ \texttt{in}\ \mathbf{E}) = \texttt{def}\ f = (x).\mathrm{As}(P)\ \texttt{in}\ \mathrm{As}(\mathbf{E})$$

One can notice that the typing rules (**PFPar**) and (**PFOut**) treat in the same way a process and its translation. This is what we state in the following Lemma.

Lemma 7.1.7 (Asynchronous output – Typability)
If $\Gamma \vdash P : l$ *then* $\Gamma \vdash \mathrm{As}(P) : l$.

Proof. *By induction on the typing judgement* $\Gamma \vdash P : l$,

- *Case* (**PFOut**). *We have* $\Gamma \vdash \overline{v}\langle w\rangle.P_1 : l$ *from which we derive* $\Gamma \vdash P_1 : l_1$, $\Gamma(v) = \sharp^k_\bullet\, T$ *and* $l = \max(k, l_1)$. *Rules* (**PFNil**) *and* (**PFOut**) *give* $\Gamma \vdash \overline{v}\langle w\rangle : k$. *The induction hypothesis gives* $\Gamma \vdash \mathrm{As}(P) : l_1$. *We conclude using rule* (**PFPar**) *to get* $\Gamma \vdash \overline{v}\langle w\rangle \mid \mathrm{As}(P) : \max(k, l_1)$ *which is* $\Gamma \vdash \mathrm{As}(P) : l$.
- *Case* (**PFDef**). *We have* $\Gamma \vdash \texttt{def}\ f = (x).P_1\ \texttt{in}\ P_2 :$ *from which we deduce* $\Gamma \vdash P_1 : l_1$, $\Gamma \vdash P_2 : l$, $\Gamma(f) = \sharp^k_1\, T$ *and* $k \geq l_1$. *We use the induction hypothesis to get* $\Gamma \vdash \mathrm{As}(P_1) : l_1$ *and* $\Gamma \vdash \mathrm{As}(P_2) : l$ *and we conclude using rule* (**PFDef**).
- *Cases* (**PFIn**) *and* (**PFRep**) *are similar and other cases are easy.*

□

Then, we prove that $\mathrm{As}(P)$ offers at least as much reduction possibilities as P. Thus, termination of $\mathrm{As}(P)$ implies termination of P.

Lemma 7.1.8 (Asynchronous output – Simulation) *The relation* $\leq^{\mathrm{as}}$ *defined by* $P \leq^{\mathrm{as}} Q$ *if* $Q = \mathrm{As}(P)$ *is a simulation.*

$\leq^{\mathrm{as}}$ is a simulation means "If $P \leq^{\mathrm{as}} Q$, for all P', $P \to P'$ there exists Q', $Q \to Q'$ and $P' \leq^{\mathrm{as}} Q'$.
Proof. *By induction on the derivation of* $P \to P'$.

- *Case* (**PFcall**). *We have* $P = \mathbf{E}_1[\texttt{def}\ f = (x).P_1\ \texttt{in}\ \mathbf{E}_2[\overline{f}\langle v\rangle.P_2]]$. *It is easy to see that* $Q = \mathrm{As}(P) = \mathrm{As}(\mathbf{E}_1)[\texttt{def}\ f = (x).\mathrm{As}(P_1)\ \texttt{in}\ \mathrm{As}(\mathbf{E}_2)[\overline{f}\langle v\rangle \mid \mathrm{As}(P_2)]]$. *We set* $\mathbf{E}_3 = \mathbf{E}_2[] \mid P_2$. *Thus, we have* $Q = \mathrm{As}(\mathbf{E}_1)[\texttt{def}\ f = (x).\mathrm{As}(P_1)\ \texttt{in}\ \mathrm{As}(\mathbf{E}_3)[\overline{f}\langle v\rangle]]$ *and by rule* (**PFcall**), $Q \to Q' = \mathrm{As}(\mathbf{E}_1)[\texttt{def}\ f = (x).\mathrm{As}(P_1)\ \texttt{in}\ \mathrm{As}(\mathbf{E}_3)[\mathrm{As}(P_1)\{v/x\}]]$. *We notice that* $Q' = \mathrm{As}(\mathbf{E}_1[\texttt{def}\ f = (x).P_1\ \texttt{in}\ \mathbf{E}_2[P_1\{v/x\} \mid P_2]])$ *and we conclude.*
- *Case* (**PFtrig**). *We have* $P = \mathbf{E}[\overline{a}\langle v\rangle.P_2 \mid\ !a(x).P_1]$ *and* $P' = P_1\{v/x\} \mid P_2 \mid\ !a(x).P_1$. *Moreover we have* $Q = \mathrm{As}(P) = \mathrm{As}(\mathbf{E})[\overline{a}\langle v\rangle \mid \mathrm{As}(P_2) \mid\ !a(x).\mathrm{As}(P_1)]$. *We set* $\mathbf{E}_1 = \mathbf{E}[[] \mid P_2]$ *and notice that* $Q = \mathrm{As}(\mathbf{E}_1)[\overline{a}\langle v\rangle \mid\ !a(x).P_1]$ *we use rule* (**PFtrig**) *to get* $Q \to Q' = \mathrm{As}(\mathbf{E}_1)[P_1\{v/x\} \mid\ !a(x).P_1] = \mathrm{As}(P')$. *We conclude.*
- *Case* (**PFcomm**) *is similar.*
- *Case* (**PFcong**) *is done by proving that if* $P \equiv Q$, $\mathrm{As}(P) \equiv \mathrm{As}(Q)$, *by induction on the* $\equiv$ *definition.*

□

As a consequence, we can put the focus on output-asynchronous processes in the remaining of the soundness proof. Suppose indeed, that we have soundness for output-asynchronous processes (every typable output asynchronous process terminates) and a diverging process P with $\Gamma \vdash P : l$. By Lemma 7.1.7 we have $\Gamma \vdash \mathrm{As}(P) : l$ and by Lemma 7.1.8, $\mathrm{As}(P)$ diverges. As $\mathrm{As}(P)$ has asynchronous outputs, we get a contradiction.

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

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

$$\mathsf{pr}^p_\Gamma((\boldsymbol{\nu} a)\, P) \stackrel{\text{def}}{=} \mathsf{pr}^p_\Gamma(P)$$

$$\mathsf{pr}^p_\Gamma(\overline{a}\langle v\rangle) \stackrel{\text{def}}{=} \mathbf{0}$$

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

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

$$\mathsf{pr}^p_\Gamma(\overline{f}^n\langle v\rangle) = \begin{cases} \overline{f}^n\langle v\rangle & \text{if } n = p \\ \mathbf{0} & \text{otherwise} \end{cases}$$

Figure 7.3: Pruning in $\pi^1_{\mathtt{ST}}$

Pruning In the following part, we introduce, as stated previously, a pruning operator, parametric w.r.t. a level p. The operator $\mathsf{pr}^p_\Gamma()$ removes the imperative parts of a process, as well as the functional parts at levels different from p, leaving the functional backbone at level p of the process.

Definition 7.1.9 (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 7.3, where a (resp. f^n) is a name whose type in Γ is imperative (resp. functional with level n).
Pruning is extended to contexts with $\mathsf{pr}^p_\Gamma([\,]) = [\,]$

Notice that pruning is defined on typed terms only, as we discuss the level of names. The following facts ensure that the pruning behaves as expected w.r.t. to evaluation contexts, well-typed substitutions and typability.

Fact 7.1.10 (Context pruning)
For all $\mathbf{E}$, $\mathsf{pr}^p_\Gamma(\mathbf{E})$ is a simply-typed context and for all adequate P, $\mathsf{pr}^p_\Gamma(\mathbf{E}[P]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(P)]$.

Proof.
Easily done by induction on the evaluation context.

□

Fact 7.1.11 (Substitution pruning)
If $\Gamma(x) = \Gamma(v)$, $\mathsf{pr}^p_\Gamma(P)\{v/x\} = \mathsf{pr}^p_\Gamma(P\{v/x\})$

Proof. *Easily done by induction on the typing judgement.* □

We prove in the following that if P is in $\pi^1_{\mathtt{ST}}$, then $\mathsf{pr}^p_\Gamma(P)$ is in $\pi^1_{\mathtt{def}}$, and its typing is obtained from the one of P simply by stripping off the levels. We write $\mathrm{PR}(\Gamma)$ for the typing of the simply-typed π-calculus resulting 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.

Fact 7.1.12 (Pruning – Typability)
Suppose $\Gamma \vdash P : l$. Then, for any p, $\vdash^\pi_{\mathrm{PR}(\Gamma)} \mathsf{pr}^p_\Gamma(P)$.

Proof. *By induction on the typing judgement:*

- *Cases* (**PFIn**), (**PFRep**), (**PFNil**) *are easy, as $\mathsf{pr}^p_\Gamma(P) = \mathbf{0}$.*
- *Cases* (**PFRes**) *and* (**PFPar**) *are treated using the induction hypothesis.*

- *Case* (**PFOut**). *We have* $P = \overline{v}\langle w\rangle$. *We discuss the type of* v*:*
  - *Either* $\Gamma(v) = \sharp^k_{\mathrm{I}}\, T$ *and* $\Gamma(w) = T$, *then* $\mathsf{pr}^p_\Gamma(P) = \mathbf{0}$. *We conclude.*
  - *Or* $\Gamma(v) = \sharp^k_{\mathrm{I}}\, T$ *with* $k \neq p$ *and* $\Gamma(w) = T$, *then* $\mathsf{pr}^p_\Gamma(P) = \mathbf{0}$. *We conclude.*
  - *Or* $\Gamma(v) = \sharp^p_{\mathrm{F}}\, T$ *and* $\Gamma(w) = T$, *then* $\mathsf{pr}^p_\Gamma(P) = \overline{v}\langle w\rangle$ *with* $\mathrm{PR}(\Gamma)(v) = \mathrm{PR}(\sharp^p_{\mathrm{F}}\, T) = \sharp\, \mathrm{PR}(T)$ *and* $\mathrm{PR}(\Gamma)(w) = \mathrm{PR}(T)$.
- *Case* (**PFDef**). *We have* $P = (\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2)$ *and we discuss the type of* f.
  - *Either* $\Gamma(f) = \sharp^k_{\mathrm{F}} T$ *with* $k \neq p$ *and* $\mathsf{pr}^p_\Gamma(P) = \mathsf{pr}^p_\Gamma(P_2)$. *We conclude using the induction hypothesis.*
  - *Or* $\Gamma(f) = \sharp^p_{\mathrm{F}}\, T$ *and* $\Gamma(x) = T$, *then* $\mathsf{pr}^p_\Gamma(P) = (\mathtt{def}\ f = (x).\mathsf{pr}^p_\Gamma(P_1)\ \mathtt{in}\ \mathsf{pr}^p_\Gamma(P_2))$. *With the induction hypothesis, knowing that* $\mathrm{PR}(\Gamma)(f) = \mathrm{PR}(\sharp^p_{\mathrm{F}}\, T) = \sharp\, \mathrm{PR}(T)$ *and* $\mathrm{PR}(\Gamma)(x) = \mathrm{PR}(T)$, *we conclude.*

□

We write $\simeq$ for *strong barbed congruence*, defined in the usual way [SW01]. $\simeq$ is needed in the statement of the following lemmas. The essential properties we need is that $\simeq$ preserves divergence and is closed by static contexts.

We write in the following $P \to^n_{\mathrm{I}} P'$ (resp. $P \to^n_{\mathrm{F}} P'$) if P has a reduction to P' in which the interacting name is imperative (resp. functional) and of level n.

A delicate aspect of the proof of Lemma 7.1.16 is that, if, for instance, $!a(x).P$ is involved in a reduction, an instantiated copy of P is unleashed at top-level; we establish that the pruning of such process is semantically $\mathbf{0}$ (which is not immediate, since P may contain top-level definitions at level p). Then, this allows us to derive the properties stated below (a similar reasoning is made for the pruning of processes obtained via a definition on a level different of p).

For instance, consider the process $!a(x).\mathtt{def}\ f = P_1\ \mathtt{in}\ P_2$ in a typing context associating a with level 3 and f with level 4. Suppose we want to compute pruning at level 4 at this process. The pruning at level 4, as defined above, maps the whole replicated process into $\mathbf{0}$.

This seems fair, as this process cannot be used to feed a divergence at level 4. However, we notice that by doing so, we remove completely the definition on f, which has level 4. We have to justify that f, even if it has level 4, cannot be used to perform reductions on this particular level.

Indeed, we are able to prove this. As a has level 3, the rule (**PFDef**) ensures that no output of level greater than 2 can be found inside P_2. This means that P_2 cannot contain outputs on f and, moreover, than the pruning at level 4 of P_2 cannot contains outputs, and thus, no way of extruding name f. As a result, the definition of f is dead-code in the pruned process and is behaviourally equivalent to $\mathbf{0}$. The following results formally prove what we hinted here.

We first notice that a definition is useless if its second component is $\mathbf{0}$. Indeed, the name of the definition cannot be directly triggered and the defined name cannot be extruded.

Fact 7.1.13 (Inactivity of definition)
We have $(\mathtt{def}\ f = (x).P\ \mathtt{in}\ \mathbf{0}) \simeq \mathbf{0}$.

Proof.
As $\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2$ *binds the name* f, *the process* $(\mathtt{def}\ f = (x).P\ \mathtt{in}\ \mathbf{0})$ *offers no visible interaction.*
□

We prove that the pruning of the continuation process of inputs or definitions on names we prune, is inactive.

Lemma 7.1.14 (Pruning – Inactivity)

1. *If* $\Gamma \vdash \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2 : n$, *with* $\Gamma(f) = \sharp^n_{\mathrm{F}}\, T$ *and* $n < p$, *then for any* v *s.t.* $\Gamma(v) = T$, $\mathsf{pr}^p_\Gamma(P_1)\{v/x\} \simeq \mathbf{0}$.

2. *If* $\Gamma \vdash !a(x).P_1 : n$, *with* $\Gamma(a) = \sharp^n_{\mathtt{F}}\, T$ *and* $n \leq p$, *then for any* v *s.t.* $\Gamma(v) = T$, $\mathsf{pr}^p_\Gamma(P_1)\{v/x\} \simeq \mathbf{0}$.

3. *If* $\Gamma \vdash a(x).P_1 : n$, *with* $\Gamma(a) = \sharp^n_{\mathtt{F}}\, T$ *and* $n \leq p$, *then for any* v *s.t.* $\Gamma(v) = T$, $\mathsf{pr}^p_\Gamma(P_1)\{v/x\} \simeq \mathbf{0}$.

Proof. *We prove the following:*
"For each process P_1, *for every* p, *for every functional name* f *and every imperative name* a,*:*

1. *if* $\Gamma \vdash \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2 : l_2$ *with* $\Gamma(u) = \sharp^k_{\mathtt{F}}\, T'$, $k < p$ *and* $\Gamma(v) = T'$, *then* $\mathsf{pr}^p_\Gamma(P_1)\{v/x\} \simeq \mathbf{0}$

2. *if* $\Gamma \vdash !a(x).P_1 : 0$ *with* $\Gamma'(a) = \sharp^k_{\mathtt{I}}\, T'$, $k \leq p$ *and* $\Gamma(v) = T'$, *then* $\mathsf{pr}^p_\Gamma(P_1)\{v/x/\simeq\}\mathbf{0}$

3. *and if* $\Gamma \vdash a(x).P_1 : 0$ *with* $\Gamma'(a) = \sharp^k_{\mathtt{I}}\, T'$, $k \leq p$ *and* $\Gamma(v) = T'$, *then* $\mathsf{pr}^p_\Gamma(P_1)\{v/x\} \simeq \mathbf{0}$"

We reason by induction on the typing judgement $\Gamma \vdash P_1 : l$.

- *Case* (**PFDef**). *We have* $P_1 = \mathtt{def}\ g = (y).P_3\ \mathtt{in}\ P_4$. *Suppose that* $\Gamma \vdash \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2 : l_2$. *We easily derive* $\Gamma \vdash \mathtt{def}\ f = (x).P_4\ \mathtt{in}\ P_2 : 0$ *using rules* (**PFRes**) *and* (**PFDef**) *rules. Thus, by induction,* $\mathsf{pr}^p_\Gamma(P_4)\{v/x\} \simeq \mathbf{0}$. *Now:*
  - *Either* $\mathtt{lvl}(g) < p$. *We have* $\mathsf{pr}^p_\Gamma(\mathtt{def}\ f' = (y).P_3\ \mathtt{in}\ P_4)\{v/x\} = \mathsf{pr}^p_\Gamma(P_4)$. *We conclude.*
  - *Or* $\mathtt{lvl}(g) \geq p$. *And* $\mathsf{pr}^p_\Gamma(\mathtt{def}\ f' = (y).P_3\ \mathtt{in}\ P_4)\{v/x\} = (\mathtt{def}\ f' = (y).\mathsf{pr}^p_\Gamma(P_3)\{v/x\}\ \mathtt{in}\ \mathsf{pr}^p_\Gamma(P_4)\{v/x\}) \simeq (\mathtt{def}\ f' = (y).\mathsf{pr}^p_\Gamma(P_3)\ \mathtt{in}\ \mathbf{0}) \simeq \mathbf{0}$, *by Fact 7.1.13.*

  The proofs that the case (**PFDef**) *holds for 2. and 3. are very similar.*

- *Case* (**PFOut**). *We have* $P_1 = \overline{v'}\langle w\rangle$. *Suppose we have* $\Gamma \vdash \mathtt{def}\ f = (x).\overline{v'}\langle w\rangle\ \mathtt{in}\ P_2 : l_2$ *with* $\mathtt{lvl}(f) < p$
  - *Either* $\mathtt{lvl}(v') < K$. *Then* $\mathsf{pr}^p_\Gamma(P_1)\{v/x\} = \mathbf{0}$ *and we conclude.*
  - *Or* $\mathtt{lvl}(v') \geq p$. *This contradicts the fact that* $\mathtt{def}\ f = (x).\overline{v'}\langle w\rangle\ \mathtt{in}\ P_2$ *is typable.*

  The proofs that the case (**PFOut**) *holds for 2. and 3. are very similar.*

- *Case* (**PFPar**). *We have* $P_1 = P_3 \mid P_4$. *Suppose* $\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2$ *is typable, then* $\mathtt{def}\ f = (x).P_3\ \mathtt{in}\ P_2$ *and* $\mathtt{def}\ f = (x).P_4\ \mathtt{in}\ P_2$ *are typable too. By the induction hypothesis, for an adequate* v, $\mathsf{pr}^p_\Gamma(P_3)\{v/x\} \simeq \mathbf{0}$ *and* $\mathsf{pr}^p_\Gamma(P_4)\{v/x\} \simeq \mathbf{0}$. *The proofs that the case* (**PFOut**) *holds for 2. and 3. are very similar.*

- *Case* (**PFRep**). *We have* $P_1 = !b(y).P_3$ *and for any* p, $\mathsf{pr}^p_\Gamma(!b(y).P_3)\{v/x\} = \mathbf{0}$.

- *Case* (**PFIn**) *is similar to case* (**PFRep**).

- *Other cases are easy.*

□

The following property will be useful to prove simulation, when the rule (**PFcong**) is invoked.

Lemma 7.1.15 (Pruning Congruence)
If $P \equiv Q$, *then* $\mathsf{pr}^p_\Gamma(P) \equiv \mathsf{pr}^p_\Gamma(Q)$.

Proof.
By induction on the derivation of $P \equiv Q$.

□

The simulation lemma is the crux of our proof technique. We relate the annotated reductions of the typechecked terms to the reductions of the associated pruned terms. If the original term P performs a functional reduction on level p to P', then, not only the pruning at level p of P, $\mathsf{pr}^p_\Gamma(P)$ is also able to

perform a reduction, but more precisely, it can reduce to $\mathsf{pr}^p_\Gamma(P')$. However, if P performs either an imperative reduction, or a functional reduction of level $< p$ to P', then the pruning of P and P' are related by $\simeq$ (if not equal). We said above that pruning at level p computes the functional backbone at level p of a process. We show here that the pruned process simulates exactly the reductions at level p of the original process. In other words, the useful property we are able to derive is that if there exists an infinite computation starting from a typable process P which contains an infinite number of functional reduction steps at level p and no reductions at level greater that p, then there exists an infinite reduction sequence starting from the pruning of the initial process $\mathsf{pr}^p_\Gamma(P)$. This crucial result will allow us to raise a contradiction from the existence of a diverging typable process.

Lemma 7.1.16 (Simulation)
Suppose $\Gamma \vdash P : l$.

1. *If* $P \rightarrow^p_{\mathrm{F}} P'$, *then* $\mathsf{pr}^p_\Gamma(P) \rightarrow \mathsf{pr}^p_\Gamma(P')$;
2. *If* $P \rightarrow^n_{\mathrm{F}} P'$ *with* $n < p$, *then* $\mathsf{pr}^p_\Gamma(P) \simeq \mathsf{pr}^p_\Gamma(P')$;
3. *If* $P \rightarrow^n_{\mathrm{I}} P'$ *with* $n \leq p$, *then* $\mathsf{pr}^p_\Gamma(P) \simeq \mathsf{pr}^p_\Gamma(P')$.

Proof.

1. *By induction on the derivation* $P \rightarrow^p_{\mathrm{F}} P'$:
   - *Case* (**PFDef**). *We have*
   $$P = \mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[\overline{f}\langle v\rangle]]$$
   and
   $$P' = \mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[P_1\{v/x\}]].$$
   We know that f *has level* p. *Thus, by Fact 7.1.10,* $\mathsf{pr}^p_\Gamma(\mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[\overline{f}\langle v\rangle]]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{def}\ f = (x).\mathsf{pr}^p_\Gamma(P_1)\ \mathtt{in}\ \mathsf{pr}^p_\Gamma(\mathbf{E}_2)[\overline{f}\langle v\rangle]]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[P_1\{v/x\}]]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{def}\ f = (x).\mathsf{pr}^p_\Gamma(P_1)\ \mathtt{in}\ \mathsf{pr}^p_\Gamma(\mathbf{E}_2)[\mathsf{pr}^p_\Gamma(P_1\{v/x\})]]$. *We conclude, using Fact 7.1.11.*
   - *Case* (**PFcong**). *We use Lemma 7.1.15 and the induction hypothesis.*
2. *By induction on the derivation* $P \rightarrow^n_{\mathrm{F}} P'$.
   - *Case* (**PFcall**). *We have*
   $$P = \mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[\overline{f}\langle v\rangle]]$$
   and
   $$P' = \mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[P_1\{v/x\}]].$$
   We know that f *has level* $n < p$. *Thus, by Fact 7.1.10,*
   $$\mathsf{pr}^p_\Gamma(\mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[\overline{f}\langle v\rangle]]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[\mathbf{0}]]$$
   and
   $$\mathsf{pr}^p_\Gamma(\mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[P_1\{v/x\}]]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[\mathsf{pr}^p_\Gamma(P_1)\{v/x\}]]$$
   . *As* v *and* x *have same type and* f *has level* $n < p$, *we use Lemma 7.1.14 to state that* $\mathsf{pr}^p_\Gamma(P_1)\{v/x\} \simeq \mathbf{0}$. *We conclude, as* $\simeq$ *is a congruence.*
   - *Case* (**PFcong**). *We use Lemma 7.1.15 and the induction hypothesis.*
3. *By induction on the derivation* $P \rightarrow^n_{\mathrm{I}} P'$:

- *Case* (**PFtrig**) *We have*
$$P = \mathbf{E}_1[\overline{a}\langle v\rangle \mid !a(x).P_1]$$
and
$$P' = \mathbf{E}_1[P_1\{v/x\} \mid !a(x).P_1].$$
We have by Fact 7.1.10 $\mathsf{pr}^p_\Gamma(\mathbf{E}_1[\overline{a}\langle v\rangle \mid !a(x).P_1]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathbf{0}]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}_1[P_1\{v/x\} \mid !a(x).P_1]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathsf{pr}^p_\Gamma(P_1\{v/x\})]$. *As v and x have same type, we use Lemma 7.1.14 to state that* $\mathsf{pr}^p_\Gamma(P_1\{v/x\}) \simeq \mathbf{0}$. *We conclude, as $\simeq$ is congruence.*
- *Case* (**PFcomm**) *is similar.*
- *Case* (**PFcong**). *We use Lemma 7.1.15 and the induction hypothesis.*

□

We now turn to the main body of the termination proof. As announced, we reason by absurd, and extract from a diverging computation involving well-typed processes a computation where infinitely many functional reductions take place.

First, we use the weight system to state that the number of active outputs on a given level decreases if no functional reduction at level p takes place.

Definition 7.1.17 (Active outputs)
The number of active outputs at level p *in P is defined on typed processes as follows (as usual, we write v^k to denote that $\Gamma(v) = \sharp^k_\bullet T$ for some T):*

$$\mathbf{Os}^p(\mathbf{0}) = 0 \qquad \mathbf{Os}^p(P_1 \mid P_2) = \mathbf{Os}^p(P_1) + \mathbf{Os}^p(P_2) \qquad \mathbf{Os}^p(!a(x).P) = \mathbf{Os}^p(a(x).P) = 0$$

$$\mathbf{Os}^p(\texttt{def } f = (x).P_1 \texttt{ in } P_2) = \mathbf{Os}^p(P_2) \qquad \mathbf{Os}^p((\boldsymbol{\nu} a)\ P) = \mathbf{Os}^p(P) \qquad \begin{array}{l}\mathbf{Os}^p(\overline{v}^k\langle w\rangle) = 0 \quad \text{if } k \neq p\\ \mathbf{Os}^p(\overline{v}^p\langle w\rangle) = 1\end{array}$$

We extend this definition to evaluation contexts by $\mathbf{Os}^p([\,]) = 0$.

Fact 7.1.18 (Active outputs in contexts)
We have $\mathbf{Os}^p(\mathbf{E}[P]) = \mathbf{Os}^p(\mathbf{E}) + \mathbf{Os}^p(P)$

Proof.
By structural induction on contexts.

□

Notice that Fact 7.1.18 holds because holes cannot be placed inside definitions nor under imperative inputs. Thus every output appearing active in the hole is active in the whole process.

Fact 7.1.19 (Active outputs congruence)
If $P \equiv Q$ then $\mathbf{Os}^p(P) = \mathbf{Os}^p(Q)$

Proof. *By induction on the derivation of $P \equiv Q$, using the associativity and commutativity of $+$ and the neutrality of 0 for $+$.* □

The following lemma states that the weight of a process P corresponds to the maximum level of an active output inside P.

Lemma 7.1.20 (Weight and active outputs)
If $\Gamma \vdash P : l$ and $l < p$ then $\mathbf{Os}^p(P) = 0$.

Proof. *By induction on the typing judgment $\Gamma \vdash P : l$,*

- *Case* (**PFPar**). *We have* $P = P_1 \mid P_2$. *We derive* $\Gamma \vdash P_1 : l_1$, $\Gamma \vdash P_2 : l_2$ *and* $l = \max(l_1, l_2)$. *As* $l < K$, $l_1 < K$ *and* $l_2 < K$, *thus we use the induction hypothesis to get* $\mathbf{Os}^K(P_1) = \mathbf{Os}^K(P_2) = 0$. *As* $\mathbf{Os}^p(P) = \mathbf{Os}^p(P_1) + \mathbf{Os}^K(P_2)$, *we conclude.*
- *Case* (**PFOut**). *We have* $P = \overline{v}\langle w\rangle$ *from which we deduce* $\Gamma(v) = \sharp^l_\bullet\, T$. *As* $l < K$, $\mathbf{Os}^K(P) = 0$.
- *Case* (**PFDef**). *We have* $P = \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2$ *from which we deduce* $\Gamma \vdash P_2 : l$. *As* $\mathbf{Os}^p(P) = \mathbf{Os}^p(P_2)$, *we use the induction hypothesis to conclude.*
- *Case* (**PFRep**). *We have* $P = !a(x).P_1$, $l = 0$ *and* $\mathbf{Os}^p(P) = 0$.
- *Case* (**PFIn**) *is similar and other cases are easy.*

□

We are now able to state the measure-decreasing lemma, proving that the imperative reductions alone cannot let a process diverge. Indeed, the well-founded measure $\mathbf{Os}^p(P)$ decreases with each imperative reduction at level p and cannot increase with reductions at level strictly smaller that p. The result holds as our type system prevents a reduction at level $< p$ from releasing active outputs at level p. The fact that our level-based type system is extended to the functional part of the calculus is crucial here, as we have to state that the functional reductions behave as expected with the measure.

Lemma 7.1.21 (Evolution of active outputs)
If $\Gamma \vdash P : l$ *then:*

1. *if* $P \rightarrow^n_{\mathrm{F}} P'$ *with* $n < p$ *then* $\mathbf{Os}^p(P') \leq \mathbf{Os}^p(P)$.
2. *if* $P \rightarrow^n_{\mathrm{I}} P'$ *with* $n < p$ *then* $\mathbf{Os}^p(P') \leq \mathbf{Os}^p(P)$.
3. *if* $P \rightarrow^p_{\mathrm{I}} P'$ *then* $\mathbf{Os}^p(P') < \mathbf{Os}^p(P)$.

Proof.

1. *By induction on the derivation of* $P \rightarrow^n_{\mathrm{F}} P'$.
   - *Case* (**PFcong**) *is treated by Fact 7.1.19 and the induction hypothesis.*
   - *Case* (**PFcall**). *We have*
     $$P = \mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[\overline{f}\langle v\rangle]]$$
     and
     $$P' = \mathbf{E}_1[\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ \mathbf{E}_2[P_1\{v/x\}]]$$
     . *From* $\Gamma \vdash P : l$, *we deduce,* $\Gamma \vdash P_1 : l_1$, $\Gamma(f) = \sharp^n_{\mathrm{F}}\, T$ *and* $n \geq l_1$. *From Lemma 7.1.2, we get* $\Gamma \vdash P_1\{v/x\} : l_1$. *As* $p > n$, *we have* $\mathbf{Os}^K(\overline{f}\langle v\rangle) = 0$ *and as* $n \geq l_1$, *we use Lemma 7.1.20 to get* $\mathbf{Os}^p(P_1\{v/x\}) = 0$. *The definition of* $\mathbf{Os}^p()$ *gives* $\mathbf{Os}^K(P) = \mathbf{Os}^p(\mathbf{E}_1) + \mathbf{Os}^p(\mathbf{E}_2) + \mathbf{Os}^p(\overline{f}\langle v\rangle)$ *and* $\mathbf{Os}^K(P') = \mathbf{Os}^p(\mathbf{E}_1) + \mathbf{Os}^K(\mathbf{E}_2) + \mathbf{Os}^p(P_1\{v/x\})$. *We conclude.*
2. *By induction on the derivation of* $P \rightarrow^n_{\mathrm{I}} P'$.
   - *Case* (**PFcong**) *is treated by Fact 7.1.19 and the induction hypothesis.*
   - *Case* (**PFtrig**). *We have* $P = \mathbf{E}[!a(x).P_1 \mid \overline{a}\langle v\rangle]$ *and* $P' = \mathbf{E}[!a(x).P_1 \mid P_1\{v/x\}]$. *From* $\Gamma \vdash P : l$, *we deduce, among other judgements,* $\Gamma \vdash P_1 : l_1$, $\Gamma(a) = \sharp^n_{\mathrm{I}}\, T$ *and* $n > l_1$. *From Lemma 7.1.2, we get* $\Gamma \vdash P_1\{v/x\} : l_1$. *As* $p > n$, *we have* $\mathbf{Os}^K(\overline{a}\langle v\rangle) = 0$ *and as* $n > l_1$, *we use Lemma 7.1.20 to get* $\mathbf{Os}^p(P_1\{v/x\}) = 0$. *The definition of* $\mathbf{Os}^p()$ *gives* $\mathbf{Os}^K(P) = \mathbf{Os}^p(\mathbf{E}) + \mathbf{Os}^p(\overline{a}\langle v\rangle)$ *and* $\mathbf{Os}^K(P') = \mathbf{Os}^p(\mathbf{E}) + \mathbf{Os}^p(P_1\{v/x\})$. *We conclude.*
   - *Case* (**PFcomm**) *is similar.*

3. *By induction on the derivation of* $P \to^p_{\mathrm{I}} P'$,

   - *Case* (**PFcong**) *is treated by Fact 7.1.19 and the induction hypothesis.*
   - *Case* (**PFtrig**). *We have* $P = \mathbf{E}[!a(x).P_1 \mid \overline{a}\langle v\rangle]$ *and* $P' = \mathbf{E}[!a(x).P_1 \mid P_1\{v/x\}]$. *From* $\Gamma \vdash P : l$, *we deduce, among other judgements,* $\Gamma \vdash P_1 : l_1$, $\Gamma(a) = \sharp^p_{\mathrm{I}}\, T$ *and* $p > l_1$. *From Lemma 7.1.2, we get* $\Gamma \vdash P_1\{v/x\} : l_1$. *We have* $\mathbf{Os}^K(\overline{a}\langle v\rangle) = 1$ *and as* $p > l_1$, *we use Lemma 7.1.20 to get* $\mathbf{Os}^p(P_1\{v/x\}) = 0$. *The definition of* $\mathbf{Os}^p()$ *gives* $\mathbf{Os}^K(P) = \mathbf{Os}^p(\mathbf{E}) + \mathbf{Os}^p(\overline{a}\langle v\rangle)$ *and* $\mathbf{Os}^K(P') = \mathbf{Os}^p(\mathbf{E}) + \mathbf{Os}^p(P_1\{v/x\})$. *We conclude.*
   - *Case* (**PFcomm**) *is similar.*

□

Remember that both the measure and the pruning were defined for a given level p. We hinted above that p should be the maximum level on which an infinite number of reductions take place in a given infinite reduction sequence from a typable process. The following lemma states this explicitly, and proves the existence of such a level, called the *maximum interesting level*. Not only does it state that for every infinite reduction sequence, there exists a level p which is "used" an infinite number of times, it also states that there is an infinite number of functional reductions on level p in this sequence. Indeed, Lemma 7.1.21 ensures that the number of outputs at level p decreases at each imperative reduction at level p and cannot increase with reductions at levels strictly smaller than p. As p is maximal, there is a point in the infinite sequence such that, beyond this point, no reduction on a level higher than p takes place. Thus, the only justification to the presence of an infinite number of decreasings of $\mathbf{Os}^p(P_i)$ is the presence an infinite number of functional reductions at level p.

Lemma 7.1.22 (Maximum interesting level)
Suppose that $\Gamma \vdash P : l$, *and that there exists* $(P_i)_{i\in\mathbb{N}}$, *an infinite computation starting from* P, *then:*

1. *for all* i, P_i *is typable.*
2. *there exists* p, i_0 *and an infinite set* $\mathcal{I}$ *of indexes such that*
   (a) *if* $i > i_0$, *and* $P_i \to^n_{\mathrm{I}} P_{i+1}$ *then* $n \leq p$;
   (b) *if* $i > i_0$, *and* $P_i \to^n_{\mathrm{F}} P_{i+1}$ *then* $n \leq p$;
   (c) *for any* $i \in \mathcal{I}$, *either* $P_i \to^p_{\mathrm{I}} P_{i+1}$ *or* $P_i \to^p_{\mathrm{F}} P_{i+1}$.
   (d) *There are infinitely many* $i \in \mathcal{I}$ *such that* $P_i \to^p_{\mathrm{F}} P_{i+1}$.

Proof.

1. *Given by Proposition 7.1.4.*
2. *It is easy to find* p *satisfying 2a, 2b and 2c as the set of levels on which an infinite number of reductions take place is finite (there is a finite number of names in* P, *thus a finite number of levels involved and it is easy to see that new names appearing along reduction are mapped to levels already existing in types used in the derivation of* $\Gamma \vdash P : l$*) and thus, admits a maximum. Lemma 7.1.21 ensures that 2d holds. Consider such a* p *and suppose, toward a contradiction, that 2d does not hold, that is, there exists an index* j *s.t. for every* $i > j$, *either* $P_i \to^n_{\mathrm{F}} P_{i+1}$ *with* $n < p$, *or* $P_i \to^n_{\mathrm{I}} P_{i+1}$ *with* $n < p$, *or* $P_i \to^p_{\mathrm{I}} P_{i+1}$. *We use Lemma 7.1.21 to deduce that the sequence* $(\mathbf{Os}^p(P_i))_{i>j}$ *is decreasing. Moreover, as 2c holds, there is an infinite number of* i *s.t.* $P_i \to^p_{\mathrm{I}} P_{i+1}$. *Thus, we use Lemma 7.1.21 to state that the sequence* $(\mathbf{Os}^p(P_i))_{i>j}$ *strictly decreases an infinite number of times. Contradiction, as* $>$ *is well-founded.*

□

We state that when we prune a well-typed process, the resulting process is functional, according to the definition we gave.

$$(\mathbf{Def-2})\ \frac{\Gamma, x:T \vdash P_1 : l \qquad \Gamma \vdash P_2 : l' \qquad \vdash f : \sharp_{\mathtt{F}}^{k} T \qquad k \geq l}{\Gamma \vdash \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2 : l'}$$

Figure 7.4: Relaxed typing rule for Definition

Fact 7.1.23
If $\Gamma \vdash P : l$, then $\mathsf{pr}^p_\Gamma(P) \in \pi_{\mathtt{def}}$.

Proof.
Given by Fact 7.1.12, noticing that the typing rule for definition ensures that in $\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2$, f does not appear inside P_1.

□

Putting together the results above, we are able to get soundness. From Lemmas 7.1.22 and 7.1.16 and Proposition 7.1.4, we are able to transform an infinite computation starting from a well-typed process into an infinite computation of pruned terms.

Theorem 7.1.24 ([San06])
If $P \in \pi_{\mathtt{def}}$ and $\vdash P : l$ for some l, then P terminates.

We conclude the proof:

Theorem 7.1.25 (Soundness)
Take P in $\pi^1_{\mathtt{ST}}$.
If $\Gamma \vdash P : l$, then P is terminating.

Proof.
Consider, by absurd, an infinite computation $\{P_i\}_i$ starting from $P = P_0$. By Lemma 7.1.22, all the P_i's are well-typed, and there is a maximal p s.t. for infinitely many i, $P_i \rightarrow^p_{\mathtt{F}} P_{i+1}$. Moreover, there exists an index i_0 s.t. every reduction from a process P_j with j greater than i_0 is performed at level $n \leq p$. Consider the sequence $(\mathsf{pr}^p_\Gamma(P_i))_{i>i_0}$. By Lemma 7.1.16, we obtain that for every $i > i_0$, either $\mathsf{pr}^p_\Gamma(P_i) \rightarrow \mathsf{pr}^p_\Gamma(P_{i+1})$ or $\mathsf{pr}^p_\Gamma(P_i) \simeq \mathsf{pr}^p_\Gamma(P_{i+1})$. Moreover, $\mathsf{pr}^p_\Gamma(P_i) \rightarrow \mathsf{pr}^p_\Gamma(P_{i+1})$ for an infinite number of i. Thus $\mathsf{pr}^p_\Gamma(P_{i_0})$ is congruent to a diverging process, thus diverging. This contradicts Theorem 7.1.24 and Fact 7.1.23. □

Parametrisation on the core functional calculus

In this section and the following we discuss how the exposed method can be enhanced and enriched. This section focuses on the parametrisation of our method.

The Parametrisation Method In Section 7.1.3, 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.

Figure 7.4 presents a new rule for definition, obtained by removing the non-recursion condition.

Remark 7.1.26 *Here* language *means a set of processes included in the set of all processes we can build with the syntax (namely $\pi_{\mathtt{ST}}$). The important point is that semantics is not a part of the definition of language: if L is a language and $P \in L$, we derive $P \rightarrow P'$ with the semantics rules for $\pi_{\mathtt{ST}}$; thus, P' is not necessarily in L. Being a* terminating language *means being a set of $\pi_{\mathtt{ST}}$ processes terminating for the $\pi_{\mathtt{ST}}$ semantics.*

Definition 7.1.27
$\pi_{\mathtt{core}}$ is defined as the language of processes typable with rules (**Nil**), (**Par**), (**In**), (**Rep**), (**Out**) *and* (**Def − 2**).

Note that the non-terminating process $\texttt{def}\ f = (x).\overline{f}\langle x\rangle\ \texttt{in}\ \overline{f}\langle v\rangle$ is in $\pi_{\texttt{core}}$. Indeed, $\pi_{\texttt{core}}$ does not enforce termination. We propose some conditions to obtain a terminating subset of $\pi_{\texttt{core}}$. Pruning is still defined as in Figure 7.3.

Definition 7.1.28 (Generic conditions)
Let F and L be two languages. The following conditions define what we call the generic conditions *of our method:*

1. *F is terminating.*
2. *$L \subseteq \pi_{\texttt{core}}$*
3. *L is reduction closed (i.e., $P \in L$ and $P \longrightarrow P'$ imply $P' \in L$);*
4. *For all p, $\mathsf{pr}^p_\Gamma(L) \subseteq F$.*

Notice that by condition 4, we will focus on parametrisation languages F which are functional (but it is not mandatory).

Theorem 7.1.29 (Parametrization)
Suppose F and L satisfy the generic conditions. Then L is terminating.

Proof. *First, condition 2 gives us typability (without the non-recursive definition assumption). We have to establish that lemmas of previous section still hold. First, Proposition 7.1.4 holds, from condition 3. Notice that Lemmas 7.1.7 and 7.1.8 were used only for the sake of clarity in proofs and are not mandatory in order to obtain soundness. As we stated above, pruning is defined as in Definition 7.1.9. We disregard Fact 7.1.23, as we will use condition 4. Lemma 7.1.14 and Lemma 7.1.16 still hold, as they use typability but not the non-recursive definition assumption. Lemma 7.1.21 is still true as the typing system for $\pi_{\texttt{core}}$ enforces decreasing the same way as previously. Thus, Lemma 7.1.22 is still true. We replace Fact 7.1.23 by condition 4 and Theorem 7.1.24 by condition 1. This allows us to derive Soundness, using the same proof as the one for Theorem 7.1.25.*

□

Building a sound impure language from a functional one A method to obtain this result is building a type system for $\pi_{\texttt{core}}$ to enforce both conditions 2 and 3. We assume therefore that we are given a terminating subset $\pi^1_{\texttt{def}}$ of the functional processes $\pi_{\texttt{def}}$. From $\pi^1_{\texttt{def}}$ we extract a terminating subset of well-typed impure processes, denoted as $Ter(\pi^1_{\texttt{def}})$ and defined as the set of all processes P such that, whenever $P \to P'$, then $\mathsf{pr}^p_\Gamma(P') \in \pi^1_{\texttt{def}}$.

Remark 7.1.30 *As far as checking whether a process P is in $Ter(\pi^1_{\texttt{def}})$, the above definition is problematic for at least two reasons:*

- *the definition requires finding all possible derivatives of P, and then checking a condition on the pruning of each of them; finding all derivatives of a process is harder than finding whether the process is terminating;*
- *characterising $Ter(\pi^1_{\texttt{def}})$ may be hard even when $\pi^1_{\texttt{def}}$ is trivial (for instance, the continuations of definitions are all $\mathbf{0}$); for instance, take a process with an imperative input $a(x).Q$, where Q contains a functional pattern that violates $\pi^1_{\texttt{def}}$; now, this pattern is irrelevant as long as the input at a is not consumed, because it will be removed in the pruning; the pattern however becomes relevant if the input is consumed. Unfortunately, knowing whether a π-calculus process can produce an output on given channel is undecidable.*

Therefore, what one is really after is isolating a sublanguage L of $Ter(\pi^1_{\texttt{def}})$ in which checking $P \in L$ is easy. We shall see examples of this in the next section.

Polymorphic functional processes. It is also possible to handle functional polymorphic channels in $\pi_{\mathtt{core}}$, along the lines of [PS00]. We discuss polymorphism in Section 7.1.4.

Polymorphic types are existential, and 'packages' associating types and channels can be transmitted. We allow values of different types to be transmitted on the same polymorphic functional channel, provided they have the same level — that is, we do not try to include any form of *level polymorphism*. Even when the termination of the functional core can be proved using measures, as 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 7.1.5.

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 [GTL89].) In $\pi^{\mathrm{pr}}_{\mathtt{def}}$, termination is given by measures, in $\pi^{1}_{\mathtt{def}}$ by logical relations. The method can be iteratively applied. For instance, having used it to establish termination for the functional $\pi^{1+\mathrm{pr}}_{\mathtt{def}}$ 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.

7.1.4 Refinements

Non-replicated imperative inputs We describe an extension to the type system of Section 7.1.1 to handle other common patterns of usage of imperative channels.

In the type system of Figure 7.2, non-replicated inputs introduce the same constraint $k > l$ as replicated inputs (rules (**PFIn**) and (**PFRep**)). This rule is in some cases too restrictive, as non-replicated inputs are considered irrelevant for termination in previous works (Section 3.1). Remember that the typing rule for linear inputs is simply:

$$(\mathbf{In})\ \frac{\Gamma, x : T \vdash P : l \qquad \Gamma(a) = \sharp^{k}_{\mathtt{I}}\, T}{\Gamma \vdash a(x).P : l}$$

Removing the constraint $k > l$ from rule (**PFIn**) would however be unsafe. For instance, we could type the divergent process

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

using a type assignment $\Gamma(a) = \sharp^{1}_{\mathtt{I}}\,\sharp^{1}_{\mathtt{F}}\,\mathtt{unit}$, $\Gamma(f) = \sharp^{1}_{\mathtt{F}}\,\sharp^{1}_{\mathtt{F}}\,\mathtt{unit}$ and $\Gamma(g) = \sharp^{1}_{\mathtt{F}}\,\mathtt{unit}$. Here both definitions could be typed, the level of all names being equal to 1. Thus, in both cases, the corresponding inequality is $1 \geq 1$. However this process diverges. In the process, a is imperative and used to propagate name g inside the continuation of the definition of g. This is something we cannot do with functional names alone. This example shows some of the subtle interferences between functional and imperative constructs that may cause divergences.

We can improve rule (**PFIn**) so to impose the constraint $k > l$ only when the value communicated in the input is functional. Rule (**PFIn**) is replaced by the rule (**PFIn′**), where "T functional" holds if T is of the form $\sharp^{k'}_{\mathtt{F}}\, T'$, for some k', T':

As we want the property $P \to^{n} P' \Rightarrow \mathsf{pr}^{2,p}_{\Gamma}(P) \simeq \mathsf{pr}^{2,p}_{\Gamma}(P')$ if $n \leq K$ to hold, we have to replace, during the pruning, every imperative name by a generic name to take care that the imperative reduction $\to^{n}$ can instantiate imperative variables in message position.

Definition 7.1.31 (Pruning)
We define the pruning on names with:

$$\mathrm{pr}(a) = \star \qquad\qquad \mathrm{pr}(f) = f$$

$$(\mathbf{PFIn}')\frac{\Gamma \vdash^2 P : l \qquad \vdash^2 a : \sharp_{\mathrm{I}}^k T \qquad \vdash^2 x : T \qquad \text{if } T \text{ functional then } k > l \text{ and } l' = 0 \text{ else } l' = l}{\Gamma \vdash^2 a(x).P : l'}$$

Figure 7.5: A refined rule for non-replicated imperative inputs.

$$\mathsf{pr}_\Gamma^{2,p}(!a(x).P) = \mathsf{pr}_\Gamma^{2,p}(\mathbf{0}) \stackrel{\text{def}}{=} \mathbf{0} \qquad \mathsf{pr}_\Gamma^{2,p}(P_1 \mid P_2) \stackrel{\text{def}}{=} \mathsf{pr}_\Gamma^{2,p}(P_1) \mid \mathsf{pr}_\Gamma^{2,p}(P_2) \qquad \mathsf{pr}_\Gamma^{2,p}((\boldsymbol{\nu} a)\, P) \stackrel{\text{def}}{=} \mathsf{pr}_\Gamma^{2,p}(P)$$

$$\mathsf{pr}_\Gamma^{2,p}(\mathtt{def}\ f^n = (x).P_1\ \mathtt{in}\ P_2) \stackrel{\text{def}}{=} \begin{cases} \mathtt{def}\ f = (x).\mathsf{pr}_\Gamma^{2,p}(P_1)\ \mathtt{in}\ \mathsf{pr}_\Gamma^{2,p}(P_2) & \text{if } n = p \\ \mathsf{pr}_\Gamma^{2,p}(P_2) & \text{otherwise} \end{cases}$$

$$\mathsf{pr}_\Gamma^{2,p}(\overline{f}^n\langle v\rangle) = \begin{cases} \overline{f}^n\langle \mathrm{pr}(v)\rangle & \text{if } n = p \\ \mathbf{0} & \text{otherwise} \end{cases} \qquad \mathsf{pr}_\Gamma^{2,p}(a(x).P) = \begin{cases} \mathbf{0} & \text{if } \Gamma(a) = \sharp_{\mathrm{I}}^n T \text{ and } T \text{ functional} \\ \mathsf{pr}_\Gamma^{2,p}(P) & \text{otherwise} \end{cases}$$

Figure 7.6: Pruning accommodating rule (**PFIn**′)

We replace the previous process pruning definition (Definition 7.1.9) by the one in Figure 7.6.

Notice that pr() collapses every imperative name on $\star$.

Fact 7.1.32 (Pruning of names)
If $\Gamma \vdash^2 P : l$ and $v \in \mathrm{fn}(\mathsf{pr}_\Gamma^{2,p}(P))$, then either $v = f$ or $v = \star$.

Proof. *By induction on the typing judgement. The only interesting case begin the functional output. We use Definition 7.1.31 to conclude.* □

Fact 7.1.33 (Pruning and substitution)
If $\Gamma \vdash^2 P : l$, $x \in \mathrm{fn}(P)$ and $\Gamma(x) = \Gamma(v) = T$ which is not functional, then $\mathsf{pr}_\Gamma^p(P) = \mathsf{pr}_\Gamma^p(P\{v/x\})$

Proof.
Easily done by induction on the typing judgement, using Fact 7.1.32.

□

Of course, non-replicated inputs no longer prevent the active outputs in their continuation to be taken into account in the measure. Thus we have to change Definition 7.1.17:

Definition 7.1.34 (Active outputs)

$$\mathbf{Os}^{2,p}(\mathbf{0}) = 0 \qquad \mathbf{Os}^{2,p}(P_1 \mid P_2) = \mathbf{Os}^{2,p}(P_1) + \mathbf{Os}^{2,p}(P_2) \qquad \mathbf{Os}^{2,p}(!a(x).P) = \mathbf{Os}^{2,p}(a(x).P) = 0$$

$$\mathbf{Os}^{2,p}(\mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2) = \mathbf{Os}^{2,p}(P_2) \qquad \mathbf{Os}^{2,p}((\boldsymbol{\nu} a)\ P) = \mathbf{Os}^{2,p}(P) \qquad \begin{array}{ll} \mathbf{Os}^{2,p}(\overline{v}^k\langle w\rangle) = 0 & \text{if } k \neq p \\ \mathbf{Os}^{2,p}(\overline{v}^p\langle w\rangle) = 1 & \end{array}$$

These alterations allow us to prove soundness:

Theorem 7.1.35 (Soundness)
If $\Gamma \vdash^2 P : l$, then P terminates.

Proof. *We have to examine the Lemmas of Section 7.1.3 and prove that they still hold. There is no problem to prove Lemma 7.1.2 and Proposition 7.1.4 as we can use the induction hypothesis in case* (**PFIn**′). *Typability of pruned terms is obtained by pruning types (we define a type-pruning operator* pr() *with:*

$\mathrm{pr}(\mathbb{1}) = \mathrm{pr}(\sharp^k_{\mathtt{I}} T) = \mathrm{pr}(\mathbb{1})$ *and* $\mathrm{pr}(\sharp^k_{\mathtt{F}} T) = \sharp^k_{\mathtt{F}} \mathrm{pr}(T)$*). Case 3 of Lemma 7.1.14 is replaced by "if* $\Gamma \vdash^2 a(x).P_1 : 0$ *and* $\Gamma(a) = \sharp^k_{\mathtt{I}} T$ *with* T *functional,* $k \leq p$ *and* $\Gamma(v) = T$*, then* $\mathsf{pr}^{2,p}_\Gamma(P_1)\{v/x\} \sim \mathbf{0}$*", the proof is the same, except for case* (**PFIn′**)*, where we use the induction hypothesis to conclude.*

In the proof of the Simulation Lemma, Case (**comm**) *has to be treated separately. We have* $P = \mathbf{E}[a(x).P_1 \mid \overline{a}\langle v\rangle]$*. Here we discuss the type* T *of* x *(and* v*):*

- *Either* T *is functional, and we conclude as in case* (**trig**)*, using the new version of Lemma 7.1.14.*
- *Or* T *is not functional. Thus* $\mathsf{pr}^{2,p}_\Gamma(P) = \mathsf{pr}^{2,p}_\Gamma(\mathbf{E})[\mathsf{pr}^{2,p}_\Gamma(P_1) \mid \mathbf{0}]$ *and* $\mathsf{pr}^{2,p}_\Gamma(P') = \mathsf{pr}^{2,p}_\Gamma(\mathbf{E})[\mathsf{pr}^{2,p}_\Gamma(P_1\{v/x\})]$*. We conclude using Fact 7.1.33.*

Thanks to Definition 7.1.34, Lemma 7.1.20 still holds. In the proof of Lemma 7.1.21, case (**comm**) *of case 2 has to be treated separately:* $P = \mathbf{E}[a(x).P_1 \mid \overline{a}\langle v\rangle]$ *and* $P' = \mathbf{E}[a(x).P_1 \mid P_1\{v/x\}]$*. From* $\Gamma \vdash^2 P : l$*, we deduce, among other judgements,* $\Gamma \vdash^2 P_1 : l_1$*,* $\Gamma(a) = \sharp^n_{\mathtt{I}} T$*. We discuss type* T*:*

- *Either* T *is functional, and* $n > l_1$*, we conclude as in case* (**trig**) *using Lemma 7.1.20.*
- *Or* T *is not functional. We prove easily by induction that* $\mathbf{Os}^{2,p}(P_1\{v/x\}) = \mathbf{Os}^{2,p}(P_1)$*. As* $p > n$*, we have* $\mathbf{Os}^{2,p}(\overline{a}\langle v\rangle) = 0$*. The definition of* $\mathbf{Os}^{2,p}()$ *gives* $\mathbf{Os}^{2,p}(P) = \mathbf{Os}^{2,p}(\mathbf{E}) + \mathbf{Os}^{2,p}(P_1) + \mathbf{Os}^{2,p}(\overline{a}\langle v\rangle)$ *and* $\mathbf{Os}^{2,p}(P') = \mathbf{Os}^{2,p}(\mathbf{E}) + \mathbf{Os}^{2,p}(P_1\{v/x\})$*. We conclude.*

In a similar way, case (**comm**) *of case 3 has to be treated separately:* $P = \mathbf{E}[a(x).P_1 \mid \overline{a}\langle v\rangle]$ *and* $P' = \mathbf{E}[a(x).P_1 \mid P_1\{v/x\}]$*. From* $\Gamma \vdash^2 P : l$*, we deduce, among other judgements,* $\Gamma \vdash^2 P_1 : l_1$*,* $\Gamma(a) = \sharp^n_{\mathtt{I}} T$*. We discuss type* T*:*

- *Either* T *is functional, and* $n > l_1$*, we conclude as in case* (**trig**) *using Lemma 7.1.20.*
- *Or* T *is not functional. We prove easily by induction that* $\mathbf{Os}^{2,P_1\{v/x\}}(=)\mathbf{Os}^{2,P_1}()$*. We have* $\mathbf{Os}^{2,K}(\overline{a}\langle v\rangle) = 1$*. The definition of* $\mathbf{Os}^{2,p}()$ *gives* $\mathbf{Os}^{2,K}(P) = \mathbf{Os}^{2,p}(\mathbf{E}) + \mathbf{Os}^{2,P_1}(+)\mathbf{Os}^{2,p}(\overline{a}\langle v\rangle)$ *and* $\mathbf{Os}^{2,K}(P') = \mathbf{Os}^{2,p}(\mathbf{E}) + \mathbf{Os}^{2,p}(P_1\{v/x\})$*. We conclude.*

Thus, we are still able to prove Lemma 7.1.22. We conclude by proving Theorem 7.1.35 the same way we proved Theorem 7.1.25.

□

Remark 7.1.36 (Refining further) *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 Section 7.1.5. (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; e.g., 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).

Remark 7.1.37 (Conditional operators and non-replicated imperative inputs) *One has to be careful when introducing the* `if then else` *operator in this formalism. Indeed, for the Simulation Lemma to hold, we have, informally, to prove the following property "prefixes we prune should not have an influence (forbidding or creating interactions) on prefixes not pruned". This extension forces us, when pruning, to map every imperative name to the same generic name* d*, as we want* $\mathsf{pr}^{2,p}_\Gamma(a(x).P \mid \overline{a}\langle v\rangle) = \mathsf{pr}^{2,p}_\Gamma(P[v/x])$*. Thus we have to be careful when typing processes such as* `def` $f^p = (x).x(n).$`if` $n = 0$ `then` P_1 `else` P_2 `in` *. If* P_1 *and* P_2 *contains functional prefixes on level* p*, then we break the simulation if we type the imperative input on* x *with the rule of this extension: as* n *will be mapped to a generic value, the* `if then else` *reduction will not remember if it should reduced to* $\mathsf{pr}^{2,p}_\Gamma(P_1)$ *or* $\mathsf{pr}^{2,p}_\Gamma(P_2)$*. Thus, in a context of introduction of* `if then else` *, we have to refine the typing rule for non-replicated imperative inputs.*

$$(\mathbf{Nil})\frac{}{\Gamma \vdash^3 \mathbf{0} : (\emptyset, 0)} \qquad (\mathbf{Res})\ \frac{\Gamma \vdash^3 P : (N, l)}{\Gamma \vdash^3 (\boldsymbol{\nu} a)\, P : (N, l)} \qquad (\mathbf{Par})\frac{\Gamma \vdash^3 P_1 : (N_1, l_1) \qquad \Gamma \vdash^3 P_2 : (N_2, l_2)}{\Gamma \vdash^3 P_1 \mid P_2 : (N_1 \uplus N_2, \max(l_1, l_2))}$$

$$(\mathbf{Def})\ \frac{\Gamma \vdash^3 P_1 : (N_1, l_1) \qquad \Gamma \vdash^3 P_2 : (N_2, l_2) \qquad \Gamma(f) = \sharp_{\mathtt{F}}^{k} T \qquad \Gamma(x) = T \qquad \{k\} >_{\mathrm{mul}} N_2 \qquad f \notin \mathrm{fn}(P_1)}{\Gamma \vdash^3 \mathtt{def}\ f = (x).P_1\ \mathtt{in}\ P_2 : (N_2, l_2)}$$

$$(\mathbf{Out-f})\ \frac{\Gamma \vdash^3 P : (N, l) \qquad \Gamma f = \sharp_{\mathtt{F}}^{k} T \qquad \Gamma(w) = T}{\Gamma \vdash^3 \overline{f}\langle w \rangle.P : (\{k\} \uplus N, \max(l, k))}$$

$$(\mathbf{Out-i})\ \frac{\Gamma \vdash^3 P : (N, l) \qquad \Gamma \vdash^3 (:(a,)) = \sharp_{\mathtt{F}}^{k} T \qquad \Gamma(w) = T}{\Gamma \vdash^3 \overline{a}\langle w \rangle.P : (\{k\} \uplus N, l)}$$

$$(\mathbf{Rep}')\ \frac{\Gamma \vdash^3 P : (N, l) \qquad \forall j, \Gamma(a_j) = \sharp_{\mathtt{I}}^{k_j} T_j \text{ and } \Gamma(x_j) = T_j \qquad \uplus_j k_j <_{\mathrm{mul}} N \qquad \max_j(k_j) > l}{\Gamma \vdash^3 !a_1(x_1).\ldots.a_p(x_p).P : (\emptyset, 0)}$$

$$(\mathbf{In}')\ \frac{\Gamma \vdash^3 P : (N, l) \qquad \forall j, \Gamma(a_j) = \sharp_{\mathtt{I}}^{k_j} T_j \text{ and } \Gamma(x_j) = T_j \qquad \uplus_j k_j <_{\mathrm{mul}} N \qquad \max_j(k_j) > l}{\Gamma \vdash^3 a_1(x_1).\ldots.a_p(x_p).P : (\emptyset, 0)}$$

Figure 7.7: Typing rules accommodating input sequences

Remark 7.1.38 (Polymorphism) *The method can be extended to handle polymorphism [Tur96]. 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).*

Remark 7.1.39 (Allowing multiple definitions of a same name) *Actually, one is able to refine the proof from [San06] in order to prove that termination still holds if one allows multiple definition on a same name. Functionality and conluence are no longer enforced, but the hierarchy induced by the syntax still ensure soundness. This remark allows us to reach greater expressivity for the termination in our impure setting, by considering as a core calculus such a language.*

Accommodating 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 Section 3.2.1.

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. More refined analyses, corresponding to more sophisticated measure-based systems, could be adapted too.

We thus assume that the syntax of the calculus allows replicated input sequences, i.e., a construct $!a_1(x_1).\ldots.a_n(x_n).P$. We use the integer l to denote the functional weight of a process in a typing judgement.

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.

In this new setting, to type a (possibly replicated) imperative input guarded by a sequence of imperative inputs, we now require the multiset sum of the levels of the input subjects to be greater than the weight of the continuation. For example, if a and b are imperative names of levels respectively 2 and 3 and f is a functional name of level 1, then we can type the replication $!b.a.(\overline{a} \mid \overline{f} \mid \overline{f})$ as the multiset sum of the levels of the inputs $\{2,3\}$ is strictly greater than the weight of the continuation $\{2,1,1\}$. This replication could not be typed previously, as, according to the previous typing rule for imperative input, only the level of a is compared to the weight of the continuation.

We have to prevent replications from trading imperative names for functional names of the same level. Otherwise, we would be able to derive $\vdash (\texttt{def}\ f = ().(\overline{a} \mid \overline{a})\ \texttt{in}\ !a.a.\overline{f} \mid \overline{f}) : 1$ with $\Gamma(a) = \sharp^1_{\mathtt{I}}$ and $\Gamma(u) = \sharp^1_{\mathtt{F}}$. Indeed, the imperative replication would be typed as the multiset sum of the levels of the two inputs on a ($\{1,1\}$) is strictly greater than the weight of $\overline{f}$ ($\{1\}$). However, this process is also diverging. This behaviour is ruled out by introducing $\mathrm{Mlf}(P)$, defined as the maximum level of an active functional output in P, and by writing the typing rule for replicated inputs as follows (a similar rule can be introduced for non-replicated imperative outputs):

Remark 7.1.40 *Notice that this extension gives us another way to type-check imperative references in the π-calculus (the other one being the refinement about non-replicated inputs presented above). To type the process $!add(acc,n).acc(x).\overline{acc}\langle x+n\rangle$, we can consider $add(acc,n).acc(x)$ as a single input sequence, which is heavier than the weight of the output $\overline{acc}\langle x+n\rangle$.*

Remark 7.1.41 (A refinement of rule (Def)) *The language $\pi^1_{\mathtt{ST}}$ was defined in Section 7.1.2 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 7.1.29, 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).*

7.1.5 Examples

The examples in this section use polyadicity, and first-order values such as integers and related constructs (including arithmetic constructs, and if-then-else statements).

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 the lock $lock_i$.

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 as well have been directed to other clients).

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

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

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

and **enc**, **dec** are operators for encryption and decryption, with equalities $\mathbf{dec}_i[\mathbf{enc}[c_i, m]] = m$ for all i.

In the typing all the c_i's and s must have the same type, because an output on c_i can follow an input on s on the server's side, and conversely on the clients' side.

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

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

Thus, the functions $\mathbf{dec}_i$ have type $\mathtt{b} \to \mathtt{b}$ and the function **enc** has type $(\sharp^1_{\mathtt{F}}\, \mathtt{b} \times \mathtt{b}) \to \mathtt{b}$. 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).

Moreover, the definitions abides to the $\pi^1_{\mathtt{def}}$ conditions: s does not appear in the definition of s, and c_i does not appear in C_i.

The loose assignment of levels to functional names (the possibility $k = l$ in rule (**Def**) of Section 7.1.2) 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 have the same level.

This, and the combination of imperative and functional features prevent Sys_1 from being typable in previous type systems.

A movie-on-demand server In this example, the server s is a movie database, and handles requests to watch a 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 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 $!c.r'(z).(\overline{c} \mid z)$ keeps interrogating the server: the client tries to watch the movie as many times as possible. (The example could be refined by having a client that allows some friends to watch the movie, by sending out r'; or by having a server that signals that the credit on a movie is extinguished.)

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

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 $\pi^{\mathrm{pr}}_{\mathtt{def}}$ (presented above) 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 type-check the example Sys_2, we use the following type-assignment:

$$f, \mathbf{tintin}, \mathbf{asterix} : \mathtt{b} \qquad h, z : \mathtt{b} \qquad r, r' : \sharp^1_{\mathtt{F}}\,\mathtt{b} \qquad s : \sharp^2_{\mathtt{F}}\,(\mathtt{nat} \times \sharp^1_{\mathtt{F}}\,\mathtt{b} \times \mathtt{b}) \qquad c : \sharp^0_{\mathtt{I}}\,\mathtt{b}$$

We assume that the weight of an `if then else` construct is the maximum weight of each branch. The 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).

Notice that the recursion on c in $!c.r'(z).(\overline{c} \mid z)$ is controlled by the input on $r'(z)$. Rule (**In**) ensures that the level of r is strictly greater than the level of c. Indeed, no divergence can arise from this recursion, as inputs on r' are required to feed the loop.

System Sys_2 can actually be typed in existing pure measure systems as those in Section 3.2.1; i.e., without appealing to the existence of the terminating functional core $\pi^{\mathrm{pr}}_{\mathtt{def}}$. Consider however 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.

If $Sys_{2'}$ contains several clients, using both s and s', then typing $Sys_{2'}$ in a pure weight-based system seems difficult: s and s', being interchangeable, must have the same level and type, and then the input-output sequence in the forwarder is not typable. In contrast, with our method 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 $\pi^{\mathrm{pr}}_{\mathtt{def}}$.

A termination proof of the core calculus $\pi^{\mathrm{pr}}_{\mathtt{def}}$ by a simple measure argument is not in contradiction with the previous claim 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 s and s' need not have the same type).

Merging the two examples The example presented here refines the two previous examples 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 7.8. $\mathrm{Server_i}$ 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. 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 $\mathrm{Server_i}$.

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

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

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

Figure 7.8: A distributed movie server interrogating a bank

In order to type-check this example, as discussed at the end of Section 7.1.3, we view it as belonging in a calculus that consists in the superposition of three calculi. Name s is a functional name in $\pi^1_{\texttt{def}}$, the servers s_i are functional names in $\pi^{\text{pr}}_{\texttt{def}}$, and the bank methods $create, add, get$ are imperative names. We first prove the termination of $\pi^{1+\text{pr}}_{\texttt{def}}$, the calculus containing both these languages, 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 7.1.4 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**), as discussed in Remark 7.1.41.

7.2 In an impure λ-calculus

This section propose a new presentation and a new soundess proof for the type and effect systems for a λ-calculus with references proposed by [Bou07] and [Ama09].

7.2.1 A λ-calculus with references

In this section, the technique presented in the previous section is adapted to $\lambda_{\texttt{ref}}$, a call-by-value λ-calculus (see Section 2) extended with imperative operations (read, write and update) acting on a store. With respect to the impure π-calculus we presented in Section 7.1.1, intuitively, references correspond to imperative names, and functions to functional names. We transport the constraints from the π-calculus processes onto the λ-calculus terms following this analogy. In $\lambda_{\texttt{ref}}$ some level and typing annotations are directly placed into the

syntax (this is not mandatory, but it lets us avoid some technical details).

The store is stratified into regions, which are referred to using natural numbers (the 'levels' of Section 7.1.1). Commands involving imperative operations are annotated by a natural number: a command acts on regions of a given level.

To define terms, we use a new set of variables $\mathcal{A}$, called *addresses*. Addresses are written $u_{(n,T)}$: they are explicitly associated both to a region n and a type T (types are described below). Note that values of different types can be stored in the same region. We suppose that there exists an infinite number of addresses for a given pair of a type and a region. Stores, ranged over using δ, are partial mappings from addresses to values. The (finite) support of δ is written $\mathrm{supp}(\delta)$, $\emptyset$ is the empty store ($\mathrm{supp}(\emptyset) = \emptyset$), and $\delta\langle u_{(n,T)} \rightsquigarrow V\rangle$ denotes the store δ' defined by $\delta'(u_{(n,T)}) = V$ and $\delta'(v) = \delta(v)$ for every $v \in \mathrm{supp}(\delta)$ such that $v \neq u_{(n,T)}$.

The syntax for terms, types, values, redexes and evaluation contexts is as follows:

$$
\begin{array}{rcl}
M & ::= & (M\ M) \mid x \mid \lambda x.\, M \mid \star \\
& & \mid \mathtt{ref}_n\ M \mid \mathtt{deref}_n(M) \mid M \mathtt{:=}_n M \mid u_{(n,T)} \\
T & ::= & \mathbb{1} \mid T\ \mathtt{ref}_n \mid T \to^n T \\
V & ::= & \lambda x.\, M \mid x \mid u_{(n,T)} \mid \star \\
R & ::= & (\lambda x.\, M)\ V \\
& & \mid \mathtt{deref}_n(u_{(n,T)}) \mid \mathtt{ref}_n\ V \mid u_{(n,T)} \mathtt{:=}_n V \\
\mathbf{E} & ::= & \mid V\ \mathbf{E} \mid \mathbf{E}\ M \\
& & \mid \mathtt{deref}_n(\mathbf{E}) \mid \mathtt{ref}_n\ \mathbf{E} \mid \mathbf{E} \mathtt{:=}_n M \mid V \mathtt{:=}_n \mathbf{E}
\end{array}
$$

Terms are constructed with applications, abstractions and variables, like in the standard λ-calculus, but also with addresses, the $\star$ constant and the three imperative operators. $\mathtt{ref}_n\ M$ is the creation of a new address of level n containing M, $\mathtt{deref}_n(M)$ is the reading of what is contained at the address M (notice that M has to be computed first) and $M \mathtt{:=}_n N$ is the update of the address M by the term N.

Types are the standard arrow types of $\lambda_{\mathbf{ST}}$ to which we add a reference type ($T\ \mathtt{ref}_n$ is the type of an address of region n containing values of type T). Arrow types are annotated with levels: intuitively $T_1 \to^n T_2$ is the type of a function taking arguments of type T_1, returning values of type T_2 using a computation which accesses regions in the memory *up to* the region n.

We impose a well-formedness condition on types, that intuitively reflects the stratification of regions: a term acting at regions less than or equal to n cannot be stored in a region smaller than $n+1$. For this, we define $\mathrm{reg}(T)$, the set of regions accessed by T, by:

$$
\mathrm{reg}(\mathbb{1}) = 0 \qquad \mathrm{reg}(T\ \mathtt{ref}_n) = \max(n, \mathrm{reg}(T))
$$
$$
\mathrm{reg}(T_1 \to^n T_2) = \max(n, \mathrm{reg}(T_2))
$$

Definition 7.2.1 (Well-formedness of types) *A type T is well-formed, written* $\mathrm{wf}(T)$*, if for all its subtypes of the form $T'\ \mathtt{ref}_n$, we have* $\mathrm{reg}(T') < n$.

In the following, we shall implicitly assume that all types we manipulate are well-formed.

7.2.2 Type and effect system

The type system we present in this section is actually the one given in [Bou07] and [Ama09]. The two presentations are quite similar, but the proofs are different. Our presentation replaces regions, defined in [Bou07] as abstract parts of the store (denoted by ρ), by natural numbers. In the former presentation, an order between regions could be extracted, we replace it here by the standard ordering on integers. Moreover, in our presentation, we make some region annotations explicit, in the syntax and in the types.

Another difference is that our well-formedness condition for types is actually looser than the one found in [Bou07], allowing us to typecheck more terms. Indeed, when considering an arrow-type, we do not propagate

Typing rules for terms

$$(\textbf{App})\ \frac{\Gamma \vdash M : (T_1 \to^n T_2, m) \qquad \Gamma \vdash N : (T_1, k)}{\Gamma \vdash M\ N : (T_2, \max(m,n,k))} \qquad (\textbf{Abs})\ \frac{\Gamma \vdash M : (T_2, n) \qquad \Gamma(x) = T_1}{\Gamma \vdash \lambda x.\, M : (T_1 \to^n T_2, 0)}$$

$$(\textbf{Ref})\ \frac{\Gamma \vdash M : (T_1, m)}{\Gamma \vdash \mathtt{ref}_n\ M : (T_1\ \mathtt{ref}_n, \max(n,m))} \qquad (\textbf{Var})\ \frac{\Gamma(x) = T_1}{\Gamma \vdash x : (T_1, 0)} \qquad (\textbf{Uni})\frac{}{\Gamma \vdash \star : (\mathbb{1}, 0)}$$

$$(\textbf{Add})\ \frac{}{\Gamma \vdash u_{(n,T_1)} : (T_1\ \mathtt{ref}_n, 0)} \qquad (\textbf{Asg})\ \frac{\Gamma \vdash M : (T_1\ \mathtt{ref}_n, m) \qquad \Gamma \vdash N : (T_1, k)}{\Gamma \vdash M \mathtt{:=}_n N : (\mathbb{1}, \max(m,n,k))}$$

$$(\textbf{Drf})\ \frac{\Gamma \vdash M : (T\ \mathtt{ref}_n, m)}{\Gamma \vdash \mathtt{deref}_n(M) : (T, \max(m,n))}$$

Typing rules for stores

$$(\textbf{Emp})\frac{}{\Gamma \vdash \emptyset} \qquad (\textbf{Sto})\frac{\Gamma \vdash \delta \qquad \Gamma \vdash V : (T, 0)}{\Gamma \vdash \delta\langle u_{(n,T)} \rightsquigarrow V\rangle}$$

Figure 7.9: $\lambda_{\mathtt{ref}}$: Type and Effect System

the checks on the well-formedness conditions in the type of the argument. The proof of [Bou07] can easily be adapted with this small refinement.

Figure 7.9 defines two typing judgements, of the form $\Gamma \vdash M : (T, n)$ for terms and $\Gamma \vdash \delta$ for stores. As above, our type system is presented *à la Church*, and we write $\Gamma(x) = T$ whenever variable x has type T according to Γ.

In the type and effect system of $\lambda_{\mathtt{ref}}$, a typing judgement has the form $\Gamma \vdash M : (T, n)$, where n defines a bound on the *effect* of the evaluation of M, intuitively the maximum level of a region accessed when evaluating M. Effects can be thought of as sets of regions, and are given by a natural number, intuitively corresponding to the maximum level of a region in the effect (thus, we will prove later that values have effect 0).

Contrarily to the π-calculus case (Figure 7.2), the typing rules do not feature inequality constraints on levels, as the stratification of the store is guaranteed by the well-formedness of types. Their main purpose is to record the effect of computations.

We extend typing to evaluation contexts by treating the hole as a term variable which can be given any type.

The execution of programs is specified by a reduction relation written $\mapsto$, relating pairs consisting of a term and a store, and which is defined on Figure 7.10. As in the previous section, we write $\mapsto^n_{\mathrm{F}}$ for a *functional* reduction, obtained using rule (β); n refers to the effect of the β-redex, that is, in this call-by-value setting, the level that decorates the type of the function being triggered (that is, we suppose in rule (β) that $\Gamma \vdash \lambda x.\, M : (T_V \to^n T, \_)$). We introduce similarly *imperative* reductions, noted $\mapsto^n_{\mathrm{I}}$, for reductions obtained using rules (**ref**), (**deref**) or (**store**) (in these cases, the level n appears explicitly in the rules of Figure 7.10).

The following fact explains that this calculus ensures determinism, as opposed to the previous π-calculi we considered.

Fact 7.2.2 (Determinism)
Given M, either M is a value or there exist a unique evaluation context $\mathbf{E}$ and a redex R such that $M = \mathbf{E}[R]$.

Proof. *By structural induction on M.* □

$$(\beta)\frac{}{(\lambda x.\, M\ V, \delta) \mapsto (M\{V/x\}, \delta)} \qquad (\mathbf{ref})\frac{u_{(n,T)} \notin \mathrm{supp}(\delta) \qquad \Gamma \vdash V : (T, \_)}{(\mathtt{ref}_n\ V, \delta) \mapsto (u_{(n,T)}, (\delta\langle u_{(n,T)} \rightsquigarrow V\rangle))}$$

$$(\mathbf{deref})\frac{\delta(u_{(n,T)}) = V}{(\mathtt{deref}_n(u_{(n,T)}), \delta) \mapsto (V, \delta)} \qquad (\mathbf{store})\frac{\Gamma \vdash V : (T, \_)}{(u_{(n,T)} \mathtt{:=}_n V, \delta) \mapsto (\star, (\delta\langle u_{(n,T)} \rightsquigarrow V\rangle))}$$

$$(\mathbf{context})\frac{(M, \delta) \mapsto (M', \delta')}{(\mathbf{E}[M], \delta) \mapsto (\mathbf{E}[M'], \delta')}$$

Figure 7.10: $\lambda_{\mathtt{ref}}$: Reduction Rules

The notions of infinite computation and termination are defined the same way as in the standard λ-calculus of Section 2.

First, we notice in the following fact that values have an effect 0. This can be easily understood as values cannot reduce and as the effect of a term stands for the maximum region accessed during its evaluation.

Fact 7.2.3 (Value effect)
If V is a value and $\Gamma \vdash V : (T, m)$, then $m = 0$.

Proof. *By examining the last rule used to derive $\Gamma \vdash V : (T, m)$. Indeed, rules* (**Abs**), (**Var**), (**Uni**) *and* (**Add**) *have an effect 0 in their conclusion.*

□

The following result will be used in the proof of Lemma 7.2.5 to handle evaluation contexts, it claims that we can replace a term inside an evaluation context with a term of the same type and a smaller effect and preserve typability. The effect of the whole term can decrease (in the case where $\mathbf{E} = [\,]$ for instance).

Lemma 7.2.4 (Subtyping)
If $\Gamma \vdash \mathbf{E}[M_1] : (T, n)$, $\Gamma \vdash M_1 : (T_1, m_1)$, $\Gamma \vdash M_{(1)} : (T_1, m_{(1)})$ and $m_{(1)} \leq m_1$,
then $\Gamma \vdash \mathbf{E}[M_{(1)}] : (T, n')$ with $n' \leq n$.

Proof.
The proof of subtyping in a way similar to what we did in the concurrent case (Lemma 7.1.3) and can be found in Appendix A.

□

Our type and effect system enjoys the two standard properties of subject substitution and subject reduction. Notice that in the statement of Lemma 7.2.5, the effect associated to $M\{V/x\}$ is the same as the one associated to M. This holds as the term V is a value and thus does not introduce accesses to the memory which are not handled by the type system. In a call-by-name setting, the statement of this proposition would be: "If $\Gamma \vdash M : (T, n)$, $\Gamma(x) = T'$ and $\Gamma \vdash N : (T', m)$ then $\Gamma \vdash M\{N/x\} : (T, \max(m, n))$".

Lemma 7.2.5 (Subject substitution)
If $\Gamma \vdash M : (T, n)$, $\Gamma(x) = T'$ and $\Gamma \vdash V : (T', m)$ then $\Gamma \vdash M\{V/x\} : (T, n)$.

Proof. *First notice that Fact 7.2.3 implies $m = 0$.*

From $\Gamma \vdash M : (T, n)$ and $\Gamma \vdash V : (T', m)$, we derive $\Gamma \vdash M\{V/x\} : (T, n)$. We proceed by induction on the typing derivation:

- *Case* (**Var**). *Rule* (**Var**) *gives $\Gamma \vdash y : (T, 0)$.*
  - *Either $x = y$, $T = T'$ and $M\{V/x\} = V$. As $n = m = 0$, we conclude.*

– *Or* $x \neq y$ *and* $M\{V/x\} = y$. *We use* (**Var**) *to derive* $\Gamma \vdash y\{V/x\} : (T, 0)$.

- *Case* (**App**). *We have* $M = (M_1\ M_2)$. *We derive* $\Gamma \vdash M_1 : (T_2 \to^{n_3} T, n_1)$ *and* $\Gamma \vdash M_2 : (T_2, n_2)$ *with* $n = \max(n_1, n_2, n_3)$. *We use the induction hypothesis to get* $\Gamma \vdash M_1\{V/x\} : (T_2 \to^{n_3} T, n_1)$ *and* $\Gamma \vdash M_2\{V/x\} : (T_2, n_2)$ *and we conclude using rule* (**App**).

- *Case* (**Abs**). *We have* $M = \lambda y.M_1$, $\Gamma(y) = T_2$, *and* $T = T_2 \to^{n_1} T_1$. *We derive* $\Gamma \vdash M_1 : (T_1, n_1)$. *We use the induction hypothesis to get* $\Gamma \vdash M_1\{V/x\} : (T_1, n_1)$ *and we conclude using rule* (**Abs**).

- *Case* (**Add**). *We have* $M = M' = u_{(n,T')}$ *and we use* (**Add**) *to derive* $\Gamma \vdash u_{(n,T')}\{V/x\} : (T, 0)$. *item Case* (**Ref**). *We have* $M = \mathtt{ref}_m\ M_1$. *We derive* $\Gamma \vdash M_1 : (T_1, n_1)$ *with* $T = T_1\ \mathtt{ref}_m$ *and* $n = \max(n_1, m)$. *We use the induction hypothesis to get* $\Gamma \vdash M_1\{V/x\} : (T_1, n_1)$ *and we conclude using rule* (**Ref**).

- *Cases* (**Sto**), (**Drf**) *are similar. Case* (**Uni**) *is easy.*

□

Proposition 7.2.6 (Subject reduction)
$\Gamma \vdash M : (T, n)$, $\Gamma \vdash \delta$ *and* $(M, \delta) \mapsto (M', \delta')$ *entail that* $\Gamma \vdash \delta'$ *and* $\Gamma \vdash M' : (T, n')$ *for some* $n' \leq n$.

Proof. *By induction on the derivation of* $(M, \delta) \mapsto (M', \delta')$,

- *Case* (**context**). *Then* $M = \mathbf{E}[M_1]$, $M' = \mathbf{E}[M_1']$ *and* $(M_1, \delta) \mapsto (M_1', \delta')$. *We derive* $\Gamma_1 \vdash M_1 : (T_1, n_1)$. *The induction hypothesis gives us* $\Gamma_1 \vdash M_1' : (T_1, n_1')$ *with* $n_1' \leq n_1$ *and a typing judgement for* δ'. *As* $n_1' \leq n_1$, *we use Lemma 7.2.4 to conclude.*

- *Case* (β). *We have* $M = \lambda x : T_V.M_1\ V$. *We derive* $\Gamma, x : T_V \vdash M_1 : (T, n)$, $\Gamma \vdash V : (T_V, 0)$ *(this holds as* $\lambda x : T_V.M_1$ *has type* $T_V \to^n T$*). We use Lemma 7.2.5 and get* $\Gamma \vdash M_1\{V/x\} : (T, n)$. *We conclude.*

- *Case* (**deref**). *We have* $M = \mathtt{deref}_n(u_{(n,T)})$, $\delta(u_{(n,T)}) = V$, *and* $M' = V$. *From* $\Gamma \vdash \delta$ *we derive* $\Gamma \vdash V : (T, 0)$. *We conclude.*

- *Case* (**ref**). *We have* $M = \mathtt{ref}_n\ V$, $M' = u_{(n,T')}$ *with* $T = T'\ \mathtt{ref}_n$, $\delta' = \delta\langle V \rightsquigarrow u_{(n,T')}\rangle$. *We derive* $\Gamma \vdash V : (T', 0)$. *We use rule* (**Add**) *to build* $\Gamma \vdash u_{(n,T')} : (T, 0)$. *We use rules* (**Sto**) *to get* $\Gamma \vdash \delta\langle V \rightsquigarrow u_{(n,T')}\rangle$.

- *Case* (**store**) *is treated similarly.*

□

Remark 7.2.7 (Comparison with [Bou07, Ama09, Tra10]) *As we hinted above, the question we address in this paper has been studied in a very similar setting in other works. In constrast with the works by Boudol and Amadio, where soundness of the type system is obtained by a 'semantic' approach, be it realisability or reducibility candidates, which is applied to the whole (impure) calculus, we somehow factor out the imperative part of the calculus, which allows us to lift a termination proof of* $\lambda_{\mathbf{ST}}$ *to a termination proof of* λ_{ref}.

Tranquilli [Tra10] proceeds similarly, in two steps: a translation into a purely functional calculus, followed by a termination argument about the latter. However, technically, our approach and his differ considerably; in particular because we project into a subcalculus, using a translation function which seems unrelated to Tranquilli's.

7.2.3 Termination of $\lambda_{\mathtt{ref}}$ programs

In this section we adapt the proof of Section 7.1.3 for the termination of an impure π-calculus to this type and effect system for $\lambda_{\mathtt{ref}}$. We define a pruning operator, obtain a simulation lemma and derive a contradiction to the existence of a diverging well-typed term of $\lambda_{\mathtt{ref}}$ by proving that it implies the divergence of a $\lambda_{\mathbf{ST}}$ term.

An important difference w.r.t. the π-calculus case in defining pruning is that we cannot completely remove a subterm the way we do it above (using $\mathbf{0}$). Moreover, the pruning function is simpler in the context of the π-calculus, because it is correct, when pruning at level p, to get rid of process $\mathtt{def}\ f^n = (x).P_1\ \mathtt{in}\ P_2$ when $n \leq p$. By typing, the subprocess P cannot perform any interaction at level p (Lemma 7.1.14).

On the contrary, in the present setting, when pruning the term $\mathtt{deref}_n(M)$ with $n \leq p$, we cannot simply ignore M, as M could perform reductions at level p. Indeed, computing an address of level n may involve dereferencing at levels above n.

Let us explain the intuition in the definition of pruning for $\mathtt{deref}_n(M)$ (similar ideas are at work when pruning $\mathtt{ref}_n\ M$ and $M :=_n M'$). Because, as explained, we cannot just throw away M, the pruning function enters recursively M, in such a way as to remove imperative constructions from M. Pruning therefore transforms $\mathtt{deref}_n(M)$ into a term that first runs the pruned version of M, and then returns a *generic value* of the appropriate type. Generic values are canonical terms that are used to replace a given subterm *once we know that no divergence can arise due to the evaluation of the subterm* (this would correspond either to a divergence of the subterm, or to a contribution to a more general divergence). They are defined as follows:

Definition 7.2.8 (Generic value)
The generic value $\mathtt{V}_T$ of type T *is defined by:* $\mathtt{V}_{T\ \mathtt{ref}_n} = \mathtt{V}_{\mathbb{1}} = \star$, *and* $\mathtt{V}_{T_1 \to^n T_2} = \lambda x.\mathtt{V}_{T_2}$ *(*x *being of type* T_1*).*

In order to program the evaluation of a pruned subterm and its replacement with a generic value, the definition of pruning makes use of the following projectors:

$$\Pi^{(1,2)} = \lambda x.\lambda y.\, x \qquad\qquad \Pi^{(1,3)} = \lambda x.\lambda y.\lambda z.\, x$$

(in the following, we suppose that these terms are always used in a well-typed fashion).

Finally, in order to define the pruning function, we need a last notion, that conveys the intuition that a given term M can be involved in a reduction at level p. This can happen for two reasons. Either M is able to perform (maybe after some preliminary reduction steps) a reduction at level p, in which case, by the typing rules, the effect of M is greater than p, or M is a function that can receive some arguments and eventually perform a reduction at level p, in which case the type system ensures that its type T satisfies $\mathrm{reg}(T) \geq p$.

Definition 7.2.9 (Related to p)
Suppose $\Gamma \vdash M : (T, n)$. *We say that* M *is* related to p *if either* $n \geq p$ *or* $\mathrm{reg}(T) \geq p$. *In the former case, we can also say that* M *is related to* p *via its effect. In the latter case, via its type.*

We extend this notion to evaluation contexts by treating the hole like a term variable, given a typing derivation for a context (this is relevant in particular in the statement of Lemma 7.2.19).

Notice that a term containing a subterm whose effect is p is not necessarily related to p: for instance $\vdash (\lambda x.\star)\ \lambda y.\mathtt{deref}_3(u_{(3,\mathbb{1})}) : (\mathbb{1}, 0)$ is not related to 3 (one can easily notice that this term cannot be used to trigger a reduction at level 3).

Definition 7.2.10 (Pruning) *Given a typable* M *of type* T, *we define the pruning at level* p *of* M, *written*

$\mathsf{pr}^p_\Gamma(M)$, *as follows:*

$$
\begin{array}{rcl}
\textit{If } M \textit{ is not related to } p\textit{:} & & \\
\mathsf{pr}^p_\Gamma(M) & = & \mathtt{V}_T \\
\textit{Otherwise:} & & \\
\mathsf{pr}^p_\Gamma(M_1\ M_2) & = & \mathsf{pr}^p_\Gamma(M_1)\ \mathsf{pr}^p_\Gamma(M_2) \\
\mathsf{pr}^p_\Gamma(x) & = & x \\
\mathsf{pr}^p_\Gamma(\lambda x.M_1) & = & \lambda x.\mathsf{pr}^p_\Gamma(M_1) \\
\mathsf{pr}^p_\Gamma(\mathtt{ref}_n\ M_1) & = & (\Pi^{(1,2)}\ \star\ \mathsf{pr}^p_\Gamma(M_1)) \\
\mathsf{pr}^p_\Gamma(\mathtt{deref}_n(M_1)) & = & (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(M_1)) \\
\mathsf{pr}^p_\Gamma(M_1 \mathtt{:=}_n M_2) & = & (\Pi^{(1,3)}\ \star\ \mathsf{pr}^p_\Gamma(M_1)\ \mathsf{pr}^p_\Gamma(M_2)) \\
\mathsf{pr}^p_\Gamma(u_{(n,T_1)}) & = & \star
\end{array}
$$

The definition above is extended to contexts as follows. Notice that the pruning of evaluation contexts is not the one obtained by considering $\mathbf{E}$ as a standard term with $[\,]$ as a standard variable. Indeed, in this definition, we do not stop the inductive deconstruction of $\mathbf{E}$ when we reaches an evaluation context which is not related to p.

$$\mathsf{pr}^p_\Gamma([\,]) = [\,] \qquad \mathsf{pr}^p_\Gamma(\mathbf{E}\ M) = \mathsf{pr}^p_\Gamma(\mathbf{E})\ \mathsf{pr}^p_\Gamma(M) \qquad \mathsf{pr}^p_\Gamma(V\ \mathbf{E}) = \mathsf{pr}^p_\Gamma(V)\ \mathsf{pr}^p_\Gamma(\mathbf{E})$$

$$\mathsf{pr}^p_\Gamma(\mathtt{deref}_n(\mathbf{E})) = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(\mathbf{E})) \text{ if } \mathtt{deref}_n(\mathbf{E}) \text{ has type } T \qquad \mathsf{pr}^p_\Gamma(\mathtt{ref}_n\ \mathbf{E}) = (\Pi^{(1,2)}\ \star\ \mathsf{pr}^p_\Gamma(\mathbf{E}))$$

$$\mathsf{pr}^p_\Gamma(\mathbf{E} \mathtt{:=}_n M) = (\Pi^{(1,3)}\ \star\ \mathsf{pr}^p_\Gamma(\mathbf{E})\ \mathsf{pr}^p_\Gamma(M)) \qquad \mathsf{pr}^p_\Gamma(V \mathtt{:=}_n \mathbf{E}) = (\Pi^{(1,3)}\ \star\ \mathsf{pr}^p_\Gamma(V)\ \mathsf{pr}^p_\Gamma(\mathbf{E}))$$

The target of the pruning is the simply typed λ-calculus $\lambda_{\mathbf{ST}}$ (with $\mathbb{1}$ as only base type), as expressed by Lemma 7.2.13. The definitions of values, redexes and evaluation contexts for simply-typed λ-calculus are given in Section 2. Notice that, in this definition, $\lambda_{\mathbf{ST}}$ uses operational semantics given by the full β-reduction, and is not restrained to a strategy.

We define the pruning on types: every reference type is mapped to $\mathbb{1}$ and region annotations on arrow-types are removed.

Definition 7.2.11 (Pruning on types)
Pruning on types is defined by:

$$\mathsf{pr}^p_\Gamma(\mathbb{1}) = \mathbb{1} \qquad \mathsf{pr}^p_\Gamma(T\ \mathtt{ref}_n) = \mathbb{1} \qquad \mathsf{pr}^p_\Gamma(T_1 \to^n T_2) = \mathsf{pr}^p_\Gamma(T_1) \to \mathsf{pr}^p_\Gamma(T_2)$$

It follows immediately from Definition 7.2.11 that $\mathsf{pr}^p_\Gamma(T)$ is a simple type.

Fact 7.2.12 (Generic value – Simple Typability)
For every type T, $\mathtt{V}_T$ is a simply-typed λ-term of type $\mathsf{pr}^p_\Gamma(T)$.

Proof. *Easily done by induction on T.* □

The following lemma will be used to prove that the image of pruning is terminating.

Lemma 7.2.13 (Pruning – Typability)
Take $p \in \mathbb{N}$, and suppose $\Gamma \vdash M : (T, n)$. Then $\mathsf{pr}^p_\Gamma(M)$ is a term of the simply-typed λ-calculus, of type $\mathsf{pr}^p_\Gamma(T)$.

Proof. *By induction on the judgement $\Gamma \vdash M : (T, n)$, we first distinguish that:*

- *either M is not related to p, and $\mathsf{pr}^p_\Gamma(M) = V_T$ and we conclude using Fact 7.2.12,*
- *or M is related to p, and we discuss:*
  - *Case* (**Var**). *We have $\mathsf{pr}^p_\Gamma(x) = x$. In $\lambda_{\mathbf{ST}}$, x is a variable of type $\mathsf{pr}^p_\Gamma(T)$. We conclude.*

– *Case* (**Add**). *We have* $T = T'\ \mathtt{ref}_n$ *and* $\mathsf{pr}^p_\Gamma(T) = \mathbb{1}$. *As* $\mathsf{pr}^p_\Gamma(u) = \star$, *we conclude.*

– *Case* (**Abs**). *We have* $M = \lambda x.M_1$ *and* $T = T' \to^n T_1$. *We have* $\mathsf{pr}^p_\Gamma(T) = \mathsf{pr}^p_\Gamma(T') \to \mathsf{pr}^p_\Gamma(T_1)$. *We consider* x *as a variable of type* $\mathsf{pr}^p_\Gamma(T')$ *in* $\mathsf{pr}^p_\Gamma(M) = \lambda x.\mathsf{pr}^p_\Gamma(M_1)$. *By induction,* $\mathsf{pr}^p_\Gamma(M_1)$ *is of type* $\mathsf{pr}^p_\Gamma(T_1)$. *We conclude.*

– *Case* (**App**). *We have* $M = M_1\ M_2$ *with* M_1 *of type* $T_2 \to^n T_1$ *and* M_2 *of type* T_2. *By induction,* $\mathsf{pr}^p_\Gamma(M_1)$ *has type* $\mathsf{pr}^p_\Gamma(T_2 \to^n T_1) = \mathsf{pr}^p_\Gamma(T_2) \to \mathsf{pr}^p_\Gamma(T_1)$ *and* $\mathsf{pr}^p_\Gamma(M_2)$ *has type* $\mathsf{pr}^p_\Gamma(T_2)$. *As* $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M_1)\ \mathsf{pr}^p_\Gamma(M_2)$, *we conclude.*

– *Case* (**Ref**). *We have* $M = \mathtt{ref}_n\ M_1$, $T = T_1\ \mathtt{ref}_n$ *and* M_1 *has type* T_1. *The induction hypothesis gives* $\mathsf{pr}^p_\Gamma(M_1)$ *of type* $\mathsf{pr}^p_\Gamma(T_1)$. *We get* $\mathsf{pr}^p_\Gamma(T) = \mathbb{1}$. *In* $\lambda_{\mathbf{ST}}$, *the term* $\Pi^{(1,2)}$ *has type* $\mathbb{1} \to \mathsf{pr}^p_\Gamma(T_1) \to \mathbb{1}$ *in the term* $\mathsf{pr}^p_\Gamma(M) = \Pi^{(1,2)}\ \star\ \mathsf{pr}^p_\Gamma(M_1)$. *We conclude.*

– *Case* (**Drf**). *We have* $M = \mathtt{deref}_n(M_1)$, $T_1 = T\ \mathtt{ref}_n$ *and* M_1 *has type* T_1. *The induction hypothesis gives* $\mathsf{pr}^p_\Gamma(M_1)$ *of type* $\mathsf{pr}^p_\Gamma(T_1) = \mathbb{1}$. *In* $\lambda_{\mathbf{ST}}$, *the term* $\Pi^{(1,2)}$ *has type* $\mathsf{pr}^p_\Gamma(T) \to \mathbb{1} \to \mathsf{pr}^p_\Gamma(T)$ *in the term* $\mathsf{pr}^p_\Gamma(M) = \Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(M_1)$. *We conclude.*

– *Case* (**Aff**) *is similar.*

□

Fact 7.2.14 (Pruning – Evaluation context)
For all $\mathbf{E}$, $\mathsf{pr}^p_\Gamma(\mathbf{E})$ *is an evaluation context.*

Proof.
By structural induction on $\mathbf{E}$, *noticing that determinism (Fact 7.2.2) does not hold for* $\lambda_{\mathbf{ST}}$ *(if* R_1 *and* R_2 *are two redexes,* $R_1\ R_2$ *can be written as* $([]\ R_2)[R_1]$ *or* $(R_1\ [])[R_2]$*).* □

Fact 7.2.15 (Pruning - Substitution)
For every well-typed term $\lambda x.M_1\ V$, *we have* $\mathsf{pr}^p_\Gamma(M_1\{V/x\}) = \mathsf{pr}^p_\Gamma(M_1)\{\mathsf{pr}^p_\Gamma(V)/x\}$.

Proof. *By induction on the typing judgement for* M_1, *the interesting case being* $M_1 = x$. *Two sub-cases can occur:*

- *Either* x *is not related to* p, *which means that* $\mathrm{reg}(T') < p$, *if* T' *is the type of* x *and thus* V, *whose effect is* 0 *and whose type is* T', *is not related to* p *neither. We have* $\mathsf{pr}^p_\Gamma(x) = \mathtt{V}_{T'}$ *and* $\mathsf{pr}^p_\Gamma(V) = \mathtt{V}_{T'}$. *Thus* $\mathsf{pr}^p_\Gamma(x)\{\mathsf{pr}^p_\Gamma(V)/x\} = \mathsf{pr}^p_\Gamma(x\{V/x\})$.
- *Or* x *is related to* p, *which means that* $\mathrm{reg}(T') \geq p$, *if* T' *is the type of* x *and thus* V, *whose type is* T', *is related to* p *too. We have* $\mathsf{pr}^p_\Gamma(x) = x$. *Thus* $\mathsf{pr}^p_\Gamma(x)\{\mathsf{pr}^p_\Gamma(V)/x\} = x\{\mathsf{pr}^p_\Gamma(V)/x\} = \mathsf{pr}^p_\Gamma(V) = \mathsf{pr}^p_\Gamma(x\{V/x\})$.

□

Simulation Result. As in the π-calculus case (Lemma 7.1.16), the pruning function enjoys simulation properties which allow us to deduce from an infinite computation a divergence involving pruned terms. As the definition of pruning is more involved in the present setting, this result is technically more difficult to obtain.

In order to reason on the transitions of pruned terms, the main point is to understand how pruning interacts with the decomposition of a term into an evaluation context and a redex (Definition 7.2.10 is extended to evaluation contexts in a natural way).

The lemma below explains how the pruning function is propagated within a term of the form $\mathbf{E}[M]$.

There are, intuitively, two possibilities, depending only on the context and the level of the pruning: either $\mathbf{E}$ is such that $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$ for all M, that is, pruning is always propagated in the hole to M, or the context is such that, if the effect of M is too small, the pruning inserts a generic value before

reaching the hole in $\mathbf{E}$, in which case $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[V]$, where $\mathbf{E}_1$ is an 'initial part' of $\mathbf{E}$, and this equality holds independently from M (as long as, like we said, the effect of M is sufficiently small in some sense).

Note that, in the latter case, if the effect of M is high, the pruning does not stop before reaching the hole of $\mathbf{E}$.

Lemma 7.2.16 (Pruning – Context effect)
If $\Gamma \vdash \mathbf{E} : (T, m)$, $\Gamma([\,]) = T_1$ and $\Gamma \vdash M_1 : (T_1, m_1)$, then $\Gamma \vdash \mathbf{E}[M_1] : (T, \max(m, m_1))$.

Proof. *By structural induction on $\mathbf{E}$:*

- *Case $[\,]$. We have $T = T_1$ and $m = 0$. As $\Gamma \vdash ([\,])[M_1] : (T_1, \max(m_1, 0))$ we conclude.*
- *Case $\mathbf{E}_2\ M$. We have $\Gamma \vdash M : (T', n)$, $\Gamma \vdash \mathbf{E}_2 : (T' \to^k T, m_2)$ and $m = \max(m_2, n, k)$. We use the induction hypothesis to get $\Gamma \vdash \mathbf{E}_2[M_1] : (T' \to^k T, \max(m_2, m_1))$. We use rule (**App**) to get $\Gamma \vdash \mathbf{E}_2[M_1]\ M : (T, \max(\max(m_1, m_2), n, k))$. We conclude, as $\mathbf{E}[M_1] = (\mathbf{E}_2[M_1]\ M)$ and $\max(\max(m_1, m_2), n, k) = \max(m, m_1)$.*
- *Case $\mathtt{deref}_n(\mathbf{E}_2)$. We have $\Gamma \vdash \mathbf{E}_2 : (T\ \mathtt{ref}_n, m_2)$ and $m = \max(m_2, n)$. We use the induction hypothesis to get $\Gamma \vdash \mathbf{E}_2[M_1] : (T\ \mathtt{ref}_n, \max(m_2, m_1))$. We use rule (**Drf**) to get $\Gamma \vdash \mathtt{deref}_n(\mathbf{E}_2[M_1]) : (T, \max(\max(m_2, m_1), n))$. We conclude, as $\mathbf{E}[M_1] = \mathtt{deref}_n(\mathbf{E}_2[M_1])$ and $\max(\max(m_2, m_1), n) = \max(m, m_1)$.*
- *Other cases are similar.*

□

Lemma 7.2.17 (Pruning – Context decomposition)
Consider a well-typed context $\mathbf{E}$ and fix a integer p. Then:

1. *Either for all well-typed process M, $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$*
2. *Or there exist $\mathbf{E}_1$ and $\mathbf{E}_2 \neq []$ s.t. $\mathbf{E} = \mathbf{E}_1[\mathbf{E}_2]$ and, for all M, we are in one of the two following cases:*
   (a) *If M has an effect $\geq p$, then $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$.*
   (b) *If M has an effect $< p$, then $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$ (where T'' is the type of $\mathbf{E}_2$).*

Proof. *By structural induction on $\mathbf{E}$, using the fact that $\Gamma \vdash \mathbf{E} : (T, m)$. We distinguish two cases:*

- *Either $\mathbf{E}$ is not related to T, which means $m < p$ and $\mathrm{reg}(T) < p$. We set $\mathbf{E}_1 = [\,]$ and $\mathbf{E}_2 = \mathbf{E}$. We have $\mathbf{E} = \mathbf{E}_1[\mathbf{E}_2]$ and T is the type of $\mathbf{E}_2$.*
  - *Suppose M has an effect $n < p$, then by Lemma 7.2.16, $\Gamma \vdash \mathbf{E}[M] : (T, \max(m, n))$. As $\mathrm{reg}(T) < p$ and $\max(m, n) < p$, $\mathbf{E}[M]$ is not related to p and $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathtt{V}_T$. We conclude, as we are in case 2b.*
  - *Suppose M has an effect $n \geq p$, then by Lemma 7.2.16, $\Gamma \vdash \mathbf{E}[M] : (T, \max(m, n))$. Thus $\mathbf{E}[M]$ is related to p. We discuss on the structure of $\mathbf{E}$:*
    * *If $\mathbf{E} = [\,]$ then $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(M)$. We conclude, as we are in case 2a.*
    * *If $\mathbf{E} = (\mathbf{E}_3\ M_3)$ we use the induction hypothesis. As M has an effect $\geq p$, we are either in case 1 or in case 2a. In both cases, $\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_3)[\mathsf{pr}^p_\Gamma(M)]$. As $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = (\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M])\ \mathsf{pr}^p_\Gamma(M_3))$ (remember $\mathbf{E}[M]$ is related to p) and $\mathsf{pr}^p_\Gamma(\mathbf{E}) = (\mathsf{pr}^p_\Gamma(\mathbf{E}_3)\ \mathsf{pr}^p_\Gamma(M_3))$, we conclude, as we are in case 2a.*

* *If* $\mathbf{E} = \mathtt{deref}_{n_3}(\mathbf{E}_3)$ *we use the induction hypothesis. As* M *has an effect* $\geq p$*, we are either in case 1 or in case 2a. In both cases,* $\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_3)[\mathsf{pr}^p_\Gamma(M)]$*. As* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(\mathbf{E}_3[M_3]))$ *(remember* $\mathbf{E}[M]$ *is related to* p*) and* $\mathsf{pr}^p_\Gamma(\mathbf{E}) = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(\mathbf{E}_3))$*, we conclude, as we are in case 2a.*
* *Other cases are similar.*

- *Or* $\mathbf{E}$ *is related to* p*. We discuss on the structure of* $\mathbf{E}$*:*
  - *Case* $[\,]$*. Then* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma([\,])[\mathsf{pr}^p_\Gamma(M)]$*. We are in case 1 and we conclude.*
  - *Case* $(\mathbf{E}_3\ M_3)$*. We use the induction hypothesis on* $\mathbf{E}_3$*.*
    * *Either we are in case 1, for any well-typed process* M*,* $\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_3)[\mathsf{pr}^p_\Gamma(M)]$*. Thus for any well-typed process* M*,* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_3[M])\ \mathsf{pr}^p_\Gamma(M_3)$ *(clearly,* $\mathbf{E}[M]$ *is related to* p*, as* $\mathbf{E}$ *is). As* $\mathsf{pr}^p_\Gamma(\mathbf{E}) = (\mathsf{pr}^p_\Gamma(\mathbf{E}_3)\ \mathsf{pr}^p_\Gamma(M_3))$*, we get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = (\mathsf{pr}^p_\Gamma(\mathbf{E}_3)[\mathsf{pr}^p_\Gamma(M)]\ \mathsf{pr}^p_\Gamma(M_3)) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$*. We are in case 1 and we conclude.*
    * *Or we get* $\mathbf{E}_{(1)}$ *and* $\mathbf{E}_2$ *s.t.* $\mathbf{E}_3 = \mathbf{E}_{(1)}[\mathbf{E}_2]$ *and the corresponding properties. We set* $\mathbf{E}_1 = (\mathbf{E}_{(1)}\ M_3)$*. Clearly,* $\mathbf{E}_1[\mathbf{E}_2] = \mathbf{E}$*.*
      · *Suppose* M *has an effect* $< p$*. Then* $\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_{(1)})[\mathtt{V}_{T''}]$ *where* T'' *is the type of* $\mathbf{E}_2$*. As* $\mathsf{pr}^p_\Gamma(\mathbf{E}_1) = (\mathsf{pr}^p_\Gamma(\mathbf{E}_{(1)})\ \mathsf{pr}^p_\Gamma(M_3))$*, we have* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = (\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M])\ \mathsf{pr}^p_\Gamma(M_3)) = \mathsf{pr}^p_\Gamma(\mathbf{E}_{(1)})[\mathtt{V}_{T''}]\ \mathsf{pr}^p_\Gamma(M_3) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$*. We are in case 2b and we conclude.*
      · *Suppose* M *has an effect* $\geq p$*. Then* $\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_3)[\mathsf{pr}^p_\Gamma(M)]$*. We have* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = (\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M])\ \mathsf{pr}^p_\Gamma(M_3)) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$*. We are in case 2a and we conclude.*
  - *Case* $\mathtt{deref}_{n_3}(\mathbf{E}_3)$*. We use the induction hypothesis on* $\mathbf{E}_3$*.*
    * *Either we are in case 1, for all well-typed process* M*,* $\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_3)[\mathsf{pr}^p_\Gamma(M)]$*. Thus for all well-typed process* M*,* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(\mathbf{E}_3[M]))$ *(clearly,* $\mathbf{E}[M]$ *is related to* p*, as* $\mathbf{E}$ *is). As* $\mathsf{pr}^p_\Gamma(\mathbf{E}) = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(\mathbf{E}_3))$*, we get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(\mathbf{E}_3)[\mathsf{pr}^p_\Gamma(M)]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$*. We are in case 1 and we conclude.*
    * *Or we get* $\mathbf{E}_{(1)}$ *and* $\mathbf{E}_2$ *s.t.* $\mathbf{E}_3 = \mathbf{E}_{(1)}[\mathbf{E}_2]$ *and the corresponding properties. We set* $\mathbf{E}_1 = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathbf{E}_{(1)})$*. Clearly,* $\mathbf{E}_1[\mathbf{E}_2] = \mathbf{E}$*.*
      · *Suppose* M *has an effect* $< p$*. Then* $\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_{(1)})[\mathtt{V}_{T''}]$ *where* T'' *is the type of* $\mathbf{E}_2$*. As* $\mathsf{pr}^p_\Gamma(\mathbf{E}_1) = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(\mathbf{E}_{(1)}))$*, we have* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(\mathbf{E}_3[M])) = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(\mathbf{E}_{(1)})[\mathtt{V}_{T''}]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$*. We are in case 2b and we conclude.*
      · *Suppose* M *has an effect* $\geq p$*. Then* $\mathsf{pr}^p_\Gamma(\mathbf{E}_3[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_3)[\mathsf{pr}^p_\Gamma(M)]$*. We have* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(\mathbf{E}_3[M])) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$*. We are in case 2a and we conclude.*
  - *Other cases are similar.*

□

The properties we now establish correspond to the situation, in the previous lemma, where M is an imperative redex acting at level p. By our typing rules, firing the redex yields a term which is not related to p via its effect: depending on the kind of imperative operator that is executed, it might either be related to p via its type, or not related to p at all (this appears more clearly in the proof of Lemma 7.2.20).

In such case, we are able to show that the pruned versions of the two terms are related by $\twoheadrightarrow^+$ (the standard reduction relation for $\lambda_{\mathbf{ST}}$, introduced in Section 2), which allows us to establish a simulation property.

Fact 7.2.18 (Pruning of contexts not related to p)
If $\mathbf{E}_2$ *is not related to* p*, then:*

1. *If* $\mathbf{E}_2 = (V_3\ \mathbf{E}_3)$ *then* V_3 *is not related to* p*.*
2. *If* $\mathbf{E}_2 = (\mathbf{E}_3\ M_3)$ *then* $\mathbf{E}_3$ *is not related to* p*.*

Proof. *Easily done by examining the rule* (**App**) *and the definition of being related to p.* □

Lemma 7.2.19 (Pruning – Reduction to a value)
If $\Gamma \vdash \mathbf{E}_2 : (T'', m)$ *and* $\mathbf{E}_2$ *is not related to p, for any well-typed* M, M'*,*

1. $\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,2)}\ \mathtt{V}_T\ M)] \twoheadrightarrow^+ \mathtt{V}_{T''}$*;*
2. $\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,3)}\ \mathtt{V}_T\ M\ M')] \twoheadrightarrow^+ \mathtt{V}_{T''}$*.*

Proof.

1. *By structural induction on* $\mathbf{E}$*,*
   - *Case* $[]$*. Then* $T'' = T$*.* $(\Pi^{(1,2)}\ \mathtt{V}_T\ N) \twoheadrightarrow\twoheadrightarrow \mathtt{V}_T = \mathtt{V}_{T''}$*.*
   - *Case* $\mathbf{E}_3\ M_3$*. Then* $\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)] = (\mathsf{pr}^p_\Gamma(\mathbf{E}_3)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)]\ \mathsf{pr}^p_\Gamma(M_3))$ *with* $\mathbf{E}_3$ *of type* $T_3 \to T''$*. Thanks to Fact 7.2.18, we can use the induction hypothesis on* $\mathbf{E}_3$ *and we get* $\mathsf{pr}^p_\Gamma(\mathbf{E}_3)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)] \twoheadrightarrow^+ \mathtt{V}_{T_3 \to T''}$*. By examining Definition 7.2.8, we get* $(\mathtt{V}_{T_3 \to T''}\ \mathsf{pr}^p_\Gamma(M_2)) \twoheadrightarrow \mathtt{V}_{T''}$ *and we conclude.*
   - *Case* $V_3\ \mathbf{E}_3$*. Then* $\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)] = (\mathsf{pr}^p_\Gamma(V_3)\ \mathsf{pr}^p_\Gamma(\mathbf{E}_3)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)])$ *with* $\mathbf{E}_3$ *of type* T_3*. We use Fact 7.2.18 to obtain that* V_3 *is not related to* K*. Thus* $\mathsf{pr}^p_\Gamma(V_3) = \mathtt{V}_{T_3 \to T''}$*. By examining Definition 7.2.8, we get* $(\mathtt{V}_{T_3 \to T''}\ \mathsf{pr}^p_\Gamma(\mathbf{E}_3)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)])) \twoheadrightarrow \mathtt{V}_{T''}$ *and we conclude.*
   - *Case* $\mathtt{deref}_{n_3}(\mathbf{E}_3)$*. Then* $\mathsf{pr}^p_\Gamma(\mathbf{E}_3)[(\Pi^{(1,2)}\ V_T\ N)] = (\Pi^{(1,2)}\ V_{T''}\ \mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,2)}\ V_T\ N)]) \twoheadrightarrow\twoheadrightarrow V_{T''}$*.*
   - *Other cases are similar.*
2. *The proof is similar.*

□

Lemmas 7.2.17 and 7.2.19 allow us to derive the desired simulation property for $\lambda_{\mathtt{ref}}$, which is similar in shape to Lemma 7.1.16; in particular, the main point is that a $\mapsto_p$ reduction is pruned into at least one reduction in the target calculus (case 4 below). Note however that we do not rely on behavioural equivalences here.

Lemma 7.2.20 (Simulation) *Consider* $p \in \mathbb{N}$ *and* $\Gamma \vdash M : (T, m)$*.*

1. *If* $(M, \delta) \mapsto^n_{\mathrm{I}} (M', \delta')$ *and* $n < p$*, then* $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$*.*
2. *If* $(M, \delta) \mapsto^p_{\mathrm{I}} (M', \delta')$*, then* $\mathsf{pr}^p_\Gamma(M) \twoheadrightarrow^+ \mathsf{pr}^p_\Gamma(M')$*.*
3. *If* $(M, \delta) \mapsto^n_{\mathrm{F}} (M', \delta')$ *and* $n < p$*, then* $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$*.*
4. *If* $(M, \delta) \mapsto^p_{\mathrm{F}} (M', \delta')$*, then* $\mathsf{pr}^p_\Gamma(M) \twoheadrightarrow \mathsf{pr}^p_\Gamma(M')$*.*

Proof.
The structure of the proof is as follows, for cases 1 *and* 2*, terms are decomposed the same way but the arguments invoked are different, in case* 1*, we use the definition of the pruning on terms not related to* p *to conclude, as in case* 2*, the pruning gives an "actual term" (not a generic value) and we use Lemma 7.2.19 to conclude. In both cases, the proof for rules* (**ref**) *and* (**deref**) *differ, as in the former case the more complex term appears before the reduction (we have* $\mathtt{ref}_n\ V$ *which reduces to* $u_{(n,T)}$*) whereas in the latter case the more complex term appears after the reduction (we have* $\mathtt{deref}_n(u_{(n,T)})$ *which reduces to* V*).*
Cases 3 *and* 4 *are treated the same way as cases* 1 *and* 2*, but only one rule of reduction* (β) *is examined.*

1. *By definition of the semantics, this reduction derivation is composed by an application of rule* (**context**) *and an application of rule* (**deref**)*,* (**store**) *or* (**ref**)*. Thus* $M = \mathbf{E}[R]$ *with* R *being a redex of type* T' *of store operation, dereferencing or reference.*

- *Case* (**ref**). *We have* $M = \mathbf{E}[\mathtt{ref}_n\ V]$, $M = \mathbf{E}[u_{(n,T')}]$ *and* $\delta' = \delta\langle u_{(n,T_0)} \rightsquigarrow V\rangle$ *with* $T' = T_0\ \mathtt{ref}_n$. *As we supposed that types are well-formed and* $n < p$, $T = T_0\ \mathtt{ref}_n$ *is not related to* p. *Moreover, the effect of* $\mathtt{ref}_n\ V$ *is* $n < p$ *and the effect of* $u_{(n,T_0)}$ *is* 0. *We use Lemma 7.2.17 on* $\mathbf{E}$*:*
  - *Either we are in case 1. We get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\mathtt{ref}_n\ V]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(\mathtt{ref}_n\ V)] = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathtt{V}_{T_0\ \mathtt{ref}_n}] = \mathsf{pr}^p_\Gamma(\mathbf{E})[\star]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[u_{(n,T_0)}]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\star]$. *As* $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$, *we conclude.*
  - *Or we are in case 2. We get* $\mathbf{E}_1$ *and* $\mathbf{E}_2$ *and their corresponding properties. Both terms* M *and* M' *correspond to case 2b and we get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\mathtt{ref}_n\ V]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[u_{(n,T_0)}]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$. *As* $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$, *we conclude.*
- *Case* (**deref**). *We have* $M = \mathbf{E}[\mathtt{deref}_n(u_{(n,T')})]$ *and* $M' = \mathbf{E}[V]$ *with* $\delta(u_{(n,T)}) = V$. *As we supposed that types are well-formed and* $n < p$, $T'\ \mathtt{ref}_n$ *is not related to* p. *Moreover, the effect of* $\mathtt{deref}_n(u_{(n,T')})$ *is* $n < p$ *and the effect of* V *is* 0. *We use Lemma 7.2.17 on* $\mathbf{E}$*:*
  - *Either we are in case 1. We get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\mathtt{deref}_n(u_{(n,T')})]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(\mathtt{deref}_)(u_{(n,T')})] = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathtt{V}_{T'}]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[V]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(V)] = \mathsf{pr}^p_\Gamma(conte)[\mathtt{V}_{T'}]$. *As* $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$, *we conclude.*
  - *Or we are in case 2. We get* $\mathbf{E}_1$ *and* $\mathbf{E}_2$ *and their corresponding properties. Both terms* M *and* M' *correspond to case 2b and we get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\mathtt{deref}_n(u_{(n,T')})]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[V]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$. *As* $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$, *we conclude.*
- *Case* (**store**) *is similar.*

2. *By definition of the semantics, this reduction derivation is composed by an application of rule* (**context**) *and an application of rule* (**deref**), (**store**) *or* (**ref**). *Thus* $M = \mathbf{E}[R]$, R *of type* T' *being a redex of store operation, dereferencing or reference.*

   - *Case* (**deref**). *We have* $M = \mathbf{E}[\mathtt{deref}_p(u_{(p,T')})]$ *and* $M' = \mathbf{E}[V]$ *with* $\delta(u_{(p,T')}) = V$. *We remark that* $\mathtt{deref}_p(u_{(p,T')})$ *has an effect* $\geq p$ *but* V *(whose effect is* 0*) has not. Moreover, as type* $T'\ \mathtt{ref}_p$ *is well-formed,* V *is not related to* p. *We use Lemma 7.2.17 on* $\mathbf{E}$*:*
     - *Either we are in case 1. We get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\mathtt{deref}_p(u_{(p,T')})]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(\mathtt{deref}_p(u_{(p,T')}))] = \mathsf{pr}^p_\Gamma(\mathbf{E})[\Pi^{(1,2)}\ \mathtt{V}_{T'}\ \mathsf{pr}^p_\Gamma(V)]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[V]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(V)] = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathtt{V}_{T'}]$. *We use Fact 7.2.14 to get* $\mathsf{pr}^p_\Gamma(M) \twoheadrightarrow\twoheadrightarrow \mathsf{pr}^p_\Gamma(M')$ *and conclude.*
     - *Or we are in case 2. We get* $\mathbf{E}_1$ *and* $\mathbf{E}_2$ *s.t.* (*i*) $\mathbf{E} = \mathbf{E}_1[\mathbf{E}_2]$ *and* (*ii*) $\mathbf{E}_2$ *is not related to* p. *Term* M *corresponds to case 2a and term* M' *corresponds to case 2b: we get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\mathtt{deref}_p(u_{(p,T')})]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(\mathtt{deref}_)(u_{(p,T')}p)] = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[\Pi^{(1,2)}\ \mathtt{V}_{T'}\ \star]]$ *(using* (*i*)*) and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[V]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$ *(using* (*ii*)*). We use Fact 7.2.14 and Lemma 7.2.19 to get* $\mathsf{pr}^p_\Gamma(M) \twoheadrightarrow^* \mathsf{pr}^p_\Gamma(M')$ *and we conclude.*
   - *Case* (**ref**). *We have* $M = \mathbf{E}[\mathtt{ref}_p\ V]$ *and* $M' = \mathbf{E}[u_{(p,T_0)}]$ *with* $T' = T_0\ \mathtt{ref}_p$. *Term* $\mathtt{ref}_p\ V$ *has effect* p *(and thus, is related to* p*) but* $u_{(p,T_0)}$ *(whose effect is* 0*) has not. Moreover,* $u_{(p,T_0)}$*, whose type is* $T_0\ \mathtt{ref}_p$*, is related to* p. *We use Lemma 7.2.17 on* $\mathbf{E}$*:*
     - *Either we are in case 1. We get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\mathtt{ref}_p\ V]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(\mathtt{ref}_n\ p)] = \mathsf{pr}^p_\Gamma(\mathbf{E})[\Pi^{(1,2)} \star \mathsf{pr}^p_\Gamma(V)]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[u_{(p,T_0)}]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\star]$. *We use Fact 7.2.14 to get* $\mathsf{pr}^p_\Gamma(M) \twoheadrightarrow\twoheadrightarrow \mathsf{pr}^p_\Gamma(M')$ *and conclude.*
     - *Or we are in case 2. We get* $\mathbf{E}_1$ *and* $\mathbf{E}_2$ *s.t.* (*i*) $\mathbf{E} = \mathbf{E}_1[\mathbf{E}_2]$ *and* (*ii*) $\mathbf{E}_2$ *is not related to* p. *Term* M *corresponds to case 2a and term* M' *corresponds to case 2b: we get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\mathtt{ref}_p\ V]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(\mathtt{ref}_p\ V)] = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[\Pi^{(1,2)}\ \star\ \mathsf{pr}^p_\Gamma(V)]]$ *(using* (*i*)*) and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[V]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$ *(using* (*ii*)*). We use Fact 7.2.14 and Lemma 7.2.19 to get* $\mathsf{pr}^p_\Gamma(M) \twoheadrightarrow^* \mathsf{pr}^p_\Gamma(M')$ *and we conclude.*
   - *Case* (**store**) *is similar.*

3. *By definition of the semantics, this reduction derivation is composed by an application of rule* (**context**) *and an application of rule* (β). *Thus* $M = \mathbf{E}[\lambda x.M_1\ V]$, $M' = \mathbf{E}[M_1\{V/x\}]$. *The term* $\lambda x.M_1$ *has effect* 0 *and type* $T_2 \to^n T'$ *(for some* T_2 *and* $n < p$*) and the term* V *has type* T_2 *and effect* 0*; by rule*

(**App**), $\lambda x.M_1\ V$ *has effect* $n < p$, *by rule* (**Abs**), M_1 *has effect* $n < K$ *and by Lemma 7.2.5,* $M_1\{V/x\}$ *has effect* $n < K$. *Thus, as type* $T_2 \to^n T'$ *is well-formed, neither* $\lambda x.M_1\ V$, *nor* $M_1\{V/x\}$ *are related to* K *(notice that* V *could). We use Lemma 7.2.17 on* $\mathbf{E}$*:*

- *Either we are in case 1. We get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\lambda x.M_1\ V]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(\lambda x.M_1\ V)] = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathtt{V}_{T'}]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M_1\{V/x\}]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M_1\{V/x\})] = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathtt{V}_{T'}]$. *We have* $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$ *and we conclude.*
- *Or we are in case 2. We get* $\mathbf{E}_1$ *and* $\mathbf{E}_2$ *and their corresponding properties. Both terms* M *and* M' *correspond to case 2b and we get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\lambda x.M_1\ V]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M_1\{V/x\}]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$. *As* $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$, *we conclude.*

4. *By definition of the semantics, this reduction derivation is composed by an application of rule* (**context**) *and an application of rule* (β). *Thus* $M = \mathbf{E}[\lambda x.M_1\ V]$, $M' = \mathbf{E}[M_1\{V/x\}]$. *The term* $\lambda x.M_1$ *has effect* 0 *and type* $T_2 \to^p T'$ *(for some* T_2*) and the term* V *has type* T_2 *and effect* 0. *By rule App,* $\lambda x : T_2.M_1\ V$ *has effect* p *(and thus is related to* p*), by rule* (**Abs**), M_1 *has effect* p *and by Lemma 7.2.5,* $M_1\{V/x\}$ *has effect* p *(and thus is related to* p*). We use Lemma 7.2.17:*

   - *Either we are in case 1. We get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\lambda x.M_1\ V]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(\lambda x.M_1\ V)] = \mathsf{pr}^p_\Gamma(\mathbf{E})[\lambda x.\mathsf{pr}^p_\Gamma(M_1)\ \mathsf{pr}^p_\Gamma(V)]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M_1\{V/x\}]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M_1\{V/x\})]$. *We use Fact 7.2.15 and Fact 7.2.14 to get* $\mathsf{pr}^p_\Gamma(M) \twoheadrightarrow \mathsf{pr}^p_\Gamma(M')$ *and we conclude.*
   - *Or we are in case 2. We get* $\mathbf{E}_1$ *and* $\mathbf{E}_2$ *and their corresponding properties. Both terms* M *and* M' *correspond to case 2a and we get* $\mathsf{pr}^p_\Gamma(\mathbf{E}[\lambda x.M_1\ V]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(\lambda x.M_1\ V)] = \mathsf{pr}^p_\Gamma(\mathbf{E})[\lambda x.\mathsf{pr}^p_\Gamma(M_1)\ \mathsf{pr}^p_\Gamma(V)]$ *and* $\mathsf{pr}^p_\Gamma(\mathbf{E}[M_1\{V/x\}]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M_1\{V/x\})]$. *We use Fact 7.2.15 and Fact 7.2.14 to get* $\mathsf{pr}^p_\Gamma(M) \twoheadrightarrow \mathsf{pr}^p_\Gamma(M')$ *and we conclude.*

□

We rely on termination of $\lambda_{\mathbf{ST}}$ to obtain the soundness of our system.

As we want to be able to state that there is an infinite number of functional reductions on level p, we want to exhibit a measure that decreases with each imperative reduction on level p and does not increase with each reduction on level $< p$, as we did in the π-calculus case. We use the active imperative operators, which are the imperative operators (references, dereferencings and assignments), that does not occur under a λ.

Definition 7.2.21 (Active imperative operators) *The number of active imperative operators at level* p *in* M, *written* $\mathbf{Os}^M()$ *is defined on typed terms as follows:*

$$\mathbf{Os}^p(x) = \mathbf{Os}^p(\lambda x.M) = \mathbf{Os}^p(u_{(n,T)}) = 0 \qquad \mathbf{Os}^p(M\ N) = \mathbf{Os}^p(M) + \mathbf{Os}^p(N)$$

$$\begin{array}{lll} \mathbf{Os}^p(\mathtt{deref}_n(M)) = & \mathbf{Os}^p(\mathtt{ref}_n\ M) = & \mathbf{Os}^p(M) \qquad \text{if } n \neq p \\ \mathbf{Os}^p(\mathtt{deref}_p(M)) = & \mathbf{Os}^p(\mathtt{ref}_p\ M) = & 1 + \mathbf{Os}^p(M) \end{array}$$

$$\begin{array}{ll} \mathbf{Os}^p(M :=_n N) = & \mathbf{Os}^p(M) + \mathbf{Os}^p(N) \qquad \text{if } n \neq p \\ \mathbf{Os}^p(M :=_p N) = & 1 + \mathbf{Os}^p(M) + \mathbf{Os}^p(N) \end{array}$$

We extend this definition to evaluation contexts with $\mathbf{Os}^p([\,]) = 0$.

Fact 7.2.22 (Active imperative operators in contexts)
We have $\mathbf{Os}^p(\mathbf{E}[M]) = \mathbf{Os}^p(\mathbf{E}) + \mathbf{Os}^p(M)$.

Proof.
By structural induction on contexts.

□

The following lemma relates the measure we just defined with our type system, stating that the effect of a term is bound by the maximum level of active imperative operators found inside this term.

Lemma 7.2.23 (Effect and active imperative operators)
If $\Gamma \vdash M : (T, m)$ *and* $m < p$ *then* $\mathbf{Os}^p(M) = 0$.

Proof. *By induction on the typing judgement* $\Gamma \vdash M : (T, m)$.

- *Case* (**App**). *We have* $M = M_1\ M_2$ *with* M_1 *of type* $m_1 \leq m < p$ *and* M_2 *of type* $m_2 \leq m < p$. *Thus, we are able to use the induction hypothesis to get* $\mathbf{Os}^p(M) = \mathbf{Os}^p(M_1) + \mathbf{Os}^p(M_2) = 0 + 0$.
- *Case* (**Ref**). *We have* $M = \mathtt{ref}_{m_1}\ M_1$. *Typing gives* $m_1 \leq m < p$. *We use the induction hypothesis and conclude, as* $\mathbf{Os}^p(M) = \mathbf{Os}^p(M_1) = 0$.
- *Cases* (**Asg**) *and* (**Drf**) *are similar.*
- *Cases* (**Add**), (**Var**), (**Uni**) *and* (**Abs**) *are easy, as every value* V *has effect* 0 *and* $\mathbf{Os}^K(V) = 0$.

□

We are finally able to state that active imperative operators yield the measure we need.

Lemma 7.2.24 (Active imperative operators decreasing)
If $\Gamma \vdash M : (T, m)$ *then*

1. *if* $(M, \delta) \mapsto_{\mathrm{F}}^n (M', \delta')$ *with* $n < p$ *then* $\mathbf{Os}^p(M') \leq \mathbf{Os}^p(M)$,
2. *if* $(M, \delta) \mapsto_{\mathrm{I}}^n (M', \delta')$ *with* $n < p$ *then* $\mathbf{Os}^p(M') \leq \mathbf{Os}^p(M)$,
3. *and if* $(M, \delta) \mapsto_{\mathrm{I}}^p (M', \delta')$ *then* $\mathbf{Os}^p(M') < \mathbf{Os}^p(M)$.

Proof.

1. *By definition of the semantics, this reduction derivation is composed by an application of rule* (**context**) *and an application of rule* (β). *Thus* $M = \mathbf{E}[\lambda x.M_1\ V]$, $M' = \mathbf{E}[M_1\{V/x\}]$. *The term* $\lambda x.M_1$ *has effect* 0 *and type* $T_2 \to^n T'$ *(for some* T_2 *and* $n < p$*) and the term* V *has type* T_2 *and effect* 0*; by rule* (**App**), $\lambda x.M_1\ V$ *has effect* $n < p$, *by rule* (**Abs**), M_1 *has effect* $n < K$ *and by Lemma 7.2.5,* $M_1\{V/x\}$ *has effect* $n < K$. *We use Lemma 7.2.23 and Fact 7.2.22 to get* $\mathbf{Os}^p(M) = \mathbf{Os}^p(\mathbf{E}) + 0$ *and we use again Lemma 7.2.23 and Fact 7.2.22 to get* $\mathbf{Os}^p(M') = \mathbf{Os}^p(\mathbf{E}) + 0$. *We conclude.*
2. *By definition of the semantics, this reduction derivation is composed by an application of rule* (**context**) *and an application of rule* (**deref**), (**store**) *or* (**ref**). *Thus* $M = \mathbf{E}[R]$, R *being a redex of type* T' *of store operation, dereferencing or reference.*
   - *Case* (**ref**). *We have* $M = \mathbf{E}[\mathtt{ref}_n\ V]$, $M = \mathbf{E}[u_{(n,T')}]$ *and* $\delta' = \delta\langle u_{(n,T_0)} \rightsquigarrow V\rangle$ *with* $T' = T_0\ \mathtt{ref}_n$. *The effect of* $\mathtt{ref}_n\ V$ *is* $n < p$ *and the effect of* $u_{(n,T_0)}$ *is* 0. *We use Lemma 7.2.23 and Fact 7.2.22 to get* $\mathbf{Os}^p(M) = \mathbf{Os}^p(\mathbf{E}) + 0$ *and we use again Lemma 7.2.23 and Fact 7.2.22 to get* $\mathbf{Os}^p(M') = \mathbf{Os}^p(\mathbf{E}) + 0$. *We conclude.*
   - *Case* (**deref**). *We have* $M = \mathbf{E}[\mathtt{deref}_n(u_{(n,T')})]$ *and* $M' = \mathbf{E}[V]$ *with* $\delta(u_{(n,T)}) = V$. *The effect of* $\mathtt{deref}_n(u_{(n,T')})$ *is* $n < p$ *and the effect of* V *is* 0. *We use Lemma 7.2.23 and Fact 7.2.22 to get* $\mathbf{Os}^p(M) = \mathbf{Os}^p(\mathbf{E}) + 0$ *and we use again Lemma 7.2.23 and Fact 7.2.22 to get* $\mathbf{Os}^p(M') = \mathbf{Os}^p(\mathbf{E}) + 0$. *We conclude.*
   - *Case* (**store**) *is similar.*
3. *By definition of the semantics, this reduction derivation is composed by an application of rule* (**context**) *and an application of rule* (**deref**), (**store**) *or* (**ref**). *Thus* $M = \mathbf{E}[R]$ *with* R *of type* T' *being a redex of store operation, dereferencing or reference.*

- *Case* (**deref**). *We have* $M = \mathbf{E}[\mathtt{deref}_p(u_{(p,T')})]$ *and* $M' = \mathbf{E}[V]$ *with* $\delta(u_{(p,T')}) = V$. *We remark that* $\mathtt{deref}_p(u_{(p,T')})$ *has an effect* $\geq p$ *but* V *(whose effect is* 0*) has not. By definition* $\mathbf{Os}^p(\mathtt{deref}_p(u_{(p,T')})) = 1 + 0$ *and by Lemma 7.2.23* $\mathbf{Os}^p(V) = 0$. *We use twice Fact 7.2.22 to conclude* $\mathbf{Os}^p(M) = \mathbf{Os}^p(\mathbf{E}) + 1 < \mathbf{Os}^p(M') = \mathbf{Os}^p(\mathbf{E}) + 0$.
- *Case* (**ref**). *We have* $M = \mathbf{E}[\mathtt{ref}_p\ V]$ *and* $M' = \mathbf{E}[u_{(p,T_0)}]$ *with* $T' = T_0\ \mathtt{ref}_p$. *Term* $\mathtt{ref}_p\ V$ *has effect* p *(and thus, is related to* p*) but* $u_{(p,T_0)}$ *(whose effect is* 0*) has not. By definition* $\mathbf{Os}^p(\mathtt{ref}_p\ V) = 1 + 0$ *and by Lemma 7.2.23* $\mathbf{Os}^p(u_{(p,T_0)}) = 0$. *We use twice Fact 7.2.22 to conclude* $\mathbf{Os}^p(M) = \mathbf{Os}^p(\mathbf{E}) + 1 < \mathbf{Os}^p(M') = \mathbf{Os}^p(\mathbf{E}) + 0$.
- *Case* (**store**) *is similar.*

□

The following lemma is the counterpart of Lemma 7.1.22, stating that there exists a maximum level on which an infinite number of reductions takes place. With the previous lemma, we are able to state more precisely that, at this level, an infinite number of *functional* reductions take place.

Lemma 7.2.25 (Maximum Interesting Level) *Suppose that* $\Gamma \vdash M : (T, l)$*, and that there exists* $(M_i)_{i\in\mathbb{N}}$*, an infinite reduction sequence starting from* M*. Then*

1. *for all* i, M_i *is typable.*
2. *There exist* p *and* i_o *s.t.*
   (a) *if* $i > i_0$ *and* $(M_i, \delta_i) \mapsto^n_{\mathrm{I}} (M_{i+1}, \delta_{i+1})$ *then* $n \leq p$,
   (b) *if* $i > i_0$ *and* $(M_i, \delta_i) \mapsto^n_{\mathrm{F}} (M_{i+1}, \delta_{i+1})$ *then* $n \leq p$,
   (c) *and there exists an infinite set of indexes* $\mathcal{I}$ *s.t. for each* $i \in \mathcal{I}$ *either* $(M_i, \delta_i) \mapsto^p_{\mathrm{F}} (M_{i+1}, \delta_{i+1})$ *or* $(M_i, \delta_i) \mapsto^p_{\mathrm{I}} (M_{i+1}, \delta_{i+1})$.
   (d) *There are infinitely many* $i \in \mathcal{I}$ *s.t.* $(M_i, \delta_i) \mapsto^p_{\mathrm{F}} (M_{i+1}, \delta_{i+1})$.

Proof.

1. *Follows by Proposition 7.2.6.*
2. *It is easy to find* p *satisfying 2a, 2b and 2c, as the set of levels on which an infinite number of reductions take place is finite and thus, admits a maximum. Lemma 7.2.24 ensures that 2d holds. Consider such a* p *and suppose, toward a contradiction, that 2d does not hold, that is, there exists an index* j *s.t. for every* $i > j$*, either* $(M_i, \delta_i) \mapsto^n_{\mathrm{F}} (M_{i+1}, \delta_{i+1})$ *with* $n < p$*, or* $(M_i, \delta_i) \mapsto^n_{\mathrm{I}} (M_{i+1}, \delta_{i+1})$ *with* $n < p$*, or* $(M_i, \delta_i) \mapsto^p_{\mathrm{I}} (M_{i+1}, \delta_{i+1})$. *We use Lemma 7.2.24 to show that the sequence* $(\mathbf{Os}^p(M_i))_{i>j}$ *is decreasing. Moreover, as 2c holds, there is an infinite number of* i *s.t.* $M_i \to^p_{\mathrm{I}} M_{i+1}$. *Thus, we use Lemma 7.2.24 to deduce that the sequence* $(\mathbf{Os}^p(M_i))_{i>j}$ *strictly decreases an infinite number of times. This yields a contradiction, as* $>$ *is well-founded.*

□

It may be noticed that in this call-by-value setting, the well-formedness of types is not used. Even if this property does not appear in the crucial simulation Lemma, it was used in Lemma 7.2.20 and it is required to apply Lemma 7.2.17

As in Section 7.1.1, we put together Lemmas 7.2.20 and 7.2.25 to prove soundness.

Theorem 7.2.26 (Soundness) *If* $\Gamma \vdash M : (T, m)$ *then* M *terminates.*

Proof. *Consider, by absurd, an infinite computation* $\{M_i\}_i$ *starting from* $M = M_0$. *By Lemma 7.2.25, all the* M_i*'s are well-typed, and there is a maximal* p *s.t. for infinitely many* i, $(M_i, \delta_i) \mapsto^p_{\mathrm{F}} (M_{i+1}, \delta_{i+1})$ *and an index* i_0 *exists such that every reduction on an index greater than* i_0 *is performed at level* $n \leq p$. *Consider*

the sequence $(\mathsf{pr}^p_\Gamma(M_i))_{i>i_0}$. *By Lemma 7.2.20, we obtain that for every* $i > i_0$, *either* $\mathsf{pr}^p_\Gamma(M_i) \twoheadrightarrow^* \mathsf{pr}^p_\Gamma(M_{i+1})$. *Moreover,* $\mathsf{pr}^p_\Gamma(M_i) \twoheadrightarrow^+ \mathsf{pr}^p_\Gamma(M_{i+1})$ *for an infinite number of* i. *Thus* $\mathsf{pr}^p_\Gamma(M_{i_0})$ *is diverging. This contradicts Theorem 2.4.1 and Lemma 7.2.20.* □

This ends the termination proof of $\lambda_{\mathtt{ref}}$, we discuss in the following the possible application of this proof technique to some extensions of this calculus.

Parametricity and extensions As in the π-calculus, so in the λ-calculus the method is parametric with respect to a terminating pure functional core and its termination proof. Other core calculi could be considered: as the simply typed discipline is enforced by our type system, the functional calculus has to be a subset of the simply typed terms. We believe that it is possible to extend our work to polymorphic types, although this extension is not trivial, as it might involve polymorphism at the level of regions: for instance, we have to check that a type like $(\forall A.A \to^0 A)\ \mathtt{ref}_n$ cannot have its A component instantiated with a type containing a level strictly greater than n.

Moreover, as in the π-calculus, the method could be extended and refined. A natural extension is to consider more sophisticated type systems, for example to imagine a counterpart to the "input sequence" analysis we performed in the π-calculus case (see Section 7.1.4).

Remark 7.2.27 (Introduction of an `if then else` construct)
Some remarks can be made regarding the treatment of conditional branching using our pruning technique. More precisely, in an extension of λ_{ref} *containing the construct* `if then else` , *when pruning* $\mathtt{if}\ \mathtt{deref}_n(u_{(n,\mathtt{B})})\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2$ *(whose counterpart, in the* π*-calculus case, could be the process* $\mathtt{if}\ a = x\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$*), we forget the potential value of* $\mathtt{deref}_n(u_{(n,T)})$ *(resp. the names* a *and* x *are forgotten by the pruning). Thus it is legitimate to ask if the pruned term can "be mistaken" and choose the wrong branch, breaking the simulation property. For instance, suppose that one of the branch perform an infinite computation at level* p, *one can wonder if the pruned process is also diverging, or if we pruned the conditional branching the "wrong way' and forget the divergence.*

One has to analyse the encoding of the conditional in λ, *in order to be convinced that our system accommodate conditional branching:* $[\mathtt{if}\ B\ \mathtt{then}\ M\ \mathtt{else}\ N] = (B\ \lambda x.M\ \lambda x.N)\ \star$ *with the encodings for values:* $[\mathtt{true}] = \lambda xy.x$ *and* $[\mathtt{false}] = \lambda xy.y$. *Adding abstractions and* $\star$ *prevents the call-by-value strategy from reducing the branch of the conditional before the condition.*

The reasoning which proves that $\mathtt{if}\ \mathtt{deref}_n(u_{(n,T)})\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2$ *is handled correctly by our system is as follows: if one of the branch is divergent, following the conditions of the proof, this branch will lead to an infinite number of reductions of level* p. *Thus the condition* $\mathtt{deref}_n(u_{(n,T)})$ *is a function which take the branches as arguments, and, as a consequence, has to be a function of at least level* p, *thus it cannot be stored at level strictly smaller than* $p+1$. *As a consequence, when we take as a starting point the index in the reduction sequence where there is no longer reduction on level greater than* p, *we place ourselves beyond the point where the* `if then else` *has been reduced.*

Chapter 8

Conclusion and future works

8.1 Summary of the contributions of this thesis

New weight-based methods We have presented in this thesis several existing type systems ensuring termination in concurrent languages, along with new proof techniques. Taking [DS06] as a starting point, we have explored some refinements increasing the expressiveness of these systems (Section 3).

Termination in higher-order calculi Not only we presented in Section 4.1 how to adapt the methods mentioned above to analyse termination higher-order (process-passing) calculi, we also relate such direct analyses to the termination techniques obtained through the encoding from the higher-order setting to the name-passing one.

An analysis of the inference problem We have studied the problem of inference of these weight-based type systems in Section 5, and explained why some features, initially introduced to extend the expressiveness of our systems, lead to NP-complete inference procedure. In this section, we also developed new type systems, with comparable expressiveness, but whose inference can be efficiently performed.

A termination method and its associated proof technique After briefly presenting the existing proof techniques for termination using logical relations, and why we need to use such techniques. We developed a new proof technique successfully combining this approach with the weight-based methods, and we applied it to two different setting: a π-calculus in which some services are distinguished as functional, and a λ-calculus with references.

8.2 Future Works

This last section is dedicated to the study of the possible ways of using the contributions of this thesis to obtain more interesting results. We present here several ideas stemming from the ones presented in this document. Most of the issues discussed here have not been thoroughly studied and are treated only as possibilities of future works.

Complexity in time After studying termination of programs, one is often willing to explore the issue further and study their complexity. Indeed, programmers developing concurrent applications do not only want them to terminate, but more precisely to terminate as fast as possible. Complexity is a well-studied notion in the sequential setting, but concurrency theory, and especially the domain of process algebras lacks a well-established definition of "concurrent complexity" (we have to mentioned however the one found in [AD07]). "What does it mean for a process to be polynomial ?" and "with respect to which quantity should

the complexity be computed ?" are crucial questions. The size of the process seems an obvious answer to the second one, but when defining services waiting for requests, it seems more natural to take into account the size of the first-order values received in the request that starts the computation. We have to decide whether the complexity of a service is computed independently from a environment, without other computations being reduced simultaneously or if scheduling and mutual exclusion issues has to be taken into account.

One way to ensure that programs run in a time bounded by a function in the size of their arguments is *implicit complexity*, which consists in constraining the syntax of the programming language in such a way that every definable program belongs to a given complexity class (see for instance [LM93], [BC92] or [BGM10]). Fruitful results can be obtained by relating implicit complexity, at least for the sequential setting, and proof theory (especially by using the linear logics [LS10]). We believe that it is possible to define implicit complexity in the context of the π-calculus, that is, to design a fragment of π-calculus allowing only the creation of services whose complexity in terms of the number of reductions is constrained by a function depending on the size of the request starting a computation. By performing fine analyses of the replication operators, and allowing the presence of recursive outputs only when their integer arguments are strictly smaller than the integer received, one is able to define a language in which every request to a typed service spawns a computation which is executed in a number of reductions polynomial (for instance) in the size of the argument of the request.

Indeed, when ensuring that, in a service $!p(n, \widetilde{x}).P$, the body P contains only one output on p, that this output is a request on an integer strictly smaller than n (for instance, $\overline{p}\langle n-1, \widetilde{v}\rangle$), and that the other calls to functions in P are also controlled in a way similar to the system presented in Section 3.1, we can prove that the service offered by p terminates in time polynomial with respect to n. Similarly, if we lift the condition on the number of recursive calls in p by allowing more than one output on p in P (of course, we need to ensure that every output on p is performed on strictly smaller arguments), we are able to prove that the service offered by p terminates in elementary time.

Yet, anyone willing to define complexity in a concurrent setting has to answer the question of whether the interactions of the environment in which the process is executed should be taken into account. More precisely, we should decide if, when computing the time it takes for a request to be completed, reductions originating from other requests shall be taken into account. For instance, suppose that two clients $\overline{p}\langle 2, r_1\rangle.r_1(x_1).P_1$ and $\overline{p}\langle 10000, r_2\rangle.r_2(x_2).P_2$ are put in parallel with a server $!p(x, r).P$ computing a function on the argument x and returning the result on channel r. Hopefully, the first request takes less time for being completed that the second one. Yet, we can imagine an execution in which the server begins to treat the first request, then executes the second one entirely, and afterwards, finishes the first one. The total time the first client P_1 has waited for an answer is huge, even if the argument of its request was small. As a consequence, it seems legitimate to define complexity with respect to causality, that is, defining it in such a way that the reductions which are taken into account are only the ones which causally depend from a unique request. In this case, the computation triggered by $\overline{p}\langle 10000, r_2\rangle$ will be ignored when computing the time taken to answer the request $\overline{p}\langle 2, r_1\rangle.r_1(x_1)$. This notion of concurrent complexity can be safely relied upon, provided the process is executed with a fair scheduler (meaning that each reduction is given an equal chance to be performed) and that the number of requests sent in parallel is known. This would allow us to build services offering guarantees such as "There are X other requests being currently executed by our server, as a consequence, completing your request of size Y will take $f(X, Y)$ seconds".

Yet, this notion of complexity, which separates the different requests with respect to causality, can not fully capture concurrent behaviours: indeed, in this setting, each request is treated in parallel, but requests cannot interact with each other (as it would definitely make the analysis difficult), causality ensures that the computations originating from different requests are independent. One can expect to build an analysis that somehow takes this notion into account. For instance, imagine a server able to handle any number of requests simultaneously, but each computation requires the use of a crucial resource. As a consequence, answering to a request spawns a computation which contains a critical section. We are willing to take into account this critical section, as it lets several requests being executed in parallel become dependent of each other: clients have to wait that the resource goes back to an available state. We believe that it is possible to design complex analyses able to handle such situations.

Complexity in space The issue of complexity in the number of reductions leads to another natural issue which is the complexity in space (studied in [AM02] for instance). In the sequential calculi, the two notions (time and space complexities) are strongly related. Yet, not only can someone ask for a program to be executed in the smallest space possible (taking as little resources as possible) but one can also ask for programs which can be executed in a bounded space. This property is very useful for people developing software used on smaller platforms, such as mobile phones or embedded systems. In the concurrent case, one has first to decide what is "space". In the case of the π-calculi, we can consider the space taken by a computation as either the maximum size (in the number of symbols) an intermediate process can take during the computation, or the total number of names created by the computation, i.e. the number of restrictions (νc) inside the body of replications released by communications. For instance the process $!a(x).((\nu v_i)\overline{a}\langle v_i\rangle) \mid \overline{a}\langle v_1\rangle$ diverges by creating an infinite number of new names v_i whereas the process $!a(x).\overline{a}\langle x\rangle \mid \overline{a}\langle v_1\rangle$ behaves similarly but does not create new names at each reduction step. We believe that type systems, following the ideas presented in this thesis, can be built in order to achieve this goal. Yet, the task is challenging as one has to take into account the "garbage-collection" process. For instance, in the case of P_1, as long as the output $\overline{a}\langle v_i\rangle$ is consumed, the restricted name v_i no longer appears in the prefixes of the process. Should this process be implemented, the same memory cell could be used to contain, in turn, all the names v_i. As a consequence, one has to find way to discard unused restricted names by using, for instance, a transformation "$(\nu\ c)\ P \mapsto P$ if c does not appear inside P".

Termination of probabilistic concurrent system An interesting extension to the analyses presented in this document is how they can be applied to a concurrent setting with explicit probabilities, for instance the stochastic π-calculus (see [Pri95] or some other probabilistic extensions of the π-calculus (like the one in [NPP09]) (see also [VY07] and [LSV07]). In our works, we always suppose that we have "simple" non-determinism: if several reductions are possible, we do not give priority to one in particular. We could refine our type systems to accommodate calculi in which non-determinism is refined with the use of a probability measure, by giving each possible reduction an explicit probability to be chosen by the scheduler. There exist several ways to perform this task, for instance, one way to doing it is adding to the syntax an explicit probabilistic choice operator $\oplus_p$ and let the semantics be defined by a probabilistic automaton (see [BCP08]). Another way, without altering the syntax, in supposing that each possible reduction has the same probability to be chosen. For instance consider the process:

$$P_1 = !a(x).((\nu v)\ \overline{a}\langle v\rangle) \mid \overline{a}\langle v_1\rangle \mid a(y).\overline{b}$$

Two reductions can occur, either the output $\overline{a}\langle v_1\rangle$ communicates with the unreplicated input $a(y)$, in this case we get the process $P_1^0 = !a(x).((\nu v)\ \overline{a}\langle c\rangle) \mid \overline{b}$ which cannot be reduced further, or it communicates with the replicated input $!a(x)$ and we get the process $P_2 = (\nu v_2)\ (!a(x).((\nu v)\ \overline{a}\langle c\rangle) \mid \overline{a}\langle v_2\rangle \mid a(y).\overline{b})$ which is still able to perform reductions on a. Notice that P_1 is not terminating: indeed, if at each step, the communication with the replicated input is chosen, we consume an input on a to get a new output on a. Now suppose that the property mentioned above is ensured, that is, if several reductions are possible, each one has an equal probability of being chosen. Here it means that P_1 reduces to P_2 with probability $1/2$ and to P_1^0 with probability $1/2$. We easily notice that, in turn, P_2 has a probability $1/2$ to reduce to some process P_3, still able to perform a communication on a and probability $1/2$ to reduce to some process P_2^0 not able to be reduced further. As a consequence, one can conclude that the measure of probability associated to the event "P_1 terminates" is $\Sigma_{k\geq 1} 2^{-k} = 1$, that is, in other terms, P_1 *almost surely terminates*. This notion can lead to interesting development, and the type systems we present here can be transformed to check this property. We also believe that our study of termination of probabilistic concurrent systems can be extended to accommodate explicit probabilistic operators in the syntax of processes. For example, one can imagine studying the termination of a π-calculus in which we have a non-deterministic operator $\texttt{choose}(P_1, P_2, p)$ associated to the semantics "$\texttt{choose}(P_1, P_2, p)$ reduces to P_1 with probability p and to P_2 with probability $1 - p$".

Termination of polymorphic concurrent systems Another possible direction which can be taken in order to get fruitful results from the contributions of this thesis is applying the methods presented here to calculi featuring polymorphism. Polymorphism for π-calculus has been studied in [Tur96] and [BHY05]: one can obtain an expressive system by introducing existential polymorphism in the π-calculus. On channels, values are communicated along with types, and the types of values can depend on the types communicated simultaneously. Thus, an input prefix $a(\widetilde{x};\widetilde{T})$ waits both for values $\widetilde{x}$ and for types $\widetilde{T}$, the type of each x_i depending on the types of $\widetilde{T}$. As a consequence, expressive processes can be defined, for instance:

$$!a(x_1, x_2; T).\overline{x_1}\langle x_2\rangle \mid \overline{a}\langle b, 3; \mathtt{int}\rangle \mid \overline{a}\langle c, b; \sharp(\mathtt{int})\rangle.$$

Here the same server $!a(x_1, x_2; T).\overline{x_1}\langle x_2\rangle$ can be used by both requests, by giving type $\sharp(T)$ to x_1 and $\sharp(T)$ to x_2. The first request $\overline{a}\langle b, 3; \mathtt{int}\rangle$ is typechecked if we give type $\sharp(\mathtt{int})$ to b and type $\mathtt{int}$ to 3. The second request $\overline{a}\langle c, b; \sharp(\mathtt{int})\rangle$ is typechecked if we give type $\sharp(\sharp(\mathtt{int}))$ to x. We notice that even if x_1 and x_2 have different types, b can instantiate both of them (but only in two different communications). We believe that one can very easily adapt our weight-based results to handle polymorphism of types, as long as a level discipline is enforced. For instance, in the case of our example, one can instantiate the type variable T with any type, but the level given to x_1 shall remain strictly smaller than the level of a. By checking that such constraints are enforced, one get a system very close to the ones of Sections 3.1 and 3.2.1, but allowing the use of polymorphic servers. A more challenging task is to design type systems accommodating polymorphism of levels (or regions, in the case of the λ-calculus of Section 7.2). Indeed, constraints have to be added to the polymorphic types in order to make sure that instantiations of level variables do not lead to arising of loops.

Broadening the application range of the hybrid method Moreover, we believe that the method we use in Section 7.1 can be explored further. By applying the "pruning and simulation" proof technique to type systems, one should be able to prove the termination of several interesting systems, such as a π-calculus with an explicit primitive recursion. We believe that it is also possible to use this technique to prove soundness of type systems ensuring other properties than termination, for instance, we would be able to prove the soundness of the type systems for complexity we mentioned above using such a method.

Bibliography

[AB08] Lucia Acciai and Michele Boreale. Responsiveness in process calculi. *Theor. Comput. Sci.*, 409(1):59–93, 2008.

[AD07] Roberto M. Amadio and Frederique Dabrowski. Feasible reactivity in a synchronous pi-calculus. *CoRR*, abs/cs/0702069, 2007.

[AM02] Roberto M. Amadio and Charles Meyssonnier. On decidability of the control reachability problem in the asynchronous pi-calculus. *Nord. J. Comput.*, 9(1):70–101, 2002.

[Ama09] Roberto M. Amadio. On stratified regions. In Zhenjiang Hu, editor, *APLAS*, volume 5904 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2009.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[BC92] Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *STOC*, pages 283–293. ACM, 1992.

[BCP08] Christelle Braun, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Compositional methods for information-hiding. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 443–457. Springer, 2008.

[BGM10] Patrick Baillot, Marco Gaboardi, and Virgile Mogbil. A polytime functional language from light linear logic. In Andrew D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 104–124. Springer, 2010.

[BHY05] Martin Berger, Kohei Honda, and Nobuko Yoshida. Genericity and the pi-calculus. *Acta Inf.*, 42(2-3):83–141, 2005.

[BN88] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1988.

[Bou07] Gérard Boudol. Fair Cooperative Multithreading. In *Proc. of CONCUR*, volume 4703 of *LNCS*, pages 272–286. Springer, 2007.

[CC04] Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In René Jacquart, editor, *IFIP Congress Topical Sessions*, pages 359–366. Kluwer, 2004.

[CCMW01] Erik Christensen, Francesco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. available from `http://www.w3.org/TR/wsdl`, 2001.

[CL87] Ahlem Ben Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137 – 159, 1987.

[CPR07a] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving thread termination. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 320–330. ACM, 2007.

[CPR+07b] Byron Cook, Andreas Podelski, Andrey Rybalchenko, Josh Berdine, Alexey Gotsman, Peter O'Hearn, and Dino Distefano. The Terminator Project: proof tools for termination and liveness. `http://research.microsoft.com/terminator/`, 2007.

[CYH09] Marco Carbone, Nobuko Yoshida, and Kohei Honda. Asynchronous session types: Exceptions and multiparty interactions. In Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors, *SFM*, volume 5569 of *Lecture Notes in Computer Science*, pages 187–212. Springer, 2009.

[Dav01] René David. Normalization without reducibility. *Ann. Pure Appl. Logic*, 107(1-3):121–130, 2001.

[DCdY07] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In Gilles Barthe and Cédric Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2007.

[DHKS07] Romain Demangeon, Daniel Hirschkoff, Naoki Kobayashi, and Davide Sangiorgi. On the Complexity of Termination Inference for Processes. In *Proc. of TGC'07*, volume 4912 of *LNCS*, pages 140–155. Springer, 2007.

[DHS08] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Static and dynamic typing for the termination of mobile processes. In Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, and C.-H. Luke Ong, editors, *IFIP TCS*, volume 273 of *IFIP*, pages 413–427. Springer, 2008.

[DHS10a] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Termination in Higher-Order Concurrent Calculi. Special issue of J. of Log. and Algebr. Program. for 20th Nordic Workshop on Programming Theory, 2010.

[DHS10b] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Termination in impure concurrent languages. In Paul Gastin and François Laroussinie, editors, *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2010.

[DN07] René David and Karim Nour. An arithmetical proof of the strong normalization for the lambda-calculus with recursive equations on types. In *TLCA'07: Proceedings of the 8th international conference on Typed lambda calculi and applications*, pages 84–101, Berlin, Heidelberg, 2007. Springer-Verlag.

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

[GCPV09] Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 16–28. ACM, 2009.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, 1989.

[HGB04] Thomas Hildebrandt, Jens C. Godskesen, and Mikkel Bundgaard. Bisimulation Congruences for Homer — a Calculus of Higher Order Mobile Embedded Resources. Technical Report TR-2004-52, Univ. of Copenhagen, 2004.

[HPH+09] Daniel Hirschkoff, Aurélien Pardon, Tom Hirschowitz, Samuel Hym, and Damien Pous. Encapsulation and dynamic modularity in the pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 241:85–100, 2009.

[HVY00] Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In Gert Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer, 2000.

[HW06] Dieter Hofbauer and Johannes Waldmann. Termination of {*aa-> bc*, *bb -> ac*, *cc-> ab*}. *Inf. Process. Lett.*, 98(4):156–158, 2006.

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

[LM93] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. *Fundam. Inform.*, 19(1/2):167–184, 1993.

[LS10] Ugo Dal Lago and Davide Sangiorgi. Light logics and higher order processes. EXPRESS'10, 2010.

[LSV07] Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Observing branching structure through probabilistic contexts. *SIAM J. Comput.*, 37(4):977–1013, 2007.

[Mil89] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MSNR96] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1):53–58, 1996.

[Nes00] Uwe Nestmann. What is a "good" encoding of guarded choice? *Information and Computation*, 156(1-2):287–319, 2000.

[NPP09] Gethin Norman, Catuscia Palamidessi, David Parker, and Peng Wu 0002. Model checking probabilistic and stochastic extensions of the pi-calculus. *IEEE Trans. Software Eng.*, 35(2):209–223, 2009.

[Pri95] Corrado Priami. Stochastic pi-calculus. *Comput. J.*, 38(7):578–589, 1995.

[PS00] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *J. ACM*, 47(3):531–584, 2000.

[San92] Davide Sangiorgi. Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. *PhD Thesis, University of Edinburgh*, 1992.

[San99] Davide Sangiorgi. The name discipline of uniform receptiveness. *Theor. Comput. Sci.*, 221(1-2):457–493, 1999.

[San06] Davide Sangiorgi. Termination of processes. *Mathematical Structures in Computer Science*, 16(1):1–39, 2006.

[SS04] Alan Schmitt and Jean-Bernard Stefani. The kell calculus: A family of higher-order distributed process calculi. In Corrado Priami and Paola Quaglia, editors, *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*, pages 146–178. Springer, 2004.

[Sta79] Richard Statman. The typed lambda-calculus is not elementary recursive. *Theor. Comput. Sci.*, 9:73–81, 1979.

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

[Ter03] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[Tho96] B. Thomsen. Calculi for Higher Order Communication Systems. *PhD Thesis, University of London*, 1996.

[Tra10] P. Tranquilli. Translating types and effects with state monads and linear logic. submitted, 2010.

[Tur36] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of *2*, pages 230–265, 1936.

[Tur96] David N. Turner. *The polymorphic pi-calculus: Theory and Implementation*. PhD thesis, Department of Computer Science, University of Edinburgh, 1996.

[VH93] Vasco Thudichum Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic pi-calculus. In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 1993.

[VY07] Daniele Varacca and Nobuko Yoshida. Probabilistic pi-calculus and event structures. *Electr. Notes Theor. Comput. Sci.*, 190(3):147–166, 2007.

[YBH04] Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the pi -calculus. *Inf. Comput.*, 191(2):145–202, 2004.

[Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. In Vijay Chandru and V. Vinay, editors, *FSTTCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer, 1996.

Appendix A

Additional Proofs

In this appendix, we put some technical proofs we do not give in the main document because of their similarities with proofs we presented above.

Proof. *[Proof of Lemma 3.2.10 from Section 3.2.1]*

1. *We prove the first result by induction over the typing judgement:*
   - *Case* (**KNil**) *is trivial, as* $\mathbf{0}\{v/x\} = \mathbf{0}$.
   - *Case* (**KPar**) *is easily done by using the induction hypothesis twice, as* $(P_1 \mid P_2)\{v/x\} = (P_1\{v/x\} \mid P_2\{v/x\})$.
   - *Case* (**KRes**) *is easily done using the induction hypothesis.*
   - *Case* (**KIn**). *We have* $\Gamma \vdash^{\kappa,a} a(y).P_1 : N$. *We derive* $\Gamma \vdash^{\kappa,a} P_1 : N$, $\Gamma(a) = \sharp^k T$ *and* $\Gamma(y) = T$ *and use the induction hypothesis to get* $\Gamma \vdash^{\kappa,a} P_1\{v/x\} : N$. *As we stated in Section 2, we suppose that the* x *is free in* P *which implies that* $x \neq y$. *Two cases can occur:*
     - *Either* $a \neq x$ *and* $(a(y).P_1)\{v/x\} = a(y).(P_1\{v/x\})$, *we use* (**KIn**) *to conclude.*
     - *Or* $a = x$ *and* $(a(y).P_1)\{v/x\} = v(y).(P_1\{v/x\})$. *As* $\Gamma(v) = \Gamma(a) = \sharp^k T$, *we use* (**KIn**) *to conclude.*
   - *Case* (**Out**). *We have* $\Gamma \vdash^{\kappa,a} \overline{a}\langle w\rangle.P_1 : N$. *We derive* $\Gamma \vdash^{\kappa,a} P_1 : M$, $\Gamma(a) = \sharp^k T$, $\Gamma(w) = T$ *and* $N = \{k\} \uplus M$ *and use the induction hypothesis to get* $\Gamma \vdash^{\kappa,a} P_1\{v/x\} : M$. *Three cases can occur:*
     - *Either* $a \neq x$ *and* $w \neq x$ *and* $(\overline{a}\langle w\rangle.P_1)\{v/x\} = \overline{a}\langle\rangle.(P_1\{v/x\})$, *we use* (**KOut**) *to conclude.*
     - *Or* $a = x$ *and* $w \neq x$, *and* $(\overline{a}\langle w\rangle.P_1)\{v/x\} = \overline{v}\langle w\rangle.(P_1\{v/x\})$. *As* $\Gamma(v) = \Gamma(a) = \sharp^k T$, *we use* (**KOut**) *to conclude* $\Gamma \vdash^{\kappa,a} (\overline{v}\langle w\rangle.P_1)\{v/x\} : \{k\} \uplus M)$.
     - *Or* $a \neq x$ *and* $w = x$, *and* $(\overline{a}\langle w\rangle.P_1)\{v/x\} = \overline{a}\langle v\rangle.(P_1\{v/x\})$. *As* $\Gamma(v) = \Gamma(w) = T$, *we use* (**KOut**) *to conclude* $\Gamma \vdash^{\kappa,a} (\overline{a}\langle v\rangle.P_1)\{v/x\} : \{k\} \uplus M)$.
     - *Case* $a = w = x$ *cannot happen as the calculus is simply-typed and* x *is supposed free in* P.
   - *Case* (**KRep**). *We have* $\Gamma \vdash^{\kappa,a} !a_1(x_1)^{l_1}.\ldots.a_n(x_n)^{l_n}.P_1 : \emptyset$. *We derive* $\Gamma \vdash^{\kappa,a} P_1 : M$, *for all* i, $\Gamma(a_i) = \sharp^{k_i} T_i$, $\Gamma(x_i) = T_i$, *and* $M <_{\text{mul}} \{k_1, \ldots, k_n\}$. *Every* x_i *is different from* x *as* x *is free in* P. *We note* σ *the mapping of names into names which is the identity except on* x *which is mapped to* v. *Clearly, for all* i, $\Gamma(\sigma(a_i)) = \Gamma(a_i) = \sharp^{k_i} T_i$. *We use the induction hypothesis to get* $\Gamma \vdash^{\kappa,a} P_1\{v/x\} : M$ *and use rule* (**KRep**) *to conclude* $\Gamma \vdash^{\kappa,a} !\sigma(a_1)(x_i)^{l_1}.\ldots.\sigma(a_n)(x_n)^{l_n}.P_1 : \emptyset$.
2. *We have* $\Gamma \vdash^{\kappa,a} !a_1(x_1)^{\mathtt{ok}}.\ldots.a_{q-1}(x_{q-1})^{\mathtt{free}}.a_q(x_q)^{\mathtt{free}}.\ldots.a_n(x_n)^{\mathtt{free}}.P_1 : N$. *We derive* $\Gamma \vdash^{\kappa,a} P_1 : M$ *and for all* i $\Gamma(a_i) = \sharp^{k_i} T_i$ *and* $\Gamma(x_i) = T_i$, *and* $M <_{\text{mul}} \{k_1, \ldots, k_n\}$. *We use the previous result to derive* $\Gamma \vdash^{\kappa,a} P_1\{v/x\} : M$. *We conclude as in case* (**KRep**) *of result* 1, *only applying* σ *to* a_i *where* $i \geq q$.

□

Proof. *[Proof of Lemma 3.2.11 from Section 3.2.1]*
By structural induction over $\mathbf{E}$,

- *Case* $[\,]$. *Condition 1 holds trivially and condition 2 holds by setting* $N_{(0)} = N_0$.
- *Case* $(\boldsymbol{\nu} a)\ \mathbf{E}_2$. *Condition 1 holds with* $N' = N$ *and condition 2 holds by setting* $N_{(0)} = N_0$.
- *Case* $\mathbf{E} = \mathbf{E}_2 \mid P_1$. *We derive* $\Gamma \vdash^{\kappa,a} \mathbf{E}_2[P] : N_2$ *and* $\Gamma \vdash^{\kappa,a} P_1 : N_1$ *with* $N = N_2 \uplus N_1$. *The induction hypothesis gives* $\Gamma \vdash^{\kappa,a} P : N'$ *for some* N', *thus we get condition 1. The induction hypothesis also gives* $\Gamma \vdash^{\kappa,a} \mathbf{E}_2[P_0] : N_{(2)}$ *for some* $N_{(2)}$. *We set* $N_{(0)} = N_{(2)} \uplus N_1$ *and we get condition 2.*

□

Proof. *[Proof of Lemma 4.3.6 from Section 4.2]*
By structural induction on $\mathbf{E}$.

- *Case* $[\,]$. *Condition 1 holds trivially and condition 2 holds by setting* $n_{(0)} = n_0 \leq n' = n$.
- *Case* $(\boldsymbol{\nu} a)\ \mathbf{E}_2$. *Condition 1 holds with* $n' = n$ *and condition 2 holds by setting* $n_{(0)} = n_0 \leq n' = n$.
- *Case* $\mathbf{E} = \mathbf{E}_2 \mid P_1$. *We derive* $\Gamma \vdash_{PaPi} \mathbf{E}_2[P] : n_2$ *and* $\Gamma \vdash_{PaPi} P_1 : n_1$ *with* $n = \max(n_2, n_1)$. *The induction hypothesis gives* $\Gamma \vdash_{PaPi} P : n'$ *with* $n' \leq n_2$, *as* $n_2 \leq n$ *we get condition 1. The induction hypothesis also gives* $\Gamma \vdash_{PaPi} \mathbf{E}_2[P_0] : n_{(2)}$ *with* $n_{(2)} \leq n_2$. *We set* $n_{(0)} = \max(n_{(2)}, n_1)$ *and we get condition 2.*
- *Case* $\mathbf{E} = l(\!|\mathbf{E}_2|\!)$. *We derive* $\Gamma(p) = \mathbf{loc}^n$, $\Gamma \vdash_{PaPi} \mathbf{E}_2[P] : n_2$, $n > n_2$. *The induction hypothesis gives* $\Gamma \vdash_{PaPi} P : n'$ *with* $n' \leq n_2 < n$. *Thus condition 1 holds. The induction hypothesis also gives* $\Gamma \vdash_{PaPi} \mathbf{E}_2[P_0] : n_{(2)}$ *with* $n_{(2)} \leq n_2$. *We set* $n_{(0)} = n$ *and we get condition 2, as* $n > n_2 \geq n_{(2)}$.

□

Proof. *[Proof of Lemma 4.3.9 from Section 4.2]*
We reason by induction on the typing derivation:

- *Cases* (**PaNil**), (**PaVar**), (**PaPar**) *and* (**PaRes**) *are treated easily using the induction hypotheses (where relevant) as well as Definition 4.3.5.*
- *Case* (**PaLoc**). *Suppose* $P = x(\!|Q_1|\!)$ *(the case* $P = l(\!|Q_1|\!)$ *with* $l \neq x$ *can be deduced from the following). Then* $T_V = \mathbf{loc}^n$, *and using* (**PaLoc**), *we derive* $\Gamma \vdash_{PaPi} Q_1 : n_1$, *for some* n_1 *s.t.* $n > n_1$. *The induction hypothesis gives* $\Gamma \vdash_{PaPi} Q_1\{q/x\} : n_1'$ *for some* $n_1' \leq n_1$ *and* $\mathbf{M}^{PaPi}(Q_1\{q/x\}) = \mathbf{M}^{PaPi}(Q_1)$. *As* $\Gamma(q) = T_V = \mathbf{loc}^n$ *and* $n > n_1 \geq n_1'$, *we can derive* $\Gamma \vdash_{PaPi} q(\!|Q_1\{q/x\}|\!) : n$. *As* q *and* x *have the same type,* T_V, *we can use use Definition 4.3.5 to conclude.*
- *Case* (**PaPas**). *Suppose* $x(X) \triangleright P_1$ *(the case* $P = l(X) \triangleright P_1$ *and* $l \neq x$ *can easily be deduced from the following). There exists* k *s.t.* $T_V = \mathbf{loc}^k$ *and using rule* (**PaPas**), *we get* $\Gamma \vdash_{PaPi} P_1 : n$. *Induction gives* $\Gamma \vdash_{PaPi} P_1\{q/x\} : n'$, *for some* $n' \leq n$. *As* $\Gamma(q) = \mathbf{loc}^k$ *we can derive* $\Gamma \vdash_{PaPi} q(X) \triangleright P_1\{q/x\} : n'$. *As* q *and* x *have same type* T_V, *we conclude using Definition 4.3.5.*
- *Case* (**PaInP**) *is treated like case* (**PaPas**).

- *Case* (**PaOutP**). *Suppose* $P = \overline{x}\langle Q_2\rangle.P_1$ *(the case* $P = \overline{p}\langle Q\rangle.P_1$ *and* $p \neq x$ *can easily be deduced from the following). There exists* k *s.t.* $T = \sharp^k \diamond$ *and, using rule* (**PaOutP**), *we get* $\Gamma \vdash_{PaPi} P_1 : n_1$ *and* $\Gamma \vdash_{PaPi} Q_2 : n_2$, *with* $k > n_2$ *and* $n = \max(k, n_1)$ *for some* n_1, n_2. *The induction hypothesis gives* $\Gamma \vdash_{PaPi} P_1\{q/x\} : n_1'$, $\Gamma \vdash_{PaPi} Q_2\{q/x\} : n_2'$, $n_1' \leq n_1$, $n_2' \leq n_2$, $\mathbf{M}^{PaPi}(P_1) = \mathbf{M}^{PaPi}(P_1\{q/x\})$ *and* $\mathbf{M}^{PaPi}(Q_2) = \mathbf{M}^{PaPi}(Q_2\{q/x\})$. *As* $k > n_2 \geq n_2'$ *and* $\Gamma(q) = \sharp^k \diamond$, *we derive* $\Gamma \vdash_{PaPi} \overline{q}\langle Q_2\{q/x\}\rangle.(P_1\{q/x\}) : \max(k, n_1')$. *We conclude by stating that* $\max(k, n_1') \leq \max(k, n_1)$ *and relying on Definition 4.3.5 to obtain the result on the measure.*

- *Case* (**PaInN**). *Suppose* $P = x(y).P_1, \Gamma(y) = T_V'$ *(the case* $P = p(y).P_1$ *and* $p \neq x$ *can be deduced from the following). As our processes abide the Barendregt Convention,* $y \neq x$. *There exists* k *s.t.* $T_V = \sharp^k T_V'$ *and, using* (**PaInN**), *we derive* $\Gamma \vdash_{PaPi} P_1 : n$. *The induction hypothesis gives* $\Gamma \vdash_{PaPi} P_1\{q/x\} : n'$ *for some* $n' \leq n$. *As* $\Gamma(q) = \sharp^k T_V'$ *we can derive* $\Gamma \vdash_{PaPi} q(y).P_1\{q/x\} : n'$. *As* x *and* q *have the same type, we can conclude using Definition 4.3.5.*

- *Case* (**PaOutN**). *Suppose* $P = \overline{x}\langle q'\rangle.P_1$ *(the case* $P = \overline{p}\langle q'\rangle.P_1$ *and* $p \neq x$ *can easily be deduced from the following). There exists* k *s.t.* $T = \sharp^k T_V'$, $\Gamma(q') = T_V'$ *and, using rule* (**PaOutN**) *we get* $\Gamma \vdash_{PaPi} P_1 : n_1$ *and* $n = \max(k, n_1)$. *The induction hypothesis gives* $\Gamma \vdash_{PaPi} P_1\{q/x\} : n_1'$ *with* $n_1' \leq n_1$. *As,* $\Gamma(q) = \sharp^k T_V'$, *we derive* $\Gamma \vdash_{PaPi} \overline{q}\langle q'\rangle.P_1\{q/x\} : \max(k, n_1')$. *We conclude by stating that* $\max(k, n_1') \leq \max(k, n_1)$, *and using Definition 4.3.5.*

- *Case* (**PaRep**) *can be deduced from cases* (**PaInN**) *and* (**PaLoc**).

□

Proof. *[Proof of Lemma 4.3.10 from Section 4.2]*
By induction on the typing judgement:

- *Cases* (**PaVar**) *when* $P = Y \neq X$, (**PaNil**), (**PaRes**), (**PaPar**), (**PaPas**), (**PaInP**), (**PaOutN**) *and* (**PaInN**) *and are easily treated using the induction hypotheses (where relevant), as well as Definition 4.3.5.*

- *Case* (**PaRep**) *is treated using the induction hypothesis and Definition 4.3.5 as we impose the condition* $n' \leq n$ *in the statement of the lemma.*

- *Case* (**PaVar**). *Suppose* $P = X$. *We conclude using the hypothese* $\Gamma \vdash_{PaPi} Q : m'$ *with* $m' \leq m$ *and Definition 4.3.5, with* $c = 1$.

- *Case* (**PaLoc**). *Suppose* $P = l(\!|Q_1|\!)$. *We have* $\Gamma(l) = \mathbf{loc}^n$ *and, using rule* (**PaLoc**), *we derive* $\Gamma \vdash_{PaPi} Q_1 : n_1$, *for some* $n_1 < k$. *The induction hypothesis gives* c_1 *s.t.* $\Gamma \vdash_{PaPi} Q_1\{Q/X\} : n_1'$ *for some* $n_1' \leq n_1$ *and* $\mathbf{M}^{PaPi}(Q_1\{Q/X\}) = \mathbf{M}^{PaPi}(Q_1) \uplus c_1.\mathbf{M}^{PaPi}(Q)$. *As* $k > n_1 \geq n_1'$, *we can derive* $\Gamma \vdash_{PaPi} l(\!|Q_1\{Q/X\}|\!) : n$. *As* $\mathbf{M}^{PaPi}(X) = \emptyset$, *we conclude using Definition 4.3.5, with* $c = c_1$.

- *Case* (**PaOutP**). *Suppose* $P = \overline{p}\langle Q_2\rangle.P_1$. *There exists* k *s.t.* $\Gamma(p) = \sharp^k \diamond$ *and, using rule* (**PaOutP**), *we derive* $\Gamma \vdash_{PaPi} Q_2 : n_2$ *and* $\Gamma \vdash_{PaPi} P_1 : n_1$ *with* $n_2 < k$ *and* $n = \max(k, n_1)$ *for some* n_1, n_2. *By the induction hypothesis, we deduce the existence of* c_1, c_2 *s.t.* $\Gamma \vdash_{PaPi} Q_2\{Q/X\} : n_2'$ *and* $\Gamma \vdash_{PaPi} P_1\{Q/X\} : n_1'$ *for some* n_1', n_2' *s.t.* $n_2' \leq n_2$ *and* $n_1' \leq n_1$, $\mathbf{M}^{PaPi}(P_1\{Q/X\}) = \mathbf{M}^{PaPi}(P_1) \uplus c_1.\mathbf{M}^{PaPi}(Q)$ *and* $\mathbf{M}^{PaPi}(Q_2\{Q/X\}) = \mathbf{M}^{PaPi}(Q_2) \uplus c_2.\mathbf{M}^{PaPi}(Q)$. *As* $k > n_2 \geq n_2'$, *we can derive* $\Gamma \vdash_{PaPi} \overline{p}\langle Q_2\{Q/X\}\rangle.P_1\{Q/X\} : \max(k, n_1')$. *We conclude using Definition 4.3.5, with* $c = c_1$.

□

Proof. *[Proof of Lemma 7.1.2 from Section 7.1.2]*
By induction over the typing judgement:

- *Case* (**PFDef**). *We have* $\Gamma \vdash \texttt{def}\ f = (y).P_1\ \texttt{in}\ P_2 : l$ *from which we derive* $\Gamma \vdash P_1 : l_1$, $\Gamma(f) = \sharp^k_{\mathrm{F}}\, T'$ *and* $\Gamma \vdash P_2 : l$ *with* $k \geq l_1$. *As* $f, y \notin \mathrm{fn}(P)$, $f \neq x$ *and* $y \neq x$, *thus* $P\{v/x\} = (\texttt{def}\ f = (y).P_1\{v/x\}\ \texttt{in}\ P_2\{v/x\})$. *The induction hypothesis gives* $\Gamma \vdash P_1\{v/x\} : l_1$ *and* $\Gamma \vdash P_2\{v/x\} : l$. *We conclude using rule* (**PFDef**).

- *Case* (**PFOut**). *We have* $\Gamma \vdash \overline{v'}\langle w\rangle.P_1 : l$ *from which we derive* $\Gamma \vdash P_1 : l_1$, $\Gamma(v') = \sharp^k_{\bullet}\, T'$, $\Gamma(w) = T'$ *and* $l = \max(k, l_1)$. *Notice that* $v, w \in \mathrm{fn}(P)$. *Several cases can occur:*
  - $v' \neq x$ *and* $w \neq x$, *then* $P\{v/x\} = \overline{v}\langle w\rangle.P_1\{v/x\}$. *We use the induction hypothesis to conclude.*
  - $v' = x$. *Notice that simple typability ensures* $w \neq x$. *Then* $P\{v/x\} = \overline{v}\langle w\rangle.(P_1\{v/x\})$ *and* $T = \sharp^k_{\bullet}\, T'$. *We use the induction hypothesis and conclude using rule* (**PFOut**).
  - $w = x$. *Notice that simple typability ensures* $v' \neq x$. *Then* $P\{v/x\} = \overline{v'}\langle v\rangle.(P_1\{v/x\})$ *and* $T = T'$. *We use the induction hypothesis and conclude using rule* (**PFOut**).
  - *Notice that case* $v' = w = x$ *cannot happen, from simple typability.*

- *Case* (**PFRep**). *We have* $\Gamma \vdash !a(y).P_1 : l$ *from which we derive* $\Gamma \vdash P_1 : l_1$, $\Gamma(a) = \sharp^k_{\mathrm{I}}\, T'$, $\Gamma(y) = T'$, $k > l_1$ *and* $l = 0$. *Notice that* $a \in \mathrm{fn}(P)$ *and* $y \notin \mathrm{fn}(P)$. *Two cases can occur:*
  - $a \neq x$, *then* $P\{v/x\} = !a(y).(P_1\{v/x\})$. *We use the induction hypothesis to conclude.*
  - $a = x$, *then* $P\{v/x\} = !v(y).(P_1\{v/x\})$ *and* $T = \sharp^k_{\bullet}\, T'$. *We use the induction hypothesis and conclude using rule* (**PFRep**) *as the constraint* $k > l_1$ *still holds.*

- *Case* (**PFIn**) *is similar.*

- *Cases* (**PFNil**), (**PFRes**) *and* (**PFPar**) *are easy, using the induction hypothesis.*

□

Proof. *[Proof of Lemma 7.2.4 from Section 7.1.2]*
By structural induction on $\mathbf{E}$*:*

- *Case* $[]$. *Then* $m_1 = n$ *and* $T = T_1$. *We set* $n' = m_{(1)}$. *As* $m_{(1)} \leq m_1$ *we conclude.*

- *Case* $(\mathbf{E}_2\ M_2)$. *We derive* $\Gamma \vdash M_2 : (T_2, n^{(2)})$ *and* $\Gamma \vdash \mathbf{E}_2[M_1] : (T_2 \to^{n^2} T, n_2)$ *with* $n = \max(n_2, n^2, n^{(2)})$. *By the induction hypothesis we get* $\Gamma \vdash \mathbf{E}_2[M_{(1)}] : (T_2 \to^{n^2} T, n'_2)$ *with* $n'_2 \leq n_2$. *We derive* $\Gamma \vdash (\mathbf{E}_2[M_{(1)}]\ V_2) : (T, \max(n'_2, n^2, n^{(2)}))$ *using rule* (**App**). *As* $n'_2 \leq n_2$, *we conclude.*

- *Case* $\texttt{ref}_{n^2}\ \mathbf{E}_2$. *We derive* $\Gamma \vdash \mathbf{E}_2[M_1] : (T_2, n_2)$ *with* $n = \max(n_2, n^2)$, $T = T_2\ \texttt{ref}_{n^2}$ *and* T *is well-formed. By the induction hypothesis we get* $\Gamma \vdash \mathbf{E}_2[M_{(1)}] : (T_2, n'_2)$ *with* $n'_2 \leq n_2$. *We derive* $\Gamma \vdash \texttt{ref}_{n^2}\ \mathbf{E}_2[M_{(1)}] : (T, \max(n'_2, n^2))$ *using rule* (**App**). *As* $n'_2 \leq n_2$, *we conclude.*

- *The other cases are similar.*

□