Università degli Studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Master Thesis

Types for Seal Calculus

Candidate
Francesco Zappa Nardelli

Supervisor
Giuseppe Castagna

Supervisor
Giorgio Ghelli

Academic Year 1999-2000

Summary

In this Master Thesis we study how the notion of typing interacts with locations and mobility primitives in the framework of process calculi. We develop a notion of interface of an agent as a suitable abstraction of its interaction protocol. Each agent is characterized at the type level by its interface, and the type system assures that all interactions which can take place in a well-typed system respect the interfaces defined by the programmer. Reduction steps do not affect interfaces, which result an useful abstraction in order to program correct systems.

Seal Calculus We propose the Seal Calculus as a computational model of the global computer which the Internet may eventually become. Seal Calculus is a process calculus that lies in the crowded amount of languages spawned by π-calculus. As in the latter, interaction takes place over named channels. Moreover hierarchical locations are added to the syntax, and locations influence the possible interactions among processes. Communication is restrained to take place inside a location, or to spread over two locations that are in parent-child relationship. Locations are also the unit of movement, abstracting both the notions of *site* and *agent*: a location, together with its contents, can be sent over a channel, mimicking mobility of active computations.

Types for Seal Calculus We define a static type system for the calculus. The type system assures two properties. The first is that good values are transmitted over appropriate channels, that is, it types communication. The second is that the protocol to interact with a location is not invalidated by its displacement, that is, it types mobility. We achieve the latter result, far more interesting, by defining a suitable notion of *interface* of a location, which abstracts its interaction protocol. This constitutes the main novelty of this work. We obtain a standard subject reduction result.

We study the expressivity of the resulting calculus by encoding in it a distributed Toy language which summarizes the most common patterns of interaction currently found in Internet programming. We also extend the type system with the possibility of declaring a location as *mobile* or *immobile*, being this one of the simplest properties related to mobility one can imagine.

In more detail, the work is structured in five chapters and one appendix:

Chapter 1. We investigate why the traditional semantic models are not suited as basis for the investigation of the global computer, and why a shift to a *computation as*

interaction standpoint is required. We briefly survey the history of process calculi, starting from the early proposal for static and dynamic connectivity, analyzing the need of adding locations in the syntax, and the modern Babel Tower of combinators and rules.

Chapter 2. We review the most successful applications of a type system in the domain of process calculi, discussing how types interact with concurrency, and pointing out what may, what have been, and what may not be achieved imposing a static typing discipline.

Chapter 3. We describe the syntax and the semantics of the Untyped Seal Calculus. The calculus defined here is a simplification and a refinement of the quite untractable original proposal for Seal Calculus by Castagna and Vitek.

Chapter 4. We develop the type system. We give its formal description, we discuss the problems that arose defining it and we explain how they have been addressed. We derive from the type system a type-checking algorithm and we discuss the main results obtained.

Chapter 5. We analize the differences between the Seal Calculus used in this work and the original proposal for the calculus. The expressiveness of the calculus is tested against the encoding of the same distribuited Toy language which is used to test the expressivity of Typed Ambient Calculus. Then, we explore a simple extension of the type system, which keeps track of mobility attributes of locations.

Appendix A. In the appendix is reported the subject reduction proof. It is long, complex, and full of technical details. Unfortunately, it is also tedious.

This Master Thesis is addressed to a wide range of people, and we suggest a bunch of possible reading tracks .

If the reader is interested in an essential survey of process calculi concepts then he will find in Chapter 1 and 2 what he is looking for. On the other hand, the reader interested in the research core should read Chapter 3 up to 5. In the first two chapters the most intriguing ideas we need in the research section hiddenly lie: even if they are not required in order to fully understand the main developments, reading the whole work is warmly suggested.

A priori, we do not refuse other paths, even if we have minor concerns on them. For instance, some readers will skip the first five chapters, reading only Appendix A. Others, interested in the history of Seal Calculus will concentrate on Chapter 3 and Chapter 5. Lastly, the reader who considers Untyped Seal Calculus as major and definitive breakthrough in process calculi evolution will devote himself to Chapter 1, Chapter 5, and Chapter 3.

Acknowledgments

Special thanks to

> Giuliana Nardelli, Paolo Zappa, Giorgio Ghelli, Giuseppe Castagna, Veronique Benzaken, Giuseppe Longo, Guy Cousineau, Davide Sangiorgi, Juliusz Chroboczek, Annick Grandemange, Alessandro Urpi, Ernesto Mistretta, Davide Malacaria, Amy Clemitshaw, Matteo Coccia, Pio Nardiello, Arnaud Shadozeau, Miriam Menichelli, Frederick DeJaeger, Gabriele Santini, Katy Wozniak, Alessandro Casanovi, Fabio Gadducci, Nico and the *chez Paco* crew, Rolando Paolicchi, Margherita Peliti, Gianni Franceschini, and Stephanie Novak

for their support, help, and frienship. Everyone knows how much I owe him.

Contents

1

Computation as Interaction

> "I reject the idea that there can be a unique conceptual model, or one preferred formalism, for all aspects of something as large as concurrent computation, which is in a sense the *whole* of our subject—containing sequential computing as a well-behaved special area. [...] But there is a complementary *claim* to make, and is this: Computer scientists, as all scientists, seek a common framework in which to link and to organize many levels of explanation; moreover, this common framework must be semantic, since our explanations (including programs) are typically in formal language."
>
> *- Robin Milner, overview of his Turing Award Lecture [26]*

Denotational Semantics The most established paradigm for the semantics of computation is Denotational Semantics [33]. If we shed a beam of abstracting light on this paradigm, we can consider it as translations from a formal system to another: traditionally, between the syntax of a programming language and some mathematical model, hopefully well-understood. The key point is the *compositionality* of the translation: the meaning of a term depends only on the meaning of its subterms.

This paradigm is the best approximation of the ideal of a mathematical theory of computation in the sense of McCarthy [21] or Scott [30]. It is focused on the notion of *mathematical function* and directly models purely functional languages: by defining appropriate translations and embeddings, its influence can be elegantly extended to include the whole sphere of computation in which the behavior of the program is adequately abstracted as the computation of a function [27].

Within the sphere of functional computation, Denotational Semantics has worked extremely well, both in describing and in indicating new directions in the development of programming languages: simply think of types and type-checking, higher order functions, polymorphism, continuations and monads.

But outside the sphere of functional computation, Denotational Semantics successes have been much more limited: reactive systems, distributed systems, real-time systems and mobile agents systems do not really fit in the paradigm. Refinements of what is natural for sequential computing just appear not to work.

Concurrency Every proposal to model the ideal of Global Computation must comprehend *concurrency*, and concurrency is enough to require a shift of the semantic model. Compositionality *is lost* if we admit that programs can be arbitrarily combined. A simple example will make the point clear. Consider the three following programs:

```
P1 :: x = 1 ; x = x + 1
P2 :: x = 2

Q :: x = 3
```

In absence of interference, `P1` and `P2` are obviously equivalent, but imagine to put them with `Q` running at the same time on the same machine (briefly, in parallel with `Q`)

```
R1 :: P1 par Q
R2 :: P2 par Q
```

and you obtain two programs `R1` and `R2` that are no more equivalent. A compositional semantics must be more refined; it has to take into account of the way the programs *interact* with the memory. Refinements of the memory model have been deeply investigated, but they grow rapidly towards a complexity which makes them an untractable source of errors.

In presence of concurrence or interference, memory is no longer under the control of a single program: it interacts with programs instead. Once memory is no longer at a single master's behest, why not trying to abstract from it, placing it at the same level as the other entities of the overall design?

Interaction It is not a coincidence if we have been so far using the term *interaction* and describing programs as *interacting*. In the late of 70's, Tony Hoare and Robin Milner focused their theories on the central idea of interaction, or more specifically communication. Their approaches were different, Milner was looking for a semantic basis to concurrent computation, while Hoare was designing the core of a programming language. The convergence of their efforts can be considered a clear evidence that the notion is a pivotal one.

In 1978, Hoare's *Communicating Sequential Processes* was published in *Communications of the ACM* [17]: it describes a programming language beautifully designed around the primitive of indivisible event. On the other hand, in 1980, Milner recorded his efforts in *A Calculus of Communicating Systems* [23].

The greatest achievement of the new semantic paradigm, named *Process Calculi*, was to develop an algebraic theory of concurrency, as a basis for structural methods of description of concurrent systems.

Processes Process calculi is not the only semantic paradigm proposed to describe reactive systems: at the root of the concurrency theory a sure place is granted to Carl Adam Petri, with his theory of *nets*. For him already, the same kind of object —namely a net— was used to model both programs and memory. What was missing in his theory is the familiar synthetic or compositional view of systems we have taken from programming and from real world. A system has a diversity, being composed of several parts each acting concurrently with, and independently of, other parts; on the other hand a system has a unity, achieved through interactions among its parts. The need for a uniform, abstracted, view of a system become clearly evident if we want to be able to formalize a system independently of the abstraction level.

The basic entity we will focus on is called *process* and the magic glue to compose processes in order to obtain more complex systems (and so again processes) will be found in the *parallel* operator. Two processes, P and Q, can freely be placed in parallel obtaining $P \mid Q$[1]: this only means that the two before independent processes P and Q are now acting side by side, free to interact in whatever way we have designed them to interact.

This is the common framework of the two most influential proposals at the very beginning of process calculi development: CSP and CCS.

1.1 Static connectivity

1.1.1 CSP

Mother nature is rich of suggestions, and a good observer should be able to extract his scientific work from the world around him, adding just a bit of notational convenience (by the way, extracting may be really hard).

Each object has been designed with a set of events or actions which will be of interest. In the case of a simple vending machine (thanks to Tony Hoary for his immortal example, [18]), there are two kinds of relevant events

coin	the insertion of a coin in the slot of the vending machine.
choc	the extraction of a chocolate from the dispenser of the machine.

Of no interest at all for a vending machine are, for example

tick	the noise of the clock in the hall.
money	changing money to buy coins.

The behaviour of the machine, as observed from somebody sitting in the hall, can be described using only the relevant two events, and will appear not much different from

$$coin \to choc \to coin \to choc \to coin \to choc \to coin \to \dots$$

Now the observer wides its horizon, and he starts looking not only to the machine, but to the ensemble machine-customer. The actions of interest to the thirsty customer are

coin	the insertion of a coin into the slot of the vending machine.
choc	the extraction of a chocolate from the dispenser of the machine.
enjoy	the ingestion of the chocolate, with plenty of personal satisfaction.

Some actions like *coin* and *choc* are shared between the machine and the customer: the machine cannot accept a coin if the customer does not insert it, and the customer cannot

[1] $P \mid Q$ is read P *parallel* Q; the original name for the $\mid$ operator was the more intriguing *composed*.

insert the coin if the machine is not ready to accept it (a rare event). Briefly, they must interact to perform the shared actions. On the other hand, if the customer has succesfully interacted with the machine and he has his hot chocolate in one hand, he must not ask permission to the machine in order to enjoy it. Resuming, two entities can be composed: they have to interact in order to perform the common events, while they are able to perform independently independent actions.

Defining suitable classes of events, all standard forms of interaction can be studied in this apparently simple framework. Hoare designed CSP as a core concurrent language (which later evolved in the Occam language), integrating his brilliant analysis with the actual programming techniques of late 70.

1.1.2 CCS

The objective of Milner's investigation was to provide a framework for constructing and comparing different models, at different levels of abstraction. We will describe it in the light shed by Hoare's analysis.

Each event in CCS has two forms, either simple (a) or overbarred ($\overline{a}$). When two processes are put together to run concurrently, synchronization occurs only when one process engages in a barred event and the other process engages in the corresponding simple event. Their joint participation in such an event is hidden by immediate conversion to τ, a distinguished, invisible event. However, synchronization is not compulsory; each of the two events can also occur visibly and indipendently as an interaction with the outer environment. If more than two processes can engage in an interaction, the choice of which pair succeeds is non-deterministic.

It was an objective of CCS to achieve the maximum expressive power with as few distinct primitive operators as possible. This is the source of the elegance and power of CCS, and greatly simplifies the investigation of families of models defined by different equivalence relations. The discovery of a full set of laws of equivalence is a significant mathematical achievement.

The basic concurrent combinator of CCS, denoted by a single bar $|$, includes aspects of hiding, nondeterminism and interleaving as well as synchronization. At the contrary, CSP introduces as many combinators as needed, separately defining composition $\|$, nondeterministic choice $\sqcap$, guarded choice $\Box$, and concealment $\backslash$, each one in its purest form. The fact that the corresponding concepts are distinct is perhaps indicated by the simplicity of the algebraic laws of CSP. A reasonably wide range (and a finer control) of operators seems to be needed in practical application of useful mathematical theories. On the other hand, minimization of operator set is also useful, especially in theoretical investigations.

Unfortunately, CCS notation is suggestive: events can be interpreted as requests for synchronization on a named channel, and interaction as synchronization to occur. The common but a bit too abstract notion of event has been transformed into something which everyone attends to see in a concurrent and distributed calculus. This led to many extensions in which the notion of event has been sublimated in favor of the more common notion of *channel* and *communication over channels*. There is nothing bad in this, if we are still able to see and to reconstruct the layering of the ideas: channels and communication over channels are just a particular, maybe a well-suited one, kind of event.

1.2 Dynamic connectivity: π-calculus

The quest for a formalism of processes without the restriction of initial fixed connectivity was still open.

Some early paradigms for concurrency, such as Actors [16], allowed highly dynamic systems. In the CCS framework, the algebraic handling of full mobility among processes results in a failure: this is the essence of a discussion between Robin Milner and Mogen Nielsen at Aarhus in 1979. The wish to achieve an algebraic specification was partially motivated by the cited Hewitt's Actors system which he introduced much earlier; only several years later the discussion Engberg and Nielsen [14] succeeded in the effort. In 1989 π-calculus appears on the scene [22] as a simplification and strengthening of their work.

The key and pervasive notion is *naming*. The act of having a name presupposes being an autonomous and independent entity; at the same time communication is inextricably confused with using a name, an address. The choice of Milner may appear somewhat subjective and questionable, but every model of the reality has to derive some ontological commitment. Indeed, focusing on the notion of naming seems to bring into focus many aspects of computing: in particular, a name can be viewed as a reference to access data, and this unifies the treatment of communication and data-access. Briefly, naming is attractive as far as it is distilled directly from computing practice.

In the basic version of π-calculus there are only two entities, names and processes. Names are an infinite collection of objects that have no structure: a name refers to a link (also called a *channel*) on which processes can interact. The actions a process can sequentially fire are:

- input some name — call it y — along the link named x,
- output the name y along the link named x.

Every action is atomic and an interaction corresponds to the synchronous communication of a name along a named channel among two processes running in parallel. Communicating a name to a process allows it to dynamically acquire the possibility to use it: the initial connectivity of the system is changed.

The main novelty is the treatment of restricted names: a name can be declared private (or secret, to use a more fashionable term) to a set of processes. Imagine a group of people who share a secret: somebody not included in this elite can not know (and use) the secret. Meanwhile, a member of the elite can communicate the secret to somebody previously ignoring it: the new initiated person is now part of their group, free to dispose of the secret as he likes. Coming back from the real world, the restricted name would correspond to the secret, the elite would be the set of processes in the scope of the restricted name and the non-elected person would represent a process outside the scope of the restricted name. The act of communicating the restricted name to somebody outside the initial scope is known as *scope extrusion*.

The list of successes of this calculus indicates well enough how much the scientific community paid attention to it. In particular the λ-calculus has been successfully encoded in π-calculus [24]: the sequential computation appears as a particular case of concurrent computation, as expected.

Towards mobility In the previous section, we did not hesitate to use the word *mobility*, without defining it.

A π-calculus channel can be transmitted over another channel, and that is why we used the term mobility: to denote the possibility of changing dynamically the connectivity of a system.

This appears rather far from the meaning we expect today: the word "mobility" evokes in our minds ideas related to the downloading of code from remote locations or somewhat science-fictional agents able to find the less expensive flight from Pisa to Paris for us. The values of the base calculus are just names: adding whole processes to values should give the possibility of sending processes over channels, abstracting the idea of code mobility.

In the extended calculus, commonly called High-Order π-calculus, also inactive processes can be sent over channels: when they are received they are placed in parallel with the receiving process, and then activated. This seems quite good, and extremely attractive.

An important result by Davide Sangiorgi [29] shows that this extension does not add expressivity to the calculus: Higher-Order processes can be faithfully compiled down to first-order, respecting an appropriate notion of equivalence. The brilliant result is naturally hidden in the nature of the notion of naming and should not be surprising. Suppose there is a server process which can elaborate some data. Either the server can be sent to a remote location to operate on local resources, or just a reference to the server can be sent remotely. In the latter case, the reference will be successfully used to interact with the server, to send the data to be elaborated, to grant him access to local resources, and to get back the result of the computation.

The two scenarios are both actually used in real-world Internet applications, and correspond the first to some forms of code mobility, the second to (actually the more common) client-server interaction. At the level of detail we are observing, in the absence of considerations on efficiency and forgetting the overhead of local-global communication, the two scenarios must be equivalent.

This is good, but unfortunately in Internet all the hidden details we have been passing over appear as a distinctive part of the world. What we may need is a model which permits to keep them into account, without introducing too many unnecessary complications.

1.3 Locations and syntax

Everyday, everyone has to deal consciously or unconsciously with the consequences of the structure of Internet, the wide area networks which came along radically changing all the fundamental assumptions of distributed computing. Everyday, everyone has to deal with the unpredictable bandwidth fluctuations which can bring even the most stable ones on nerves.

But bandwidth fluctuations are only an observable effect of something more pervasive: the idea of *locations*. A world wide web, a planet-size structure, has to deal with the speed of light as an absolute lower bound to latency of communications: the existence of such delays implies the notion of physical distance between locations, which in turn implies the existence of separate locations.

Physical locations are a consequence of distance, but distance is not the only origin of locations. The presence of attackers causes the erection of virtual barriers between mutually distrustful administrative domains: communication now depends on the right access the specific user can have with respect to the specific administrative domain.

Whereas these new observable phenomena appear on the Web, failures become *un*-observable: *a priori* there is no upper bound on communication delays and failures become indistinguishable from long delays.

In π-calculus there is no notion of location, and no localization of processes and of resources; all processes act on the same flat structure. Concurrency already is too extended a phenomenon to be tractable in a single model: hoping that simply adding some concepts of locations to a previous successful but incomplete calculus will reveal in a comprehensive understanding of the reality is amusingly naïve. But like naming, location is a concept that can be successfully studied to shed a beam of light on the domain.

Towards mobility, revised Once we admit the existence of separate locations, delimited by physical or virtual barriers, the concept of mobility (and the whole concept of distribution) acquires many new shades of meaning. Now we can imagine that every location provides some services we may want to use, protected by some policies of access. To be able to do this, we must have the possibility to communicate through the barriers that delimit the location, using remotely the resource we are interested in, or to move ourselves inside the location, in order to use the resource as local. Our notion of process mobility is tightly linked to the idea of crossing domain barriers.

1.3.1 A calculus of locations: Ambient Calculus

After an extremely sharp analysis of the new observables arising on the Web [7], Luca Cardelli and Andrew Gordon concentrated their research to capture in an abstract way the notions of locality, mobility, and ability to cross barriers. The deriving calculus, named *Ambient Calculus* [9], is focused on *mobile computational ambients*; that is, places where computation happens and that are themselves mobile.

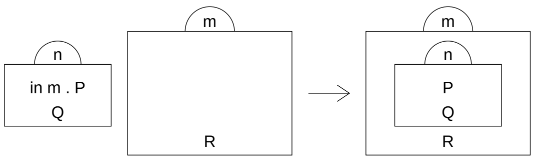
Ambients A place is naturally bounded, and a boundary determines what is inside and what is outside a place. For instance, an ambient can be viewed as the abstraction of a web page (bounded by a file), a virtual address space (bounded by an addressing range), a Unix file system (bounded within a physical volume), a single data object (bounded by "self") and a laptop (bounded by its case and its port). A Unix file system is contained in a laptop, and a Unix file system contains many single data objects. This suggests that ambients can be arbitrarily nested, forming a tree structure: administrative domains are often organized hierarchically and mobility can be represented as navigation across a hierarchy of ambients. For example, if we want to move a running application from work to home, the application must be removed from its initial enclosing ambient (work) and inserted in a different enclosing ambient (home).

Inside each ambient there is a collection of *local* running processes: local means that the process is contained inside the ambient, but in none of its subambients.

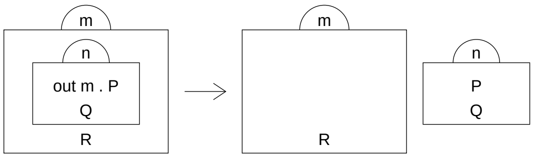
Interaction as Restructuring In Ambient Calculus the notion of interaction among processes changes radically. The set of local processes has direct control on the movement of the enclosing ambient: they can force it to enter in a "near-by" ambient, or they can force it to exit from its enclosing ambient. The third primitive permits a process to open a "near-by" ambient, discarding its boundaries and putting his content at the current level. If you have understood the philosophy of the calculus, you would have correctly realized that "near-by" means "within the same surrounding folder".

Considering that a picture is more valuable than a thousand of words, we report a graphic representation (originally due to Cardelli) of the reduction rules. An ambient is depicted as a folder: the ambient name is on the tag of the folder and processes are represented as gremlins active inside the folder. Reductions are depicted by means of arrows.

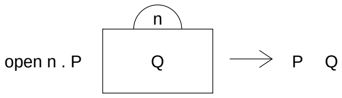
The `in` action can cause an ambient to enter another ambient: in the example the ambient n will enter the ambient m. If there was not an ambient named m nearby, the action would block until such an ambient could be found.



The `out` action causes an ambient to exit its enclosing ambient. The name of the parent ambient, although unique, must be explicitly specified. This might seem redundant, but can be easily exploited as a minimal security property: an ambient cannot escape from its enclosing ambient if it does not know its name. For example, this may be used to encode protocols resembling Java sand-boxes policies. As before, `out m` blocks if the parent ambient is not labeled m.



The `open` action is used to discard an ambient while keeping its content. As before, `open m` blocks if there is not an ambient named m nearby.



Interaction becomes rearrangement of the structure of nested ambients and processes are the authors of the changes.

The reduction rules are nondeterministic: when many possible reductions can take place at the same time, only one of them takes place. For instance, we may have two ambients with the same name n and one `open n` operation will select nondeterministically only one of them to be opened.

It is quite interesting to notice that these simple operations are computationally complete: Turing machines can be coded in a fairly direct way, so as arithmetic, and data structures like records and trees.

Communication Also communication can be coded, but the encoding is unbearably clumsy and we want to talk about communication in a direct way. So communication primitives compatible with the Ambient philosophy are added to the calculus: a new entity which can sit inside an ambient, the message, is introduced. A process can now also do either *output* operations, that is write a value on a message and attach it to the inside of the ambient, or an input operation, that is grab a message attached to the inside of the current ambient, and proceed with the knowledge of the current message. Communication may be viewed as the placement, and the subsequent removal, of a Post-It note on a message board. Each student (process) is free of leaving and removing messages, without having the possibility of making assumptions on the ordering of sent messages.

Among the messages, forgetting base values (which are not really interesting) for a while, we consider ambient names and actions. This may sound strange, but it is not. Messages can be viewed as capabilities: they allow the reader to perform some kind of privileged actions. Names are a kind of strong capabilities, since knowing the name of an ambient gives a total control on it, including the possibility to destroy it (with the open operation). More restrictive capabilities are found in the *navigation* capabilities: what is sent is only an action containing a name. For example, the capability `in m` will allow the receiving process to enter one of the ambients named `m`, but the receiving process will not allowed to escape or, even less welcomed, open it.

Ambients and Pigs

Ambients are like pigs, you never throw any part of them away.[2]

Forgetting these considerations for a moment, Ambient Calculus brings to the attention of the community the notion of location in a way somewhat extreme: this is surely a good way to try to understand what the notion of location implies and can imply. We will meet Ambients along this work many times: our calculus, the Seal Calculus, is an Ambient Calculus's nephew and we will test the expressiveness of it in Chapter 5 by comparing it with its uncle.

On the other hand, the study of simple calculi must be interleaved with experimentation with actual programming languages: at the moment the only practical language based on Ambient Calculus is TQL, a language for the query on semistructured database [8]. Although extremely innovative and surprisingly efficient, TQL is definitely not a language for programming Web applications.

1.3.2 The many flavors of Distributed π-calculus

In Ambient Calculus, ambients can be arbitrarily nested and interaction is essentially restructuring, with a limited form of communication. It derives directly from the ideal of *global computation* which someday in the future might be appealing to exploit. Our technology and our science of programming is far from being able to do that, at least on a reliable way; on the other hand distributed computing is a reality.

Many proposed calculi limit themselves in extending π-calculus with some kind of notion of locations, keeping the notion of interaction as communication. The design space is indeed large; we shall try to give an idea of the possible parameters that interact in designing a calculus.

The new ingredient, the locations, comes essentially in two flavors: they can form a flat structure or can be nested. Besides, a name can refer to a location only, as in Distributed-Join Calculus [15], or many locations with the same name can coexists.

Now, the existence of different locations can influence the communication pattern, or not. In the second case the calculus abstracts problems like routing of messages and the major cost (at least in terms of latency) of delivering remote packets. An example of the first class can be Seal Calculus, a calculus which admits locations arbitrarily nested, and in which communication is restricted to threads inside the same location, or that belong to two locations in parent-child relationship.

Finally, calculus can admit mobility of active agents, threads or whole locations. Movement capabilities can be *subjective* or *objective*: in the first case it is the agent who decides when and where migrate, in the latter it is the surrounding environment. Seal Calculus includes primitives to cause a location to move under the control of the environment, while boxed-π is a variant of Seal which does not admits mobility.

We must notice that theory should not be disjointed from practice. Many theoretical

[2] Fabio Gadducci, personal communication.

calculi forget implementations issues. For example, in presence of separate physical sites, not locating channels raises a global consensus problem for nearly every communication.

The Babel tower

Samson Abramsky, introducing his Interaction Categories [2], points out:

> " ... [in Process Calculi] there is veritable Babel of formalisms, combinators and equivalences."

Undoubtedly he is right. The absence of a canonical model might suggests that we are missing something fundamental, as it might not. And the best way to end the chapter is with the words of Creon, in Sophocle's Oedipus Tirannux:

> **Creon:** I do not know: where I lack insight it is my custom to be silent.[3]

[3] Sophocle, Oedipus Tirannux, 569, ed. Sir Richard Jebb.

2

Type Systems for Process Calculi

As soon as we start working on any domain, we begin to organize it in different ways: types arise naturally to categorize objects according to their usage and behavior. In set theory some sets are chosen to denote ordered pairs, and some sets of ordered pairs are then called functions. In λ-calculus, some functions are chosen to represent boolean values, others to represent integer.

At the same time, we are free to violate our organization: an unintended programming error can cause the application of the λ-encoding of the `or` function to a λ-term chosen to encode an integer, thus giving an ununderstandable term. A malicious violation could try to use an arbitrary integer as a pointer to reference a cell of memory.

A *type system* is the formalization of our intuitive organization: typed versions of set theory, just like typed programming languages, impose constraints on object interaction which prevent inconsistent interaction with other objects.

Cardelli and Wegner in their seminal paper "On Understanding Types, Data Abstraction, and Polymorphism" [13] use a metaphor to describe what a type is:

> "A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or uninteded use."

A type is the abstraction of a set of values that have the same behavior: once the behavior has been abstracted, the correctness of all the interactions taking place during all the executions of a program can be statically checked. This guarantees that certain classes of errors will not arise during execution, helping the programmer in the painful art of writing correct programs.

Types and Concurrency

The question of how types interact with concurrency is an interesting one: concurrency deals with flow of control, while type systems normally avoid flow of control questions. Like in the best known sequential computation, types are used to prevent inconsistent interactions.

But defining what constitutes an inconsistent interaction is not as evident as in sequential computation. An accurate choice permits to prove non trivial properties of the typed calculus in apparently unrelated domains, such as correctness of protocols or security. In general, type systems are strongly influenced only by the pattern of interaction among processes: many details of the calculus are abstracted, and it should not be surprising that many of the ideas originally proposed for a specific calculus can be adapted with no particular effort to other calculi.

We are now going to give a review of historical and current definitions of inconsistent interaction, pointing out the numerous properties that can be proved from the resulting type system.

2.1 Milner's Sorts

In the base π-calculus, only one name can be sent during a communication: a simple and useful extension consists in allowing a tuple of names to be sent and received in the same atomic communication. The new calculus, namely Polyadic π-calculus [25], presents a notion of run-time error. Thus, in an optic of *types to avoid run-time errors*, it is the minimal calculus that can support a notion of type system. We will make this point clear briefly.

In π-calculus, interaction occurs every time there is a process ready to output and another ready to input, both on the same channel. If we consider the polyadic extension we can have the following situation: a process is ready to output n names on channel x while another process is ready to input m names on the same channel x. Interaction can not take place, because the different arities prevent it: briefly, a run-time error occurs.

A brilliant observation of Milner is that all the case studies obey some discipline in the use of names: the length and the nature of the vector of names which a given name may carry in a communication is constant. This suggests a notion of name-use discipline, something like the *arities* of Martin-Löf, that could prevent deadlock to occur.

Milner introduced the notion of *sort*. Names are organized in sets named sorts, or equivalently sorts form a partition of the name space. Then, a *sorting* is just an application from sorts to vector of sorts. A process is said to respect a sorting if it use the names consistently with the sorting.

Suppose x is a name occurring in the process P, and *ob* is a sorting for the names in P. Suppose also that the name x belongs to the sort $\mathcal{S}$ and that the sorting for $\mathcal{S}$ is the vector $(\mathcal{S}_1, \cdots, \mathcal{S}_n)$. If P respects the sorting *ob* on the channel x are sent only vectors of names $(x_1, \cdots, x_n)$ where $x_1 \in \mathcal{S}_1 \wedge \cdots \wedge x_n \in \mathcal{S}_n$.

Formation rules for well-sorted processes are then given, and these can be statically checked. Two points must be remarked:

1. it is possible to send a name on itself,

2. the type of a well-sorted process is uninteresting.

Polyadic communication can be encoded in the base calculus: for instance, the communication of a tuple of names can be mimicked by the sequential communication of each of the names separately on a newly allocated name to prevent interference. Imagine now the encoding of the previous system: we have the sequential output of n names, in parallel with the sequential input of m names. If n is less than m (the other case is symmetrical), the output process will send all the packets, and after it will continue, while the input process will be blocked waiting for the $m - n$ missing messages. What was an error of protocol going to block the system (as suitable) encodes in a bizarre system in which a process remains blocked, while the other can continue uncontrolled. Other encoding are possible, but, according to the nature of the base calculus, they will no more present a notion of run-time (communication) error: in this sense Polyadic π-calculus is the minimal calculus which supports a type system.

2.2 Sorts and Types

Milner's sort discipline uses a by-name matching of sorts. Replacing it with the more straightforward structural matching (and the term *sort* with *type*) is a purely technical modification that will allow us to contrast typing for π-calculi with the more standard typing for λ-calculi.

A quite standard type system for λ-calculus might have the following syntax of types

$$T \;::=\; \mathrm{Int} \quad \mid \quad T \times T \quad \mid \; T \to T$$

where $T \to T'$ represents the type of functions taking arguments T and returning results of type T'. Given assumptions on the types of free variables, the type system will define the type of each λ-term, if any.

By contrast, for π-calculi the fundamental type constructor is not that of functions but of channels carrying values of given types. In a simple type system, one might have types

$$T \;::=\; \mathrm{Int} \quad \mid \quad T \times T \quad \mid \; \mathtt{Ch}\, T$$

where $\mathtt{Ch}\ T$ is the type of channels carrying values of type T. A type system will, under assumption on the types of free names, define whether a π-term is well-formed, or equivalently whether every action involves channels of the correct type only. For example, during reductions over a name of type $\mathtt{Ch}$ Int only integer value are sent, and over a name of type $\mathtt{Ch}\ \mathtt{Ch}$ Int only names of channel are sent.

We remark that in order to achieve the full expressivity of Milner's original sort system, we have to consider recursive types in the structural presentation of the type system. If not, we are not able to type configurations like a channel that is sent on itself.

The simple type system for π-calculus has been extended in many directions. To cite some of the most relevant proposals, David N. Turner in his PhD. Thesis [34] studies how to add polymorphism to the simple type system, much in the same way polymorphism was added to λ-calculus, while Benjamin Pierce and Davide Sangiorgi, in their excellent "Typing and Subtyping for Mobile Processes" [28], study how to integrate a notion of subtyping in the π-calculus framework. Besides, the type assigned to processes is uninteresting: other research directions include giving more informative types to processes.

Subtyping

A *subtyping relation* is a partial order relation on types, subjected to an exquisitely dynamic restriction: it must be possible to use a value of type T in all the contexts in which a value of a "super-type" of T is expected. The terms "super-type" and "sub-type" are commonly used to denote a type which is respectively greater than or less than another in the ordering relation. For example, the `natural` type can be viewed as a sub-type of the `integer` type since a natural number can be "safely" used every time an integer number is expected.

Subtyping captures the intuitive notion of inclusion between types, where types are seen as collections of values.

Pierce and Sangiorgi refine channel types so as to treat separately the capabilities for reading and writing on channels. Specifically, they annotate channel types with tags t

$$T \ ::= \ \ldots \quad | \quad {}^{t}\mathtt{Ch}\ T$$

which are taken from

$$\begin{tikzcd} + & & - \\ & \pm \arrow[ul] \arrow[ur] & \end{tikzcd}$$

Channel names of type ${}^{+}\mathtt{Ch}\ T$ (resp. ${}^{-}\mathtt{Ch}\ T$) can be used only for output (resp. input); names of type ${}^{\pm}\mathtt{Ch}\ T$ can be used for both input and output. The arrows in the diagram represent the subtyping relation: a channel able to support input and output operations can be safely used to perform only input or only output operations.

The greater precision of the proposed type discipline yields stronger versions of some standard theorems about the π-calculus: in particular it is used to prove the validity of the β-reduction for the most efficient of Milner's encodings of λ-calculus. The same result does not hold if we consider the untyped or simply-typed π-calculus. The type system is helpful also in preventing a class of programming errors. For example a library channel *printInt*, used for printing integers on some device, can be typed as an only output channel ${}^{+}\mathtt{Ch}$ Int. Type checking will detect erroneous attempts to read from this channel.

2.3 Type systems for Ambient Calculus

An ambient is a place where other ambients can come in and out, and where processes can exchange messages. Both aspects of interaction can be controlled by defining what an inconsistent interaction is and accordingly giving a suitable type systems.

2.3.1 Types to avoid deadlock: topics of conversation

Within an ambient, multiple processes can freely execute input and output operations, in the style of a message board. Since messages are undirected, there is no possibility to control that a process receives only messages it is ready to deal with. As in the previously described type systems, an inconsistent interaction is a communication of unexpected values. In practice, different folders will be dedicated to different kinds of conversation —in the methaphor, different message boards will be dedicated to different kinds of service—, in order to make assumptions about what should be written and what can be read in a given folder.

Typing messages is not a novelty: the innovative idea is to restrain the type of the messages which can be exchanged inside a particular ambient. In Ambient Calculus messages can be ambient names or capabilities. The type of an ambient name describes it as an ambient where messages of some type M can be exchanged. The type of a capability describes it as a capability that, when exercised, can unleash the exchange of messsages of some type M. The output of a name will be typed as a capability that may unleash messages of the type of the name, and will be allowed only inside an ambient that authorizes this kind of messages.

The resulting type system is interesting because, for the first time, the definition of a type system has to deal explicitely with the notion of location. In particular, the `open` operation can unleash, at the current level, pending messages that belonged to a subambient; the definition of a sensible type system to keep tracks of the topics of conversation must be carefully built around this tricky operation.

2.3.2 Types to reason about processes

Adding locations to the syntax and new primitives to deal with them has as major drawback that the calculus quickly becomes untractable because many forms of interference can occur.

Interference is one of the trickiest phenomena that can be found in concurrency. Interference is caused by two, or more, reduction rules which are eligible and which share one of the interaction partners. We can make the distinction between *plain* interference and *grave* interference (for a formal definition refer to [3]). The first occurs when a process may engage the "same" interaction with two different partners, intuitively a process is offering a service and finds at least two processes willing to react. Plain interference may be desirable, for example to model nondeterminism. In CCS or in π-calculus plain interference is the only form we can find.

In Ambient Calculus (and generally in all calculi in which logically distinct interaction can share terms) we find another form of interference, which may be called *grave* interference. The origin is the same, a term shared by two different redexes: but in contrast with the previous form of interaction, the shared term may be involved in two logically different operations. For example, we can easily imagine a term in which an ambient is subject to an `open` operation and to an `in` operation. Depending on which action will fire first, the shared ambient will be dissolved or will hide himself inside another ambient. The presence of grave interference might be regarded as a programming error: in this hypothesis, the calculus should try to limit it.

Because of the existence of grave interference it is extremely difficult to write programs which behave correctly in all contexts, and it is difficult to prove behavioral properties of processes: the algebraic theory of Ambient Calculus is poor.

Trying to constraint interference at the level of the syntax leads to impose draconian restrictions on the calculus, altering its nature. On the contrary, the types level appears to be the right one.

Mobility and immobility

In many calculi we find syntactic distinctions used to denote mobile and immobile locations and/or processes: the same purpose can be achieved with a type system.

In Ambient Calculus the mobility unit is the ambient: we can easily annotate the type of each ambient to include informations about the possibility of movement. The mobility annotations can be extremely simple, such as *mobile* and *immobile*, with some laws which constraint the well-typed terms. Some examples of such laws are:

1. if a process contains a movement capability, then it must be regarded as mobile,

2. an immobile ambient can not contain any process labeled as mobile,

and so on. The laws should be fairly intuitive: a programmer should always be capable of imagine if a term is well-typed or not and type errors should appear evident when reported by the type checker, otherwise a type system is not of any help in writing correct programs. Besides, laws must be statically checked: *the type as specification metaphor ends where the undecidability of the specification begins.*

In such kind of type system, the notion of inconsistent interaction has been chosen in order to guarantee for a well typed system security properties, such as

> nobody will never be able to move an immobile ambient, or
> nobody will never be able to open a sealed ambient.

Single Threadness

A type system which removes all forms of grave interference is studied by Levi and Sangiorgi in [20]. Their aim is to exploit the type system to prove some laws which relate reduction relation and behavioral equivalence, developing the kernel of the set of equivalences which forms an algebraic theory.

The capabilities of the local processes of an ambient control the behavior of this ambient: in the base calculus each process can exercise a capability, causing unwanted overlappings of interactions. In a *Single Threaded* ambient, by contrast, at any moment at most one process has the control thread, and may therefore exercise a capability. In consequence of it, a Single Threaded ambient can engage at most one interaction at a time with external or internal ambients. It is clear that this restriction eliminates grave interference, since it avoids *a priori* the existence of overlapping eligible transition rules. On the other hand, we do not want to obtain a purely sequential calculus, and inside an ambient many activities can take place concurrently: reductions may take place inside a subambient, two subambients may interact and messages can get consumed. The thread of control can be passed among processes inside an ambient by means of messages, and non-determinism in the receive operation is allowed.

The type system developed is one of the most interesting and most subtle examples of what can be achieved by imposing a static discipline on a calculus.

2.3.3 The expressivity of Typed Ambient Calculus

The absence of a reference model has an important consequence: studying the expressiveness of a newly defined calculus is quite puzzling. In particular it is unclear what a language should be able to do in order to result *expressive enough*.

What can be done is trying to encode some well known and widely used patterns of interaction, and observing how clumsy the coding is. Patterns may be inspired from the theoretical world, as well as from the actual Internet technology.

In all the Ambient papers series, the various versions of the calculus are checked against communication on typed channels —that is, an encoding of typed π-calculus (but without recursive types)—, an encoding of Call-by-Value λ-calculus, and more interestingly some stripped-down version of actual distributed programming languages.

2.4 Other applications of type systems

The notion of inconsistent interaction is a pervasive one. It can be adapted to reveal useful in the most different domains, and it should not be surprising if type systems arise in process calculi to prove the most disparate security properties.

The previous section introduced us to a type system which explicitly deals with flow of control; now we will proceed by briefly reviewing other highly influential proposals. What should be clear is that this is just a personal selection. By no means does the development of type systems stop here: there is always more slack to pick up.

Communication control

In any implementation of a distributed system, there is a tension between the expressivity of the communication mechanism and the cost of its implementation. In particular, communication between threads in the same machine can be implemented with shared data structures while remote communications cannot, with awful consequences on the latency of the operation.

Peter Sewell in "Global/Local Subtyping for a Distributed π-calculus" [31] develops the idea of using a type system to detect which channels are guaranteed to be local to one processor, and which are not. He starts with a dialect of Distributed π-calculus. Different localities exist in the syntax, interaction takes place on named channels but the syntax of the calculus does not localize them explicitly. But if a channel name is used only by processes inside the same location, it can safely be considered local and implemented rather with shared data structures than remote communication mechanism.

He extends the Pierce and Sangiorgi's subtyping relation on channel types, introducing local channels as a subtype of global channels: the resulting type system is extremely expressive and far less complicated than equivalent static analysis techniques. He also introduces also a type reconstruction algorithm which takes a term and gives back the least type for every channel used. These informations can be used successively by a compiler to implement the system, and at the same time the programmer can control if channels are used as he intended.

Resource Access Control

Different locations offer different services and different resources to local processes and this is one of the reason that cause a process to migrate. Locations can impose access policies to local resources, in order to limit the effects of the migrating code on local environments. Yoshida and Hennessy, in "Assigning Types to Processes" [35], propose a fine-grained type system in which the type of a process takes the form of an interface limiting the resource to which it has acces. In the simple calculus they use, the only resources at any locations are channels to which the process may wish to accede to perform reading, writing, or both. A location, before accepting an agent, can check statically the resource it is interested in, rejecting the agent if it does not respect the access policies of the particular location.

Leaking secrets

It is naïve think that hostile agents do not exist in an etherogeneus network as Internet. There is a lot of work in the direction of developing secure system, as well as trying to add security properties to an already existing system.

The *spi*-calculus of Abadi and Gordon [1] goes in the first direction: they extends the syntax of π-calculus with cryptography primitives, and they study the security protocols used in real world from an abstracted point of view. Working at the type level, two remarkable works are those of Cardelli, Ghelli, and Gordon [12] and of Sewell and Vitek [32].

Because the nature of the *parallel* operator, virtually every process can interact in a malicious way with our system, where malicious means trying to disrupt the system, as well as trying to discover some reserved informations. Paradoxically, this second type of attack is the most dangerous since it may leave no tracks behind him.

Adding to the π-calculus the notion of *group of names* is a brilliant idea to restrict the unlimited extrusion of names: when groups of names are defined, a name is free to be extruded only inside the boundaries induced by the scope of the group name belongs to. In particular, they are able to guarantee that opportunately restricted names will remain secret also in presence of untyped attackers. Major drawback of the approach is the faith in the respect of the semantics of the calculus: a real attacker may have been developed in order to being able to perform non-standard actions, without respecting the semantics and invalidating the security properties.

The same problem appears in an apparently unrelated field. In the development of big software projects software-reuse is a standard and fundamental technique. This implies giving to an untrusted or partially trusted off-the-shelf components a strong control on the whole system. Using *wrapper programs* to encapsulate components and enforce security policies reveals a good technique to prevent unattended flows of data: for instance, we may want that the libraries send informations to our components, but we may want to prevent the opposite flow of data. The approach of Sewell and Vitek makes use of a *causal type system* that statically captures the allowed flows between wrapped components. Their type system is useful in proving properties of the wrappers programs, the entity of the system which allows data-exchanges among components: for instance it can statically check that an unidirectional wrapper let informations flow in a direction and effectively forbids it in the other. In this case the inconsistent interaction follows from the definition of a *coloured reduction semantics* which keeps track of the causal flow.

3

Untyped Seal Calculus

3.1 Seal at a glance

Seal Calculus lies in the large design space of distributed calculi which are built on the π-calculus. The principal ingredient added, the *seal*, is the generalization of the common notions of *agent* and *location*. Like a location, a seal bears interactions inside itself; like an agent, a seal can be moved across locations where it can interact with other agents. Seals are named, and can be arbitrarily nested generating a hierarchical structure. We will use the terms parent and child seal to move across the tree of locations. The seals of Seal Calculus do not differ conceptually from the ambients of the Ambient Calculus; similarities between the two calculi ends here.

Only processes and sub-seals sit inside a seal; as before, we denote the set of processes contained in a seal, but in none of its sub-seals, as the *processes local to a seal*.

Interaction takes place on named localized channels. Channels come in two flavors, *local* and *shared*.

- A local channel is a channel that allows interactions among local processes, and it is inaccessible to the other processes.
- A shared channel is a channel that allows interactions among processes local to two seals in parent-child relationship, and it is inaccesible to the other processes.

An interaction can consist in the synchronous communication of a name between two processes that can access the same channel. Alternatively, to mimic code mobility, a whole seal (including its active processes and sub-seals) can be sent over a channel: when the seal is received it is reactivated inside the same seal of the receiving process. In this way mobility is achieved, by sending and receiving seals over shared channels. The term *agent* will be used to mean both processes and seals, as they are respectively the unit of action and of movement of the calculus.

A seal is eternal: there is no way to dissolve the barriers which constitutes a seal.

Underlying in the Seal calculus design is the principle that each seal should be able to control all interactions of its children, both with the outside world and with each other.

3.2 Untyped Seal Calculus

We assume an infinite set of *names*, ranged over by x,y, and z. In the syntax there is no distinction between names and variables, as there is no distinction between channel names and seal names.

The set of *processes* is defined by the following grammar:

$$
\begin{array}{rcll}
P, Q, R & ::= & \mathbf{0} & \text{inactivity} \\
& | & P \mid P & \text{composition} \\
& | & !P & \text{replication} \\
& | & x[P] & \text{seal} \\
& | & (\boldsymbol{\nu}\, x)P & \text{restriction} \\
& | & \alpha.P & \text{action}
\end{array}
$$

Intuitively, the constructions of the Seal Calculus have the following meanings:

- The *dead* process $\mathbf{0}$ simply does nothing.
- A *composition* $P \mid Q$ behaves as processes P and Q running in parallel. Each process may interact with each other on channels known to both, or with the outside world, independently of each other.
- The *replication* ! operator produces as many copies as needed of the process it is applied to. Each copy is then free to interact with its environment, and with the other copies as well.
- A *restriction* $(\boldsymbol{\nu}\, x)P$ denotes a process P in which a new, private name x has been allocated.
- A *seal* $x[P]$ is the representation in the syntax of a place named x that is delimited by boundaries, and where computation P takes place.
- An *action* $\alpha.P$ denotes a process waiting for the interaction α to take place.

The bare structure of the processes is exactly the same we find in Ambient Calculus: what characterizes Seal is the set of possible interactions in which processes may be involved.

The set of actions α, together with the set of locations η, are defined by the grammar below.

$$
\begin{array}{rcll}
\alpha & ::= & x^\eta(y) & \text{input} \\
& | & \overline{x}^\eta(y) & \text{output} \\
& | & \overline{x}^\eta\{\!|y|\!\} & \text{send} \\
& | & x^\eta\{\!|y|\!\} & \text{receive}
\end{array}
\qquad
\begin{array}{rcll}
\eta & ::= & * & \text{local} \\
& | & \uparrow & \text{up} \\
& | & z & \text{down}
\end{array}
$$

The basic computational step in Seal is *interaction*. In interaction a name x is communicated from an output process to an input process, or a seal is moved. All interactions take place on named localized channels. Location denotations $*$, $\uparrow$, and z denote respectively the current seal, the parent seal, and a child-seal named z. Is such way, a local channel named x will be denoted by x^*, a shared channel which admits interactions among processes in the current seal and in the parent seal will be denoted by $x^\uparrow$, a shared channel which admits interactions among processes in the current seal and in a child seal named z will be denoted by x^z.

- An *output process* $\overline{y}^\eta(x).P$ is ready to output on channel y^η. If an interaction occurs, name x is communicated on y and then process P runs.
- An *input process* $y^\eta(x).P$ is ready to perform an input on channel y^η. If an interaction occurs, the received name is bound to x in the continuation P: in other words, if an interaction occurs and z is the received name, then process P, in which each occurrence of y has been replaced by z, runs.
- A *send process* $\overline{y}^\eta\{\!\{x\}\!\}.P$ is ready to send a seal named x, which have to be a local seal, on the channel y^η. If an interaction occurs (alas, there is a sibling seal named x and there is a receiver process) then it behaves as P.
- A *receive process* $y^\eta\{\!\{x\}\!\}.P$ is ready to receive a seal on the channel y^η. If an interaction occurs, then the process P runs in parallel with the received seal renamed as x.

Before looking for the formal semantics, we introduce some convenient syntactic abbreviations.

$(\boldsymbol{\nu}\, x).P \mid Q$	is read	$((\boldsymbol{\nu}\, x).P) \mid Q$	
$!P \mid Q$	"	$(!P) \mid Q$	
$\alpha.P \mid Q$	"	$(\alpha.P) \mid Q$	
$(\boldsymbol{\nu}\, x_1, \cdots, x_n)\ P$	"	$(\boldsymbol{\nu}\, x_1) \cdots (\boldsymbol{\nu}\, x_n)\ P$	
$x[]$	"	$x[\mathbf{0}]$	
P	"	$P.\mathbf{0}$	
$\alpha.P \mid Q$	"	$(\alpha.P) \mid Q$	
$x^l().P$	"	$x^l(y).P$	if $y \notin fn(P)$
$\overline{x}^l().P$	"	$\overline{x}^l(y).P$	if y is not important

In communication operations like $x^l(y)$, we refer to x as the *subject name*, and to y as the *object name*.

3.3 Operational semantics

We formally define the operational semantics of the Seal calculus in the style of Milner's π-calculus semantics [24], which was itself inspired by Berry and Boudol's Chemical Abstract Machine [5]. We will focus on internal reduction of processes by means of reduction axioms and a structural congruence relation.

3.3.1 Free names and substitution

The only binders in the Seal are the $(\boldsymbol{\nu}\, x)$ prefix and the input prefix $x^{\eta}(y)$. This leads to the following definition of the set of *free names* of a term:

$$
\begin{array}{rcl}
fn(\mathbf{0}) & = & \emptyset \\
fn(P \mid Q) & = & fn(P) \cup fn(Q) \\
fn(!P) & = & fn(P) \\
fn(x[P]) & = & fn(P) \cup \{x\} \\
fn((\boldsymbol{\nu}\, x)P) & = & fn(P) \setminus \{x\} \\
fn(x^{\eta}(y).P) & = & (fn(P) \setminus \{y\}) \cup \{x\} \cup fn(\eta) \\
fn(\overline{x}^{\eta}(y).P) & = & fn(P) \cup \{y\} \cup \{x\} \cup fn(\eta) \\
fn(\overline{x}^{\eta}\lmoustache y \rmoustache.P) & = & fn(P) \cup \{y\} \cup \{x\} \cup fn(\eta) \\
fn(x^{\eta}\lmoustache y \rmoustache.P) & = & fn(P) \cup \{y\} \cup \{x\} \cup fn(\eta) \\
fn(\uparrow) & = & \emptyset \\
fn(*) & = & \emptyset \\
fn(x) & = & \{x\}
\end{array}
$$

We will always work up to alpha renaming of bound names, so whenever we write a term we actually mean its alpha equivalence class.

We will write $P\{{}^{y}\!/_{x}\}$ to denote the process in which every free occurrence of x has been replaced by y, renaming if necessary to avoid capture.

3.3.2 Structural congruence

To make our reduction system simple, we wish to identify several expressions whose behavior always ought to be equivalent. To accomplish this, we define a *structural equivalence* relation, noted $P \equiv Q$, as the smallest relation over P which satisfies the rules in Fig. 3.1. As its name suggests, we require $\equiv$ to be an equivalence relation:

$$P \equiv P \qquad P \equiv Q \Rightarrow Q \equiv P \qquad P \equiv Q \wedge Q \equiv R \Rightarrow P \equiv R$$

It must also be a congruence (it must be preserved by arbitrary context):

$$
\begin{array}{l}
P \equiv Q \Rightarrow (\boldsymbol{\nu}\, x)P \equiv (\boldsymbol{\nu}\, x)Q \\
P \equiv Q \Rightarrow P \mid R \equiv Q \mid R \\
P \equiv Q \Rightarrow x[P] \equiv x[Q] \\
P \equiv Q \Rightarrow \alpha\,.\,P \equiv \alpha\,.\,Q
\end{array}
$$

The $\mid$ operator has a structure of commutative monoid and its zero is the dead process:

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

One rule deals with the behavior of the ! operator:

$$!P \equiv !P \mid P$$

It says that a process being replicated is equivalent to the number of copies of the process are needed. The ! operator is usually called *bang*, for its resemblance with the ! operator of Girard's Linear Logic.

The following two rules are not required to calculate the reduction steps: they are usually referred to as the *garbage collection* rules. Both state that defining a name local to a process which does not use it — the dead process in the first case, a process which does not explicitly mention the newly created name in the second — "is not really useful".

$$\begin{array}{ll} (\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0} & \\ (\boldsymbol{\nu}\, x)P \equiv P & \text{for } x \notin \mathit{fn}(P) \end{array}$$

Please notice that $(\boldsymbol{\nu}\, x)P \equiv P$ for $x \notin \mathit{fn}(P)$ can be derived from the other rules (we need also the rules we have not yet explained):

$$(\boldsymbol{\nu}\, x)P \equiv (\boldsymbol{\nu}\, x)P \mid \mathbf{0} \equiv (\boldsymbol{\nu}\, x)(P \mid \mathbf{0}) \equiv P \mid (\boldsymbol{\nu}\, x)\mathbf{0} \equiv P \mid \mathbf{0} \equiv P$$

The last three rules, the *extrusion* rules, define the conditions to enlarge and to restrict the scoping of a name. The first and the last are quite standard in π-calculi literature, while the second is the natural extension of the first to agent calculi.

$$\begin{array}{ll} (\boldsymbol{\nu}\, x)(P \mid Q) \equiv P \mid (\boldsymbol{\nu}\, x)Q & \text{for } x \notin \mathit{fn}(P) \\ (\boldsymbol{\nu}\, x)y[P] \equiv y[(\boldsymbol{\nu}\, x)P] & \text{for } x \neq y \\ (\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P \equiv (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P & \end{array}$$

Both states that in absence of name clash the scoping of a name can be arbitrarily extended or reduced.

$P \equiv P$		(Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$		(Struct Comm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$		(Struct Trans)
$P \mid \mathbf{0} \equiv P$		(Struct Dead Par)
$P \mid Q \equiv Q \mid P$		(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$		(Struct Par Assoc)
$!P \equiv !P \mid P$		(Proc Repl)
$(\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0}$		(Struct Res Dead)
$(\boldsymbol{\nu}\, x)P \equiv P$	for $x \notin \mathit{fn}(P)$	(Struct Res Garb)
$(\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P \equiv (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P$		(Struct Res Res)
$(\boldsymbol{\nu}\, x)(P \mid Q) \equiv P \mid (\boldsymbol{\nu}\, x)Q$	for $x \notin \mathit{fn}(P)$	(Struct Res Par)
$(\boldsymbol{\nu}\, x)y[P] \equiv y[(\boldsymbol{\nu}\, x)P]$	for $x \neq y$	(Struct Res Agent)
$P \equiv Q \Rightarrow (\boldsymbol{\nu}\, x)P \equiv (\boldsymbol{\nu}\, x)Q$		(Struct Cong New)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$		(Struct Cong Par)
$P \equiv Q \Rightarrow x[P] \equiv x[Q]$		(Struct Cong Agent)
$P \equiv Q \Rightarrow \alpha . P \equiv \alpha . Q$		(Struct Cong Action)

Figure 3.1: Structural congruence $P \equiv Q$

A key fact of these rules is that all *unguarded* restrictions can be pulled out to the top level of an agent, up-to respecting their ordering. This can be considered a *canonical form* of a process, and structural rules can be viewed as a *heating* phase preceding the reduction.

We remark that alpha equivalent terms are to be considered identical (for example, by choosing an appropriate representation), as opposed to structurally equivalent.

3.3.3 Reduction rules

The behavior of a process is given by the reduction rules summarized in Figure 3.2. The first six rules are the base one-step reductions for communication and mobility.

$$
\begin{array}{rcl}
x^{*}(u).P \mid \overline{x}^{*}(v).Q & \rightarrow & P\{^{v}/_{u}\} \mid Q \\
x^{y}(u).P \mid y[\overline{x}^{\uparrow}(v).Q \mid R] & \rightarrow & P\{^{v}/_{u}\} \mid y[Q \mid R] \\
\overline{x}^{y}(v).P \mid y[x^{\uparrow}(u).Q \mid R] & \rightarrow & P \mid y[Q\{^{v}/_{u}\} \mid R]
\end{array}
$$

$$
\begin{array}{rcl}
x^{*}\{u\}.P \mid \overline{x}^{*}\{v\}.Q \mid v[B] & \rightarrow & P \mid u[B] \mid Q \\
x^{y}\{u\}.P \mid y[\overline{x}^{\uparrow}\{v\}.Q \mid v[R] \mid S] & \rightarrow & P \mid u[R] \mid y[Q \mid S] \\
\overline{x}^{y}\{v\}.P \mid v[R] \mid y[x^{\uparrow}\{u\}.Q \mid S] & \rightarrow & P \mid y[Q \mid S \mid u[R]]
\end{array}
$$

The next three rules propagate reduction across parallel composition, scopes, and seal nesting.

$$
\begin{array}{rcl}
P \rightarrow Q & \Rightarrow & P \mid R \rightarrow Q \mid R \\
P \rightarrow Q & \Rightarrow & (\boldsymbol{\nu}\, x)P \rightarrow (\boldsymbol{\nu}\, x)Q \\
P \rightarrow Q & \Rightarrow & u[P] \rightarrow u[Q]
\end{array}
$$

In particular the last rule implies that a process inside a seal is always running, even when the seal is moving.

The final rule allows the use of structural equivalence during reduction.

$$P \equiv P' \wedge P' \rightarrow Q' \wedge Q' \equiv Q \;\Rightarrow\; P \rightarrow Q$$

$x^{*}(u).P \mid \overline{x}^{*}(v).Q$	$\rightarrow$	$P\{^{v}/_{u}\} \mid Q$	(write local)
$x^{y}(u).P \mid y[\overline{x}^{\uparrow}(v).Q \mid R]$	$\rightarrow$	$P\{^{v}/_{u}\} \mid y[Q \mid R]$	(write out)
$\overline{x}^{y}(v).P \mid y[x^{\uparrow}(u).Q \mid R]$	$\rightarrow$	$P \mid y[Q\{^{v}/_{u}\} \mid R]$	(write in)
$x^{*}\{u\}.P \mid \overline{x}^{*}\{v\}.Q \mid v[B]$	$\rightarrow$	$P \mid u[B] \mid Q$	(move local)
$x^{y}\{u\}.P \mid y[\overline{x}^{\uparrow}\{v\}.Q \mid v[R] \mid S]$	$\rightarrow$	$P \mid u[R] \mid y[Q \mid S]$	(move out)
$\overline{x}^{y}\{v\}.P \mid v[R] \mid y[x^{\uparrow}\{u\}.Q \mid S]$	$\rightarrow$	$P \mid y[Q \mid S \mid u[R]]$	(move in)
$P \rightarrow Q$	$\Rightarrow$	$P \mid R \rightarrow Q \mid R$	(red par)
$P \rightarrow Q$	$\Rightarrow$	$(\boldsymbol{\nu}\, x)P \rightarrow (\boldsymbol{\nu}\, x)Q$	(red res)
$P \rightarrow Q$	$\Rightarrow$	$u[P] \rightarrow u[Q]$	(red agent)
$P \equiv P' \wedge P' \rightarrow Q' \wedge Q' \equiv Q$	$\Rightarrow$	$P \rightarrow Q$	(red $\equiv$)

Figure 3.2: Reduction rules.

3.4 A parallel calculus?

In order to familiarize with the syntax of the Seal Calculus, we show that Seal is a *very parallel* calculus[1]. Not a long time ago, people studying Denotational Semantics had some problems with the *parallel-or*: it is a logical operator which operates according the following truth table

	t	f	$\perp$
t	t	t	t
f	t	f	$\perp$
$\perp$	t	$\perp$	$\perp$

In Seal, Computation can be carried on in the π-calculus core, and we could encode the parallel-or operator using only local communications in the style of Milner's encoding of boolean. But this is not amazing (and maybe not much instructive), so we encode boolean values using a mixture of communication and mobility.

We localize the bolean values at a channel, in the style of the π-calculus encoding, and we abstract the encoding from the actual names for true and false.

$$\begin{aligned} \text{True at } b &\stackrel{def}{=} b^*(t).b^*(f).t[] \\ \text{False at } b &\stackrel{def}{=} b^*(t).b^*(f).f[] \end{aligned}$$

Thus, a true (resp. false) value localized at b will appear as a process listening on the channel b for the names for true and false, and relases at the current level a seal (with no active gremlins inside) named true (resp. false).

The base operation on truth values is *case-analysis*: activating a process or another according to a boolean value.

$$\begin{aligned} &\texttt{if } b \texttt{ then } P \texttt{ else } Q \stackrel{def}{=} \\ &\quad (\boldsymbol{\nu}\, t, f)(\overline{b}^*(t).\overline{b}^*(f)(\boldsymbol{\nu}\, g, z)(\overline{z}^g\{t\}.P \mid \overline{z}^g\{f\}.Q \mid g[(\boldsymbol{\nu}\, p)z^{\uparrow}\{p\}])) \end{aligned}$$

where $t, f, g, z \notin fn(P)$ and $t, f, g, z \notin fn(Q)$. Our encoding initializes the boolean located at b, and then in parallel tries to send a seal named t or f inside another seal named g (garbage). The boolean located at b will release according to its a value a seal named t or f at the current level, so only one of the two send will succed, activating its continuation. The seal g receives the seal, renames it into a new local name p (poubelle) in order to garbage collect it.

[1] I am grateful to Juliusz Chroboczeck for the suggestion.

For instance,

$$\begin{array}{rl}
\multicolumn{2}{l}{\texttt{if } b \texttt{ then } P \texttt{ else } Q \mid \text{False at } b} \\
\equiv & (\boldsymbol{\nu}\, t, f)(\overline{b}^*(t).\overline{b}^*(f)(\boldsymbol{\nu}\, g, z)(\overline{z}^g\{t\}.P \mid \overline{z}^g\{f\}.Q \mid g[(\boldsymbol{\nu}\, p)z^{\uparrow}\{p\}])) \\
& \mid b^*(t).b^*(f).f[] \\
\rightarrow^* & (\boldsymbol{\nu}\, t, f, g, z)(\overline{z}^g\{t\}.P \mid \overline{z}^g\{f\}.Q \mid g[(\boldsymbol{\nu}\, p)z^{\uparrow}\{p\}] \\
& \mid f[]) \\
\rightarrow & (\boldsymbol{\nu}\, t, f, g, z)(\overline{z}^g\{t\}.P \mid Q \mid g[(\boldsymbol{\nu}\, p)p[]]) \\
\succ & Q
\end{array}$$

as attended. The last step is quite intuitive, and will not be discussed in this work.

We are now ready to take fun with the parallel-or: to avoid a too clumsy term, we avoid parametrizing it about the two values, encoding the *por* between the boolean values b and c.

$$\begin{array}{l}
b \; por \; c \stackrel{def}{=} \\
\quad por^*(t).por^*(f). \\
\quad (\boldsymbol{\nu}\, t', f')(\overline{b}^*(t').\overline{b}^*(f') \mid \overline{c}^*(t').\overline{c}^*(f') \mid \\
\quad (\boldsymbol{\nu}\, g, z)(\overline{z}^g\{t'\}.t[] \mid \overline{z}^g\{f'\}.\overline{z}^g\{f'\}.f[] \mid \\
\quad g[!(\boldsymbol{\nu}\, p)z^{\uparrow}\{p\}]))
\end{array}$$

The result is to be read as a boolean value located at the location *por*. The term initializes in parallel the two booleans located at b and c: receiving two false, or at least one true, will unleash at the current level respectively a seal named false and a seal named true. The partial results are garbage collected as before inside the seal named g (which must be ready to do a lot of work and so is replicated with the bang operator).

The reader could take even more fun, operationally executing the reduction steps of a term which denotes the parallel-or among true and a livelocked process.

4

Typed Seal Calculus

4.1 Typing Mobility

In Chapter 2 we described how to type communication in π-calculus. This is not enough for Seal: we also have to type mobility, besides communication. This means that we do not want to be ensured only that appropriate values will transit over channels of a given type, but we also want to ensure that the movement of a seal does not invalidate the protocol to cooperate with it. We need two ingredients to statically achieve this: a way to characterize the possible interactions of a seal with the surrounding environment, and a way to guarantee that the previous characterization is invariant with respect to the seal mobility.

All interactions with the environment have to take place by means of the channels passing across the seal's boundary: keeping track of the set of communications in which the processes local to a seal may eventually be interested on is not only possible but also fruitful. We will call *upper channels* the channels characterized by a $\uparrow$ localization and we shall frame the key definition of this work.

> We define the *interface* of a seal as the set of upper channels which the process local to the seal may be listening on, with the type expected from interactions on them.

Knowing the interface of a seal permits to deal with it without fearing that communicating inside the seal a value of the wrong type could destroy the agent.

The movement of a seal take place only over channels. Thus, to guarantee the invariance of a seal's interfaces with respect to mobility, the type system allows to declare some channels *mobility channels*, as opposed to *communication channels*. The type of a mobility channel describes it as a channel where agents of a given interface may be sent over: this is enough to undertake that the interface associated to a seal does not change during the movement.

Services vs. effects

The interfaces look like the types of the objects in the "objects as records" analogy: just an array of methods we can invoke. In fact, the analogy between agents and objects is not a piece of news, from the first approach to concurrency in the Actor model [16], to the latest developments [6], through many classical works.

In this work, there is an important distinction to point out from the object analogy. In the object framework, sending a message should result in a method being activated: the type of an object reports the set of messages the object will answer to. We can say that the interface of an object characterizes the *services* that the object can offer to its environment.

According to our definition, a channel which appears in the interface of an agent does not guarantee that interaction on this channel is always going to happen. It is better to say that the interface limits the *effects* the agent can have on the environment: if an interaction occurs, it occurs on a channel defined in the interface and not on other channels.

Imagine a vending machine always ready to accept our coins and to give us some choccolate or some coffe, hopefully according our selection. The interface of the machine would consist in the drop slot for the coins and in the buttons to exercise our choice. When the machine is broken, we drop our coin, we exercise our will but we never get anything from the machine: we offered an interaction on a channel in the interface, the button to select coffee —the drop slot never breaks down—, but interaction never occurs.

Reasoning in terms of effects is directly derived from concurrency practice and every day's life.

4.1.1 Type syntax

The syntax of the types is reported in the following table.

V	$::= M$	messages
	$\mid A$	agents
M	$::= \texttt{Shh}$	silent
	$\mid \texttt{Ch}\ V$	channel names
	$\mid \texttt{Id}\ A$	agent names
A	$::= [x_1{:}V_1; \cdots ; x_n{:}V_n]$	agent's interfaces
T	$::= M$	messages
	$\mid \texttt{Proc}\ A$	processes

There are four syntactic categories in the type syntax used respectively to denote values, messages, agents, and types. The intuition behind the syntax of the types is the following one:

V: The syntactic category V is used to classify *values* in the calculus. A value is a computational entity that can be sent over a channel. While in π-calculus values are just channel names, in Seal we have both messages M —which includes base values, channel names and agent names— and seals A.

M: The category M is used to classify *messages*. A message can be a name or a base value, such as an integer or a string.

The only base values we keep consider are the synchronization messages: $\vdash x : \texttt{Shh}$ means that x is just a synchronization message carrying no information. Other base values can be easily added in real implementations.

Names are used to denote the entities of calculus. In the syntax there are no distinctions between channel names and seal names, so this useful distinction is introduced at the type level:

$\vdash x : \texttt{Ch}\ V$ means that x is the name of a channel which transports values of type V.

$\vdash x : \texttt{Id}\ A$ means that x is the name of a seal with interface A.

A: The category A is used to classify the *agents* of the calculus, keeping track of their interface.

The notation $[x_1{:}V_1; \cdots; x_n{:}V_n]$ is used to record the information about the interface of an agent, and is syntactic sugar to represent a set of couples *name* : *type*. The field *name* of a couple identifies uniquely the couple inside the set. An agent with a given interface may listen from the upper level only on channels that appear in the interface; furthermore we know also the type of the information it can possibly read on a given channel that appears in the interface.

T: There are two syntactical categories in the Seal syntax —names and processes— and, in the same way, there are two syntactical categories in the type syntax. The former, M, refers to messages, as described above. On the other hand, assigning a type $\texttt{Proc}\ A$ to a process P means that P may perform input operations on the channels that appear in the interface A.

As the notation can look cumbersome, here follow some examples:

- A name of a seal which does not perform any input (it can still send something outside himself): $x : \texttt{Id}\ []$.
- A synchronization channel: $\texttt{Ch Shh}$.
- $[x_1{:}\texttt{Shh}, x_2{:}\texttt{Ch}\ V, x_3{:}A, x_4{:}\texttt{Id}\ A]$ is an interface which characterizes an agent that may:
  - synchronize with an input operation on the upper channel x_1,
  - read over x_2 a channel name of type $\texttt{Ch}\ V$,
  - receive over x_3 a seal whose interface is A,
  - read over x_4 a seal name of type $\texttt{Id}\ A$.

  It is really important to understand the difference between what can be transmitted over x_3 and x_4, respectively seals and seal names: the former requires mobility primitives, the latter communication primitives.

4.1.2 Ideas about interfaces

Let us paraphrase Einstein: the notion of interface, although simple, is not simpler. It introduces names of the calculus at the type level: consider the process

$$P = (\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Ch}\ M)(\boldsymbol{\nu}\, w, y{:}\mathtt{Ch}\ M)\, \overline{x}^{*}(w) \mid x^{*}(y{:}\mathtt{Ch}\ M).y^{\uparrow}(z{:}M)$$

The process P can perform an internal reduction on the channel x resulting in w bound to y, and then offering an input from $\uparrow$ on the channel w. Forgetting for a while its dynamic and observing only its syntax it appear as a process that someday may offer an input from $\uparrow$ on the channel y, thus justifying the following informal typing judgment

$$\vdash P : \mathtt{Proc}\ [y{:}M].$$

If P performs the internal communication on the channel x

$$\underset{\mathtt{Proc}\ [y:M]}{P} \quad \rightarrow \quad \underset{\mathtt{Proc}\ [w:M]}{(\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Ch}\ M)(\boldsymbol{\nu}\, w, y{:}\mathtt{Ch}\ M)\ w^{\uparrow}(z{:}M)}$$

the interface of the process changes. And this is not good, to be optimistic.

This is not a new problem in the community: to solve it one may consider using dependent types and deal explicitly with types that change during computation. Dependent types work fine for calculi where the notion of interaction is syntactically well determined, as λ-calculus. Unfortunately in process calculi, where interaction is a consequence of parallel composition (which admits arbitrary rearrangements of sub-terms), all the essays are somewhat unsatisfactory: usually they are usually restricted to a subset of the calculus, allowing dependent types only in particular, well determined, constructions.

Following a suggestion of Davide Sangiorgi, we decide to disallow input operation on names bound in an input operation, recording in interface only the channels on which the process perform an input operation.

This is an elegant way to enforce statically that interfaces do not change during reduction: for example the process P is not well-typed, since y is first bound during an input on x, and then used to perform an input from $\uparrow$.

To make the type system uniform, we impose this condition on all the input operation, not only on the input operations from $\uparrow$, which are the only ones determining the interface.

This restriction surprisingly does not really limit the expressive power of the calculus: nearly all programming languages based on π-calculus impose this constraint, while programs written in concurrent languages which do not, magically seem to obey to the same condition. Concurrent calculi are far too non-deterministic, and the programmer is naturally forced to find some programming schemes to impose some control on the language.

4.1.3 Operations on Interfaces

There is an important operator defined on the interfaces, in order to be able to manipulate them. The $\oplus$ operator merges two interfaces. The two interfaces must agree on the type of the channels defined in both, while all other couples *name:type* are simply added. In other words, the $\oplus$ operator is nothing more than a common union of two sets of couples, with the condition that there may exist no more than a couple with a given first element. Formally,

$$[x_1{:}V_{x_1},\ldots,x_n{:}V_{x_n},y_1{:}V_{y_1},\ldots,y_m{:}V_{y_m}] \oplus [x_1{:}V_{x_1},\ldots,x_n{:}V_{x_n},z_1{:}V_{z_1},\ldots,z_k{:}V_{z_k}] =$$

$$[x_1{:}V_{x_1},\ldots,x_n{:}V_{x_n},y_1{:}V_{y_1},\ldots,y_m{:}V_{y_m},z_1{:}V_{z_1},\ldots,z_k{:}V_{z_k}]$$
$$\text{where } \forall\, i,j\ y_i \neq z_j.$$

If the same label appears bound to two different types in A_1 and A_2, then $A_1 \oplus A_2$ is undefined.

It may be noteworthy to remark that $(A, \oplus, [\,])$ is a commutative monoid.

4.2 Typed Seal Calculus

We summarize the Typed Seal Calculus syntax and semantics, pointing out the differences between the typed and the untyped version.

Syntax

The set of *processes* is defined by the following grammar:

types

$V ::= M$	messages	
$\mid\ A$	agents	
$M ::= \texttt{Shh}$	silent	
$\mid\ \texttt{Ch}\ V$	channel names	
$\mid\ \texttt{Id}\ A$	agent names	
$A ::= [x_1{:}V_1;\cdots;x_n{:}V_n]$	agent's interfaces	
$T ::= M$	messages	
$\mid\ \texttt{Proc}\ A$	processes	

locations

$\eta ::= * \quad\mid\quad \uparrow \quad\mid\quad z$

processes

$P,Q,R ::= \mathbf{0}$	inactivity
$\mid\ P \mid P$	composition
$\mid\ !P$	replication
$\mid\ x[P]$	seal
$\mid\ (\boldsymbol{\nu}\, x{:}M)P$	restriction
$\mid\ \alpha.P$	action

actions

$\alpha ::= x^\eta(y{:}M)$	input
$\mid\ \overline{x}^\eta(y)$	output
$\mid\ \overline{x}^\eta\{\!\!\{y\}\!\!\}$	send
$\mid\ x^\eta\{\!\!\{y\}\!\!\}$	receive

In the syntax of the types there are no binders and the definition of the set of *free names* of a type keeps track of all the names which appear in it. The definition is straightforward:

$$\begin{array}{rcl} fn(\mathtt{Shh}) &=& \emptyset \\ fn([x_1{:}V_1;\ldots;x_n{:}V_n]) &=& \{x_1,\ldots,x_n\}\cup fn(V_1)\cup\cdots\cup fn(V_n) \\ fn(\mathtt{Ch}\ V) &=& fn(V) \\ fn(\mathtt{Id}\ A) &=& fn(A) \\ fn(\mathtt{Proc}\ A) &=& fn(A) \end{array}$$

The definition of the free names of a process changes in order to keep track of the names which appear in the types:

$$\begin{array}{rcl} fn(\mathbf{0}) &=& \emptyset \\ fn(P \mid Q) &=& fn(P)\cup fn(Q) \\ fn(!P) &=& fn(P) \\ fn(x[P]) &=& fn(P)\cup\{x\} \\ fn((\boldsymbol{\nu}\, x{:}M)P) &=& (fn(P)\setminus\{x\})\cup fn(M) \\ fn(x^\eta(y{:}M).P) &=& (fn(P)\setminus\{y\})\cup fn(M)\cup\{x\}\cup fn(\eta) \\ fn(\overline{x}^\eta(y).P) &=& fn(P)\cup\{y\}\cup\{x\}\cup fn(\eta) \\ fn(\overline{x}^\eta\{\!\!\{y\}\!\!\}.P) &=& fn(P)\cup\{y\}\cup\{x\}\cup fn(\eta) \\ fn(x^\eta\{\!\!\{y\}\!\!\}.P) &=& fn(P)\cup\{y\}\cup\{x\}\cup fn(\eta) \\ fn(\uparrow) &=& \emptyset \\ fn(*) &=& \emptyset \\ fn(x) &=& \{x\} \end{array}$$

May be useful to notice that the scoping of the name x introduce by the two binders $(\boldsymbol{\nu}\, x{:}M).P$ and $y^\eta(x{:}M).P$ is the process P. So x can not appear in $fn(M)$.

As in the untyped case, we write $P\{{}^{y}\!/\!_{x}\}$ to denote the process in which every free occurrence of x has been replaced by y, renaming as necessary to avoid capture.

Semantics

The rules that define the structural congruence relation are reported in Figure 4.1. They are as those of the untyped calculus with the exception of the *(Struct Res Res)* rule.

Since our type grammar admits names in types, the simple operation of swapping two restrictions gets potentially dangerous. For instance, the process

$$(\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})\ (\boldsymbol{\nu}\, y{:}\mathtt{Id}\ [x{:}\mathtt{Shh}])\ \mathbf{0}$$

is well formed, while

$$(\boldsymbol{\nu}\, y{:}\mathtt{Id}\ [x{:}\mathtt{Shh}])\ (\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})\ \mathbf{0}$$

is not since x is free in `Id` $[x{:}\mathtt{Shh}]$.

Adding to the original rule the condition $x \notin \mathit{fn}(N') \wedge y \notin \mathit{fn}(N)$ suffices to avoid this. The *(Struct Res Res)* rule takes the following form:

$$(\boldsymbol{\nu}\, x{:}M)(\boldsymbol{\nu}\, y{:}M')P \equiv (\boldsymbol{\nu}\, y{:}M')(\boldsymbol{\nu}\, x{:}M)P \qquad x \notin \mathit{fn}(M') \wedge y \notin \mathit{fn}(M)$$

$P \equiv P$		(Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$		(Struct Comm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$		(Struct Trans)
$P \mid \mathbf{0} \equiv P$		(Struct Dead Par)
$P \mid Q \equiv Q \mid P$		(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$		(Struct Par Assoc)
$!P \equiv !P \mid P$		(Struct Repl Par)
$(\boldsymbol{\nu}\, x{:}M)\mathbf{0} \equiv \mathbf{0}$		(Struct Res Dead)
$(\boldsymbol{\nu}\, x{:}M)P \equiv P$	for $x \notin \mathit{fn}(P)$	(Struct Res Garb)
$(\boldsymbol{\nu}\, x{:}M)(\boldsymbol{\nu}\, y{:}M')P \equiv (\boldsymbol{\nu}\, y{:}M')(\boldsymbol{\nu}\, x{:}M)P$	$x \notin \mathit{fn}(M'), y \notin \mathit{fn}(M)$	(Struct Res Res)
$(\boldsymbol{\nu}\, x{:}M)(P \mid Q) \equiv P \mid (\boldsymbol{\nu}\, x{:}M)Q$	for $x \notin \mathit{fn}(P)$	(Struct Res Par)
$(\boldsymbol{\nu}\, x{:}M)y[P] \equiv y[(\boldsymbol{\nu}\, x{:}M)P]$	for $x \neq y$	(Struct Res Agent)
$P \equiv Q \Rightarrow (\boldsymbol{\nu}\, x{:}M)P \equiv (\boldsymbol{\nu}\, x{:}M)Q$		(Struct Cong New)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$		(Struct Cong Par)
$P \equiv Q \Rightarrow x[P] \equiv x[Q]$		(Struct Cong Agent)
$P \equiv Q \Rightarrow \alpha . P \equiv \alpha . Q$		(Struct Cong Action)

Figure 4.1: Structural congruence $P \equiv Q$

The reduction rules are the same of the untyped calculus and are reported in Figure 4.2.

$x^*(u{:}M).P \mid \overline{x}^*(v).Q$	$\rightarrow$	$P\{^v/_u\} \mid Q$	(write local)
$x^y(u{:}M).P \mid y[\overline{x}^\uparrow(v).Q \mid R]$	$\rightarrow$	$P\{^v/_u\} \mid y[Q \mid R]$	(write out)
$\overline{x}^y(v).P \mid y[x^\uparrow(u{:}M).Q \mid R]$	$\rightarrow$	$P \mid y[Q\{^v/_u\} \mid R]$	(write in)
$x^*\lbrace u\rbrace.P \mid \overline{x}^*\lbrace v\rbrace.Q \mid v[B]$	$\rightarrow$	$P \mid u[B] \mid Q$	(move local)
$x^y\lbrace u\rbrace.P \mid y[\overline{x}^\uparrow\lbrace v\rbrace.Q \mid v[R] \mid S]$	$\rightarrow$	$P \mid u[R] \mid y[Q \mid S]$	(move out)
$\overline{x}^y\lbrace v\rbrace.P \mid v[R] \mid y[x^\uparrow\lbrace u\rbrace.Q \mid S]$	$\rightarrow$	$P \mid y[Q \mid S \mid u[R]]$	(move in)
$P \rightarrow Q$	$\Rightarrow$	$P \mid R \rightarrow Q \mid R$	(red par)
$P \rightarrow Q$	$\Rightarrow$	$(\boldsymbol{\nu}\, x{:}M)P \rightarrow (\boldsymbol{\nu}\, x{:}M)Q$	(red res)
$P \rightarrow Q$	$\Rightarrow$	$u[P] \rightarrow u[Q]$	(red agent)
$P \equiv P' \wedge P' \rightarrow Q' \wedge Q' \equiv Q$	$\Rightarrow$	$P \rightarrow Q$	(red $\equiv$)

Figure 4.2: Reduction rules

4.3 The Type System

Each judgment has the structure $\Gamma \vdash_{\Xi} \Im$. The form of the assertion $\Im$ depends on the judgment; the couple Γ, Ξ will be referred to as *typing environment*.

Γ consists of a list of assumptions for names, each of the form *name* : *type*. Its syntax is reported below.

$$\begin{aligned} \Gamma ::= {} & \varnothing \\ \mid {} & \Gamma, x{:}M \end{aligned}$$

At the same time, we need some machinery to enforce the restriction we imposed on the use of channel names. A name which is bound in an input operation may change during reductions, and to have a subject reduction result it cannot appear in the interface of a process. Since only names used to perform input operations appear in the interface, the type system must assure that names *not* bound by an input action (equiv. names introduced by a $\boldsymbol{\nu}$ or free) are used to perform input operations.

Thus in a second environment, noted Ξ, we record the set of names not bound by an input action. The syntax of Ξ is reported below.

$$\begin{aligned} \Xi ::= {} & \varnothing \\ \mid {} & \Xi, x \end{aligned}$$

There are four typing judgments: the first constructs well-formed environments, the second constructs well-formed types, the third tracks the type of names, and the fourth tracks the interface of the processes.

$\Gamma \vdash_{\Xi} \diamond$	well-formed environment Γ, Ξ,
$\Gamma \vdash_{\Xi} T$	type T is well-formed under Γ, Ξ,
$\Gamma \vdash_{\Xi} x : M$	name x has type M under Γ, Ξ,
$\Gamma \vdash_{\Xi} P : \texttt{Proc}\ A$	process P has type $\texttt{Proc}\ A$ under Γ, Ξ.

NOTATION: As useful abbreviations, we often describe Ξ in two components, Ξ', Ξ'', with the following meanings:

$\Gamma', x{:}M, \Gamma'' \vdash_{\Xi',\Xi''} \Im$ In Ξ' are registered names defined in Γ', in Ξ'' names defined in Γ'', and x does not appear in Ξ.

$\Gamma', x{:}M, \Gamma'' \vdash_{\Xi',x,\Xi''} \Im$ In Ξ' are registered names defined in Γ', in Ξ'' names defined in Γ'', and x appears in Ξ.

$\Gamma', x{:}M, \Gamma'' \vdash_{\Xi',(x),\Xi''} \Im$ In Ξ' are registered names defined in Γ', in Ξ'' names defined in Γ'', and no hypothesis if x is or is not n Ξ can be made.

The component Ξ'' can be empty.

4.3.1 Type rules

Well formed environments

(Env Empty)

$$\frac{}{\varnothing \vdash_{\varnothing} \diamond}$$

(Env Add Seal)

$$\frac{\Gamma \vdash_{\Xi} A}{\Gamma, x{:}\mathtt{Id}\ A \vdash_{\Xi} \diamond} \quad x \notin \mathrm{dom}(\Gamma)$$

(Env Add Channel)

$$\frac{\Gamma \vdash_{\Xi} V}{\Gamma, x{:}\mathtt{Ch}\ V \vdash_{\Xi,(x)} \diamond} \quad x \notin \mathrm{dom}(\Gamma)$$

The empty environment is noted $\varnothing, \varnothing$, and is well-formed. It is possible to add seal and channel names with their types: the only condition is that the name must not be already defined in the environment. In case a channel name is being added, it may be inserted also in Ξ — as in case of a name introduced by $\boldsymbol{\nu}$ or free —, or it may not — as in the case of a name bound in an input operation.

Notice that there is a relationship between Ξ and Γ: $\mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\Gamma)$.

The following standard rule allows us to extract the type of a variable from an environment:

(Identity)

$$\frac{\Gamma \vdash_{\Xi} \diamond}{\Gamma \vdash_{\Xi} x : \Gamma(x)} \quad x \in \mathrm{dom}(\Gamma)$$

Well formed types

(Type Shh)

$$\frac{\Gamma \vdash_{\Xi} \diamond}{\Gamma \vdash_{\Xi} \mathtt{Shh}}$$

(Type Id)

$$\frac{\Gamma \vdash_{\Xi} A}{\Gamma \vdash_{\Xi} \mathtt{Id}\ A}$$

(Type Ch)

$$\frac{\Gamma \vdash_{\Xi} V}{\Gamma \vdash_{\Xi} \mathtt{Ch}\ V}$$

(Type Proc)

$$\frac{\Gamma \vdash_{\Xi} A}{\Gamma \vdash_{\Xi} \mathtt{Proc}\ A}$$

(Type Interface)

$$\frac{\Gamma \vdash_{\Xi} x_i : \mathtt{Ch}\ V_i \qquad x_i \in \mathrm{dom}(\Xi)}{\Gamma \vdash_{\Xi} [x_1{:}V_1, \ldots, x_n{:}V_n]} \quad \forall i \in 1..n$$

The $\mathtt{Shh}$ type is always well formed in a well formed environment. The types $\mathtt{Id}\ A$, $\mathtt{Ch}\ V$, and $\mathtt{Proc}\ A$ are well formed if respectively A, V, and A are well formed. An interface type is well formed if each name in it has been previously declared with the correct type. Each name is either free or bound by the $\boldsymbol{\nu}$ operator, and so it must appear in Ξ.

Processes

(Proc Dead)

$$\frac{\Gamma \vdash_{\Xi} \diamond}{\Gamma \vdash_{\Xi} \mathbf{0} : \mathtt{Proc}\ []}$$

(Proc Par)

$$\frac{\Gamma \vdash_{\Xi} P_1 : \mathtt{Proc}\ A \qquad \Gamma \vdash_{\Xi} P_2 : \mathtt{Proc}\ A}{\Gamma \vdash_{\Xi} P_1 \mid P_2 : \mathtt{Proc}\ A}$$

(Proc Res Agent)

$$\frac{\Gamma, x{:}\mathtt{Id}\ A \vdash_{\Xi} P : \mathtt{Proc}\ A'}{\Gamma \vdash_{\Xi} (\boldsymbol{\nu}\, x{:}\mathtt{Id}\ A)P : \mathtt{Proc}\ A'}$$

(Proc Res Ch)

$$\frac{\Gamma, x{:}\mathtt{Ch}\ V \vdash_{\Xi,x} P : \mathtt{Proc}\ A}{\Gamma \vdash_{\Xi} (\boldsymbol{\nu}\, x{:}\mathtt{Ch}\ V)P : \mathtt{Proc}\ A} \quad x \notin fn(A)$$

(Proc Seal)
$$\frac{\Gamma \vdash_{\Xi} x : \texttt{Id}\ A \qquad \Gamma \vdash_{\Xi} P : \texttt{Proc}\ A}{\Gamma \vdash_{\Xi} x[P] : \texttt{Proc}\ [\,]}$$

(Proc Bang)
$$\frac{\Gamma \vdash_{\Xi} P : \texttt{Proc}\ A}{\Gamma \vdash_{\Xi} !P : \texttt{Proc}\ A}$$

The dead process $\mathbf{0}$ does not perform any action, so in a well-formed environment its interface is trivially $\texttt{Proc}\ [\,]$, as evident from the *(Proc Dead)* rule. This fairly simple rule has an important consequence: the type checking of a term will in the end apply the empty-process rule, testing that the typing environment constructed typing the term is well formed.

The static scoping of names implies that typing a process in which a name has been restricted is equivalent to typing the process in an environment where the restricted name has been added.

A particular attention has to be paid in the case of the restriction of a channel name, as the channel name might appear in the interface. A possible approach imposes the erasing of the newly restricted name from the interface, with a rule of the form:

(Naïve Res Ch)
$$\frac{\Gamma, x{:}\texttt{Ch}\ V \vdash_{\Xi,x} P : \texttt{Proc}\ A}{\Gamma \vdash_{\Xi} (\boldsymbol{\nu}\, x{:}\texttt{Ch}\ V)P : \texttt{Proc}\ (A - [x{:}V])}$$

This rule is not sound as regards to the structural congruence relation. Consider the processes
$$P = (\boldsymbol{\nu}\, y{:}[\,])y[(\boldsymbol{\nu}\, x{:}\texttt{Ch Shh})x^{\uparrow}()]$$
$$Q = (\boldsymbol{\nu}\, y{:}[\,])(\boldsymbol{\nu}\, x{:}\texttt{Ch Shh})y[x^{\uparrow}()]$$
By *(Struct Res Agent)* they are equivalent, but while the former is well typed, the latter is not.

We adopt the safe rule which imposes that a restricted name can not appear in the interface of the process. As all the names must have been firstly introduced by a restriction, it may look that this condition forces all the interfaces to be empty. But if we consider a seal as a process, it looks like a dead process: it does not perform any input from $\uparrow$ at the current level. Thus its type is $\texttt{Proc}\ [\,]$. At the same time we want to check that the process inside the seal respects the interface declared for the Seal's name and this is achieved by imposing that the interface of the process inside the seal is the same as the interface of the name. In other words, a channel appearing in an interface must be already known to the enclosing environment. This is a reasonable feature of the type system: the interface's names must be somewhat public. A brief example to clarify what "somewhat" means. Consider the following two terms in the light of the *(Proc Res Ch)* rule:

$$y[(\boldsymbol{\nu}\, x{:}\texttt{Ch Shh})\ x^{\uparrow}()]$$
$$(\boldsymbol{\nu}\, x{:}\texttt{Ch Shh})y[x^{\uparrow}()]$$

Clearly the first is not well typed, since the process inside the seal should offer in the interface a channel which is restricted, and this is forbidden by the *(Proc Res Ch)* rule. Interestingly, also the latter is not well-typed: the type of the name y should include the channel x in the interface, and there are no contextes which may define the name y. The correct term is the following:
$$(\boldsymbol{\nu}\, x{:}\texttt{Ch Shh})(\boldsymbol{\nu}\, y{:}\texttt{Id}\ [x{:}\texttt{Shh}])\ y[x^{\uparrow}()]$$

in which the channel x is declared before the seal y. Briefly, a name that is used by a seal to read from its environment must already exist in the environment when the seal is declared.

The rule for the parallel composition *(Proc Par)* makes sure that both subprocesses have the same interface: we will say more about this in the section about subtyping.

The *(Proc Bang)* rule says that when a process is well-typed and has an interface, then the replication of the process will always have the same interface.

Communication actions

All the rules for typing a process of the form $\alpha.P$ follow a common pattern: this is especially true if we consider input and output rules separately.

(Input Local)
$$\frac{\Gamma \vdash_\Xi x : \texttt{Ch}\ M \qquad \Gamma, y{:}M \vdash_\Xi P : \texttt{Proc}\ A}{\Gamma \vdash_\Xi x^*(y{:}M).P : \texttt{Proc}\ A} \quad x \in \mathrm{dom}(\Xi)$$

(Input Up)
$$\frac{\Gamma \vdash_\Xi x : \texttt{Ch}\ M \qquad \Gamma, y{:}M \vdash_\Xi P : \texttt{Proc}\ A}{\Gamma \vdash_\Xi x^\uparrow(y{:}M).P : \texttt{Proc}\ (A \oplus [x{:}M])} \quad x \in \mathrm{dom}(\Xi)$$

(Input Down)
$$\frac{\Gamma \vdash_\Xi z : \texttt{Id}\ A' \qquad \Gamma \vdash_\Xi x : \texttt{Ch}\ M \qquad \Gamma, y{:}M \vdash_\Xi P : \texttt{Proc}\ A}{\Gamma \vdash_\Xi x^z(y{:}M).P : \texttt{Proc}\ A} \quad x \in \mathrm{dom}(\Xi)$$

The name y in a $x^\eta(y{:}M)$ is not free, so it must be added to the set of names which are in the scope of P. So we check that x denotes a channel of the good type to transmit a value of type M, and we analyze P after having added y to the set of names known by P. We notice that y is not added to Ξ since it is bound in an input operation. Because we are doing an input, we also have to check that the subject name has not been previously used as object name in another input operation: this is done by imposing the condition $x \in \mathrm{dom}(\Xi)$.

In *(Input Local)* the input operation is local and nothing more has to be done.

In *(Input Down)* the input is from a child seal and we control also that the name of the seal we are going to read is really a seal name.

In *(Input Up)* the input is from $\uparrow$: the channel the process is going to read on must be added to the interface already deduced for P: this is done by the $\oplus$ operator.

(Output Local)
$$\frac{\Gamma \vdash_\Xi x : \texttt{Ch}\ M \qquad \Gamma \vdash_\Xi y : M \qquad \Gamma \vdash_\Xi P : \texttt{Proc}\ A}{\Gamma \vdash_\Xi \overline{x}^*(y).P : \texttt{Proc}\ A}$$

(Output Up)
$$\frac{\Gamma \vdash_\Xi x : \texttt{Ch}\ M \qquad \Gamma \vdash_\Xi y : M \qquad \Gamma \vdash_\Xi P : \texttt{Proc}\ A}{\Gamma \vdash_\Xi \overline{x}^\uparrow(y).P : \texttt{Proc}\ A}$$

(Output Down)
$$\frac{\Gamma \vdash_\Xi z : \texttt{Id}\ A' \qquad \Gamma \vdash y : M \qquad \Gamma \vdash_\Xi P : \texttt{Proc}\ A}{\Gamma \vdash_\Xi \overline{x}^z(y).P : \texttt{Proc}\ A} \quad (x{:}M) \in A'$$

In the case of local and up output actions the type rules *(Output Local)* and *(Output Up)* check that the subject name is the name of a channel on which messages of the type of the object name can be transmitted over. Also, the continuation P must be well typed.

This is enough to check the correctness of communication, but something more is required in order to achieve the attended functionality from the interfaces of seals. So typing a write down action, follows a slightly different strategy, used in rule *(Output Down)*. First of all, we get the interface of the seal we are going to write in, and the type of the object name. Then we check if the subject name figures in the interface with the right type. This enforces the interpretation from the interface of a seal: a process can write inside a seal only if the processes local to the seal are going to read it. As usual, the continuation must be well typed.

Mobility actions

(Rcv Local)

$$\frac{\Gamma \vdash_\Xi x : \texttt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \texttt{Id}\ A \qquad \Gamma \vdash_\Xi P : \texttt{Proc}\ A'}{\Gamma \vdash_\Xi x^*\{\!\{ y \}\!\}.P : \texttt{Proc}\ A'} \qquad x \in \mathrm{dom}(\Xi)$$

(Snd Local)

$$\frac{\Gamma \vdash_\Xi x : \texttt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \texttt{Id}\ A \qquad \Gamma \vdash_\Xi P : \texttt{Proc}\ A'}{\Gamma \vdash_\Xi \overline{x}^*\{\!\{ y \}\!\}.P : \texttt{Proc}\ A'}$$

(Rcv Up)

$$\frac{\Gamma \vdash_\Xi x : \texttt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \texttt{Id}\ A \qquad \Gamma \vdash_\Xi P : \texttt{Proc}\ A'}{\Gamma \vdash_\Xi x^\uparrow\{\!\{ y \}\!\}.P : \texttt{Proc}\ (A' \cup [x{:}A])} \qquad x \in \mathrm{dom}(\Xi)$$

(Snd Up)

$$\frac{\Gamma \vdash_\Xi x : \texttt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \texttt{Id}\ A \qquad \Gamma \vdash_\Xi P : \texttt{Proc}\ A'}{\Gamma \vdash_\Xi \overline{x}^\uparrow\{\!\{ y \}\!\}.P : \texttt{Proc}\ A'}$$

(Rcv Down)

$$\frac{\Gamma \vdash_\Xi z : \texttt{Id}\ A'' \qquad \Gamma \vdash_\Xi x : \texttt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \texttt{Id}\ A \qquad \Gamma \vdash_\Xi P : \texttt{Proc}\ A'}{\Gamma \vdash_\Xi x^z\{\!\{ y \}\!\}.P : \texttt{Proc}\ A'} \qquad x \in \mathrm{dom}(\Xi)$$

(Snd Down)

$$\frac{\Gamma \vdash_\Xi z : \texttt{Id}\ A_1 \qquad \Gamma \vdash_\Xi y : \texttt{Id}\ A \qquad \Gamma \vdash_\Xi P : \texttt{Proc}\ A_2}{\Gamma \vdash_\Xi \overline{x}^z\{\!\{ y \}\!\}.P : \texttt{Proc}\ A_1} \qquad (x{:}A) \in A_1$$

The typing rules for mobility actions do not differ from the respective communication actions. The main point is that in a receive operation the object name is not bound, so it is not added to the name in the scope of the continuation. Again, because we are transmitting agents and not names, nothing is modified in Ξ.

Subtyping relation

During reductions, actions can get consumed. Consider the process

$$P \;=\; x^{\uparrow}(y{:}M).\overline{z}^{*}(y)$$

It is ready to input a name of type M on channel x and its type is $\texttt{Proc}\ [x{:}M]$. Now place it in the context

$$C[-] \;=\; \overline{x}^{a}(w) \mid a[-]$$

and consider the type of P and of its reduct:

$$\begin{array}{ccc} \overline{x}^{a}(w).Q \mid a[x^{\uparrow}(y{:}M).\overline{z}^{*}(y)] & \rightarrow & Q \mid a[\overline{z}^{*}(w)] \\ P : \texttt{Proc}\ [x{:}M] & & P : \texttt{Proc}\ [\,] \end{array}$$

To achieve a subject reduction result, we introduce a subtyping relation. We already discussed that the interface of a process should be regarded as the set of channels on which the process *may* perform input operations from $\uparrow$. This suggests that adding new channels in the interface of a process should not be considered as an error, since they are channels on which interaction will never take place. This gives rise to a subtyping notion on the $\texttt{Proc}$ type, showed in *(*$\texttt{Proc}$ *sub)* rule.

(Proc sub)
$$\frac{A \subseteq A'}{\texttt{Proc}\ A \leq \texttt{Proc}\ A'}$$

The symbol $\subseteq$ denotes the usual inclusion relation between sets, and thus let add channels in the interface of a process. We have as usual the basic rules which characterize every subtyping relation: reflexivity and transitivity.

(Sub Refl)
$$T \leq T$$

(Sub Trans)
$$\frac{T_1 \leq T_2 \qquad T_2 \leq T_3}{T_1 \leq T_3}$$

At last, the subsumption rule throws a bridge between the subtyping relation and the type system, allowing a term m of type T to be used in all contexts where a term of type T' with $T \leq T'$ is attended.

(subs)
$$\frac{\Gamma \vdash_{\Xi} m : T \qquad \Gamma \vdash_{\Xi} T' \qquad T \leq T'}{\Gamma \vdash_{\Xi} M : T'}$$

The subtyping notion, as possibility of extending the interface, is limited to the process types and is not extended to the seal interfaces, appearing an exquisitely dynamic feature. The interface of a seal, after having been declared, is immutable and characterizes forever the range of interactions admitted (rember, seals are eternal). At the same time, a seal is the most natural context for a process. The type rule of the seal construct requires the interface declared for the seal and the interface of the inside process to coincide. Using the subsumption, also a process with a more limited interface can now be placed inside the seal: the more limited interface may be caused, as in the previous example, by the consumption of some actions.

In this way, actions can get consumed inside a seal, while the seal preserves its crystallized interface.

4.3.2 Typing algorithm

The type rules in the previous section just need some "hacks" to be converted in a type algorithm. As usual in type system with subtyping, we must deal with subsumption in a deterministic way. Actually there are only two rules that need subsumption.

The first is the *(Proc Par)* rule.

(Proc Par Algo)
$$\frac{\Gamma \rhd_{\Xi} P_1 : \texttt{Proc}\ A_1 \qquad \Gamma \rhd_{\Xi} P_2 : \texttt{Proc}\ A_2}{\Gamma \rhd_{\Xi} P_1 \mid P_2 : \texttt{Proc}\ A_1 \oplus A_2}$$

In order to typecheck $P_1 \mid P_2$ in the environment Γ, Ξ both P_1 and P_2 are checked resulting respectively in the two types $\texttt{Proc}\ A_1$ and $\texttt{Proc}\ A_2$. If the process $P_1 \mid P_2$ can perform an input at $\uparrow$ then P_1 or P_2 must be able to perform it, and so it has been registered in one of $\texttt{Proc}\ A_1$ and $\texttt{Proc}\ A_2$. Thus we have to merge the type informations kept in A_1 and A_2, and this is achieved by means of the $\oplus$ operator.

The second rule we need to modify is the *(Proc Seal)* rule. We have to take into account that the interface of the process inside the seal is compatible with the declared interface also if it is a subtype of the attended process type. This can be imposed directly in the typing rule, rather than deriving it by using the subsumption rule.

(Seal Algo)
$$\frac{\Gamma \rhd_{\Xi} x : \texttt{Id}\ A \qquad \Gamma \rhd_{\Xi} P : \texttt{Proc}\ A'}{\Gamma \rhd_{\Xi} x[P] : \texttt{Proc}\ [\,]} \qquad \texttt{Proc}\ A' \leq \texttt{Proc}\ A$$

As consequence of the admissibility of the rules *(Sub Refl)* and *(Sub Trans)*, the subtyping relation $\leq$ is decidable. This completes the description of the typing algorithm.

4.4 Results

There is an immediate relationship between the type system and the type algorithm, expressed by the two following lemmas.

Lemma 1 *If* $\Gamma \rhd_{\Xi} P : \texttt{Proc}\ A$ *then* $\Gamma \vdash_{\Xi} P : \texttt{Proc}\ A$.

Lemma 2 *If* $\Gamma \vdash_{\Xi} P : \texttt{Proc}\ A$ *then* $\exists A'$ *such that* $\texttt{Proc}\ A' \leq \texttt{Proc}\ A$ *and* $\Gamma \rhd_{\Xi} P : \texttt{Proc}\ A'$.

The first asserts that the type algorithm is correct, that the inferred type is a type allowed by the type system. The second asserts that the type algorithm infers the minimum type that the type system assigns to a given term. These two lemmas allow us to reason directly on the algorithmic derivation which has the property of being unique. Another useful property is the minimality of the type inferred by the typing algorithm, as expressed by the following quite standard corollary:

Corollary 1 $\Gamma \rhd_{\Xi} P : \min\{T \mid \Gamma \vdash_{\Xi} P : T\}$

In order to prove the subject reduction we need a substitution lemma that asserts that substituting names in a term under some conditions results in a well-typed term. In particular, we need that the name we are going to substitute has the same type of the substituted one; this is not enough, since channel names appear in the types. So we ask also that in the case of a channel name, the substituted name is never used in an input operation: this is obtained imposing the condition that the substituted name does not appear in Ξ. This implies that it will never be used to perform an input operation and thus it will never appear in any type.

Proposition 1 (Substitution Lemma) *If* $\Gamma, x{:}M \rhd_{\Xi} P : \texttt{Proc}\ A$, $x \notin dom(\Xi)$, *and* $\Gamma \rhd_{\Xi} y : M$ *then* $\Gamma \rhd_{\Xi} P\{{}^{y}/_{x}\} : \texttt{Proc}\ A$.

The correctness of the type system is then proved, with a subject reduction theorem. It can be interpreted as stating that each interaction is well typed, and that all communication through seal's boundary respect the interface declared for the seal.

Proposition 2 (Subject Reduction) *If* $\Gamma \rhd_{\Xi} P : \texttt{Proc}\ A$ *and* $P \rightarrow Q$ *then* $\Gamma \rhd_{\Xi} Q : \texttt{Proc}\ A$.

A particularity of the type system is that the type of a term does not decrease during reductions, as customary when working with the algorithmic derivation in a type system which allows subtyping. The actions which influence the interface are only the input operations from $\uparrow$: at top-level, interactions from $\uparrow$ will never be fired. In order to interaction to occur, actions must be localized inside seals. The interfaces of the seals crystallize the interfaces of the processes, while the subtyping rule allows actions to get consumed.

5

The expressivity of Seal

5.1 Seal-1 vs. Seal-2

The Seal Calculus was defined by Castagna and Vitek in close contact with the Object System Group of the University of Geneva which later developed the *JavaSeal Mobile Agent Kernel*. In order to implement a system for the distribution of newspaper articles on the web, the Object System Goup found the Java platform not enough expressive and the Java Security Architecture not able to guarantee the security properties requested by the application. This motivated the work on a mobile agent framework. JavaSeal is a secure kernel for mobile environments and mobile objects implemented as a collection of trusted packages on top of the Sun's JDK1.2 virtual machine. It provides a minimal interface for writing mobile agent system: the original Seal Calculus was the formal definition of the JavaSeal kernel, allowing abstract reasoning on the behaviour of systems.

The calculus presented here in this work as Seal Calculus differs in many aspects from the original presentation: we will point out the differences explaining their origins and compare the expressiveness of the two calculi. To avoid confusion, in this section we will refer the original calculus as Seal-1 and the calculus described and used all along this work as Seal-2.

There are two main differences between the designs of the two calculi, the pattern of remote interaction and the possibility of making copies of a seal.

Resources In Seal-1 all channels are local to a seal, and are considered as local resources: shared channels (and so shared resources) do not exist. At the same time, a process is able to interact on channels which belong to its enclosing seal, as well as on channels which belong to the parent seal, or to one of children seals. The location labels have the following meanings:

$$
\begin{array}{rcll}
\eta & ::= & * & \text{channel to be found in the current seal} \\
 & \mid & \uparrow & \text{channel to be found in the parent seal} \\
 & \mid & z & \text{channel to be found in the child seal named } z
\end{array}
$$

Two processes local to the same seal can interact on channels labelled $*$: this means that both processes will find the requested resource at the current level. Two processes local to seals in parent-child relationship can interact on channels local to the parent seal as well as local to the child seal. For instance, in the configuration $P \mid n[Q]$ if the two processes P and Q are willing to interact (remotely) on the channel x they will refer to it according the following table:

x belongs to	P	Q
the parent seal	x^*	$x^\uparrow$
the child seal	x^n	x^*

This form of interaction characterizes a local action as an "open" offer, that is, it does not prescribe which seal should provide a matching offer (it can also be matched by another local action). On the other hand, a remote action is a "closed" offer in the sense that the target seal is named explicitely.

This pattern of interaction is extremely flexible and appealing, but the nature of the local actions does not provide a suitable notion of interface, that is, a way to characterize the possible interactions of a seal with its environment.

Besides, in order to dynamically grant access rights to the local resources, Seal-1 presents a complex mechanism of *portals*. A process can not access a remote resource if a process local to the remote seal does not open a portal on the local resource. This is a necessary security mechanism in Seal-1, but appears useless in Seal-2, where the notion of local resource has been replaced by the notion of shared one. Portals have been eliminated in Seal-2.

Duplicating seals In Seal-1, the operation of receiving a seal is extended with the possibility of making copies. Thus, the syntax of the mobility actions is:

$$\begin{array}{lll} \alpha ::= \dots & & \text{communication actions} \\ \quad \mid \ \overline{x}^\eta \lbrace\!\vert y \vert\!\rbrace & & \text{send} \\ \quad \mid \ x^\eta \lbrace\!\vert y_1, \cdots, y_n \vert\!\rbrace & & \text{receive} \end{array}$$

with the informal semantics (we describe only the interaction on a local channel: the extension is straightforward):

$$y[P] \mid \overline{x}^* \lbrace\!\vert y \vert\!\rbrace.Q \mid x^* \lbrace\!\vert y_1, \cdots, y_n \vert\!\rbrace.R \rightarrow y_1[P] \mid \cdots \mid y_n[P] \mid Q \mid R$$

In the case of $n = 0$ the transmitted seal is dissolved, like it was written in a distributed version of the `/dev/null` sink. This has an interesting consequence in the semantics of the calculus: the structural equivalence rule

$$x[(\boldsymbol{\nu}\, y)P] \equiv (\boldsymbol{\nu}\, y)x[P] \quad \text{for } x \neq y$$

is no longer true. We recall that the structural equivalence rules defines the minimal behavior equivalence relation: in presence of the possibility of copying seal the two side of the rule are not equivalent. For instance, called Q the process $(\boldsymbol{\nu}\, z)(\overline{z}^* \lbrace\!\vert m \vert\!\rbrace \mid z^* \lbrace\!\vert m, n \vert\!\rbrace)$ which copies the seal m in the seal n, consider

$$\begin{array}{lcl} m[(\boldsymbol{\nu}\, y)P] \mid Q & \rightarrow & m[(\boldsymbol{\nu}\, y)P] \mid n[(\boldsymbol{\nu}\, y)P] \\ (\boldsymbol{\nu}\, y)m[P] \mid Q & \rightarrow & (\boldsymbol{\nu}\, y)(m[P] \mid n[P]) \end{array}$$

The first term yelds a configuration where seals n and m have each a private channel y, while in the other case they share a common channel y. This problem derives directly by the interaction between scope extrusion and `copy` operation: thus, it is not restricted to the Seal Calculus but it appears in all calculi which admits the two operations.

In order to avoid unnecessary semantics complications to the already clumsy syntax of the calculus, we do not allow duplicating seals in Seal-2.

5.1.1 Restrictive patterns

The pattern of interaction of Seal-1 allows many systems to be programmed with fairly modest efforts. For example, communicating a message to all children seals (as many as they are, including some that may not be present in the initial configuration) can be encoded as

$$!\overline{b}^{*}(msg) \mid y_1[b^{\uparrow}(m).P_1] \mid \cdots \mid y_n[b^{\uparrow}(m).P_n]$$

The absence of the open offer implies that in Seal-2 this protocol cannot be encoded, at least not in a direct way. Since all communications with a child seal must specify the name of the child seal, a way to implement a down-broadcast consists in keeping track of all the names of the children seals, dynamically bring up-to-date the list of the children seals according to their movements, repeatedly scan the list and send messages. This can be done explicitely, but it is really complex.

In order to avoid clumsy encodings[1], we introduce a notion of open offer compatible with our notion of interface. We add a new location, $\downarrow$, which represents the set of children seal.

$$\begin{array}{lll} \mathit{loc} ::= & \ldots & \text{as before} \\ & \mid \ \downarrow & \text{children} \end{array}$$

The only operations we allow from $\downarrow$ is the input of a value and the receive of a seal, according the following reduction rules:

$$\begin{array}{rcll} x^{\downarrow}(u).P \mid y[\overline{x}^{\uparrow}(v).Q \mid R] & \rightarrow & P\{^{v}/_{u}\} \mid y[Q \mid R] & \text{(read children)} \\ x^{\downarrow}\{u\}.P \mid y[\overline{x}^{\uparrow}\{v\}.Q \mid v[R] \mid S] & \rightarrow & P \mid u[R] \mid y[Q \mid S] & \text{(receive children)} \end{array}$$

What characterize these rules is that the reduction is independent from the name y of the child seal. We add the following typing rule to deal with $\downarrow$ (the extension of the subject-reduction proof is straightforward):

(Loc $\downarrow$)

$$\frac{}{\Gamma \vdash_{\Xi} \downarrow : \mathtt{Id}\ [\,]}$$

This allows considering the new location as a (distinguished) name which denotes a seal with an empty interface: trying to write something inside $\downarrow$ will raise a type error, while reading from $\downarrow$ is allowed.

[1] We are actually investigating if the extension we propose is really an extension or just notational convenience.

The set of children of each seal can change following the evolution of the system: mobility operations can allow to enter other seals, which offer extremely different interfaces, in a given seal. We are not interested in trying to characterize this set: also at the type level it would require some form of dynamic typing which is beyond the horizon of this work. Thus this notion of open offer appears extremely simple and sufficiently expressive. For instance, we can encoded the broadcast-down protocol in a simple way:

$$!x^{\downarrow}(y).\overline{b}^{y}(msg) \mid y_1[\overline{x}^{\uparrow}(y_1).b^{\uparrow}(m).P_1] \mid \cdots \mid y_n[\overline{x}^{\uparrow}(y_n).b^{\uparrow}(m).P_n]$$

Every seal that desires to known the broadcasted message communicates its name to the broadcast "station". The type of channel x along which the broadcaster receives the name of the children seal imposes that all seal receiving the broadcasted messages must have the same interface: this restriction appears reasonable, since one can imagine that all agents willing to receive the broacasted message have similar functions, and the broadcast service can be easily extended with different broadcasters dealing with different services.

5.2 Encoding a Distributed Language

In order to test the expressivity of Seal-2, we consider a fragment of a typed, distributed language where mobile threads can migrate between immobile network nodes. This toy language has been introduced in [11], and its semantics has been given via a translation into the Ambient Calculus. We will stick to the original syntax and semantics (except for polyadic communication replaced by monadic communication), as well as to the original presentation.

The computational model is that there is an unstructured collection of named network nodes, each of which hosts a collection of named communication channels and anonymous threads. Communication is based on named communication channels: each channel is local to one node, and only threads currently inside that node can access it. Messages on communication channels are assigned types, ranged over by W. The type *Node* is the type of names of network nodes. The type $Ch(W)$ is the type of a monadic communication channel in which the message communicated have type W.

Next, we describe the formal grammar of the Toy language. A *network, Net*, ranged over by n, is a collection of nodes, built up using composition and restrictions. A *crowd, Cro*, is the group of threads and channels hosted by a node. A *thread, Th*, is a mobile thread of control. The actions a thread can perform include migration to a remote node, communication on named channel (ranged over by c), forking a new threads in the current node, and spawning new nodes at the network level.

W	$::= Node$	type of node name
	$\mid\ Ch(W)$	type of channel name
Net	$::= (\boldsymbol{\nu}\, x{:}W)Net$	restriction
	$\mid\ Net \mid Net$	network composition

$$
\begin{array}{lll}
 & \mid \ \mathit{node}\ n[\mathit{Cro}] & \text{node} \\
\mathit{Cro} & ::= (\boldsymbol{\nu}\, x{:}W)\,\mathit{Cro} & \text{restriction} \\
 & \mid \ \mathit{Cro} \mid \mathit{Cro} & \text{crowd composition} \\
 & \mid \ \texttt{channel}\ c & \text{channel} \\
 & \mid \ \texttt{thread}\ [\mathit{Th}] & \text{thread} \\
\mathit{Th} & ::= \texttt{go}\ n.\mathit{Th} & \text{migration} \\
 & \mid \ \overline{c}(x) & \text{output to a channel} \\
 & \mid \ c(x).\mathit{Th} & \text{input from a channel} \\
 & \mid \ \texttt{fork}\ n[\mathit{Th}].\mathit{Th} & \text{fork a thread} \\
 & \mid \ \texttt{spawn}\ n[\mathit{Cro}].\mathit{Th} & \text{spawn a new node}
\end{array}
$$

As customary, the binders of the language are the ν operator and the input constructs. The type system of the Toy language controls the typing of messages on communication channels. Its formalization follows fairly standard rules, using the following five judgements:

$$
\begin{array}{ll}
\Gamma \vdash \diamond & \text{good environment} \\
\Gamma \vdash x : W & \text{name } x \text{ has type } W \\
\Gamma \vdash \mathit{Net} & \text{good network} \\
\Gamma \vdash \mathit{Cro} & \text{good crowd} \\
\Gamma \vdash \mathit{Th} & \text{good thread}
\end{array}
$$

The type rules are quite standard, and we will not comment them.

Good environment:

$$
\frac{}{\varnothing \vdash \diamond} \qquad \frac{\Gamma \vdash \diamond \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x{:}W \vdash \diamond}
$$

Name has type:

$$
\frac{\Gamma, x{:}W, \Gamma' \vdash \diamond}{\Gamma, x{:}W, \Gamma' \vdash x : W}
$$

Good network:

$$
\frac{\Gamma, x{:}W \vdash \mathit{Net}}{\Gamma \vdash (\boldsymbol{\nu}\, x{:}W)\mathit{Net}} \qquad \frac{\Gamma \vdash \mathit{Net} \quad \Gamma \vdash \mathit{Net}'}{\Gamma \vdash \mathit{Net} \mid \mathit{Net}'} \qquad \frac{\Gamma \vdash n : \mathit{Node} \quad \Gamma \vdash \mathit{Cro}}{\Gamma \vdash \texttt{node}\ n[\mathit{Cro}]}
$$

Good crowd:

$$
\frac{\Gamma, x{:}W \vdash \mathit{Cro}}{\Gamma \vdash (\boldsymbol{\nu}\, x{:}W)\,\mathit{Cro}} \qquad \frac{\Gamma \vdash \mathit{Cro} \quad \Gamma \vdash \mathit{Cro}'}{\Gamma \vdash \mathit{Cro} \mid \mathit{Cro}'}
$$

$$
\frac{\Gamma \vdash c : \mathit{Ch}(W)}{\Gamma \vdash \texttt{channel}\ c} \qquad \frac{\Gamma \vdash \mathit{Th}}{\Gamma \vdash \texttt{thread}\ [\mathit{Th}]}
$$

Good thread:

$$\frac{\Gamma \vdash n : \mathit{Node} \qquad \Gamma \vdash \mathit{Th}}{\Gamma \vdash \texttt{go}\ \ n.\mathit{Th}}$$

$$\frac{\Gamma \vdash c : \mathit{Ch}(W) \qquad \Gamma \vdash n : W}{\Gamma \vdash \overline{c}(n)} \qquad\qquad \frac{\Gamma \vdash c : \mathit{Ch}(W) \qquad \Gamma n{:}W \vdash \mathit{Th}}{\Gamma \vdash c(n).\mathit{Th}}$$

$$\frac{\Gamma \vdash \mathit{Cro} \qquad \Gamma \vdash \mathit{Th}}{\Gamma \vdash \texttt{fork}\ \ (\mathit{Cro}).\mathit{Th}} \qquad\qquad \frac{\Gamma \vdash n : \mathit{Node} \qquad \Gamma \vdash \mathit{Cro} \qquad \Gamma \vdash \mathit{Th}}{\Gamma \vdash \texttt{spawn}\ \ n[\mathit{Cro}].\mathit{Th}}$$

As in the Ambient Calculus encoding, seals model nodes, threads and channels. We will use a three level seals hierarchy, in which the top level corresponds to the underling net, the first level to the nodes which lie on the network, and the second to the threads which are free to migrate between nodes. The process which models the behaviour of a thread lies inside the corresponding seal, and performs all interactions by means of channels to $\uparrow$.

Following our intuitions, we define the translations of threads and nodes in the quite obvious way:

$$\begin{array}{rcl} [\![\, \texttt{node}\ \ n[\mathit{Cro}]\,]\!] & = & n[[\![\, \mathit{Cro}\,]\!] \mid \textsc{Servers}] \\ [\![\, \texttt{thread}\ \ [\mathit{Th}]\,]\!] & = & (\boldsymbol{\nu}\, m{:}\texttt{ThreadName})\ m[[\![\, \mathit{Th}\,]\!]^m] \end{array}$$

This corresponds to constructing the seals which will model respectively the node and the thread, placing inside the first the translation of the crowd of threads and channels (together with some processes we will define later), in the second the process which models thread's behaviour. At the language level, threads are anonymous; in order to deal with them, we assign them a name during translation (something like *thread identifiers*, which are assigned transparently by the run-time support of the language). In translating threads we need the name of the thread, so we label the translation function.

Let us concentrate on communication. We incapsulate messages in packets, represented by seals: the only interaction a packet can be involved is reading the message it carries.

$$\mathit{msg} = p[\overline{rp}^{\uparrow}(\mathit{msg})]$$

A channel is modelled as a seal where packets can be stored and retrieved:

$$[\![\, \texttt{channel}\ c\,]\!] = c[!\ sp^{\uparrow}\{p\} \mid !\ \overline{gp}^{\uparrow}\{p\}]$$

As the original semantics, no assumption on the ordering of packets is done. Threads interact with channel by means of a ChannelServer process, hosted by the node. A thread willing to perform an output operation constructs the packet containing the message, and contacts the ChannelServer, specifying the channel to which the packet is destinated. The newly constructed packet is then moved by the ChannelServer inside the destination channel (or if you prefer, inside the seal which models the destination channel). A thread willing to perform an input operation contacts the ChannelServer, specifing the channel it wants to read from. The ChannelServer moves a packet from the specified channel to the threads, which will read its content, terminating the interaction.

$$
\begin{aligned}
&\textsc{ChannelServer} \stackrel{def}{=} \\
&\qquad !\, op^{\downarrow}(m{:}\texttt{ThreadName}).oc^{m}(ch{:}\texttt{Channel}).om^{m}\{p\}.\overline{sp}^{ch}\{p\}.\overline{sincro}^{m}() \\
&\quad \mid\ !\, ip^{\downarrow}(m{:}\texttt{ThreadName}).ic^{m}(ch{:}\texttt{Channel}).gp^{ch}\{p\}.\overline{im}^{m}\{p\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\, \overline{c}(v)\,]\!]^{m} &= \overline{op}^{\uparrow}(m).\overline{oc}^{\uparrow}(c).\overline{om}^{\uparrow}\{c\}.sincro^{m}() \mid c[\overline{rp}^{\uparrow}(v)] \\
[\![\, c(u{:}W).\, Th\,]\!]^{m} &= (\boldsymbol{\nu}\, p{:}\texttt{Id}\,[\,])\ \overline{ip}^{\uparrow}(m).\overline{ic}^{\uparrow}(c).im^{\uparrow}\{p\}.rp^{p}(u{:}[\![\, W\,]\!]).[\![\, Th\,]\!]^{m}
\end{aligned}
$$

Notice that the input operation declares a new name for the incoming packet: this allows garbage collection of "empty packets".

Unfortunately, this encoding is not able to distinguish at the type level the type of the read message: the channel used to read content of the packet must have fixed type. In our Toy language, messages are names of channels and names of nodes, so both must have the same type. We will discuss this at the end of the section.

More intuitive encodings usually fail to work. For instance, modelling channels as wrapper processes like

$$[\![\, channel\ c\,]\!] = !c^{\downarrow}(x).c\_n^{\downarrow}(n).\overline{c}^{n}(x)$$

require that the type of the thread contains all the channels from which he might perform an input operation. Also allowing this, it is easy to fall in the necessity of employing recursive types due, for example, to recursive definition of the interfaces of threads and nodes.

Now, let us concentrate on migration of threads. In order to have the possibility of displacing a thread process, we encapsulated it inside a seal. A seal cannot autonomously move, it must be moved by processes located in the environment: at the same time in our Toy language the thread is the only responsible for its movement. We translate the movement operation that a thread can unleash in a request to a *mobility server*, to be found at the level of the node and so able to move the seal which encloses the thread. At the same time, the thread suspends itself, until it is notified of being in the new node.

The *mobility server* is splitten in two parts. One (called ServerMobGen) is responsible for the transport of threads at the network level, while the other one (the ServerMobLoc) is responsible for the transport at the node level: thus it is replicated inside every node, listening for request from threads. When a thread asks to be moved to another node, the ServerMobLoc contacts the ServerMobGen and together they move the thread to the network level. Then the ServerMobGen contacts the ServerMobLoc local to the destination node, and together they move the thread from the network level inside the destination node.

In order to avoid interference, since we are interested only in specification problem, we introduce the *sequential bang* notation ¡. The operator makes independent copies of the process P, which will execute sequentially. A possible implementation, in which we require that P does not contain any composition operator, is the following:

$$¡P \stackrel{def}{=} (\boldsymbol{\nu}\, s{:}\mathtt{Ch\ Shh})\ !\ s^*().P.\overline{s}^*()\ |\ \overline{s}^*() \qquad \text{for } s \notin fn(P)$$

$$\begin{array}{l}
\textsc{ServerMobLoc} \stackrel{def}{=} \\
\quad ¡\ mob^{\downarrow}(m{:}\mathtt{ThreadName}).mob_1^{m}(dest{:}\mathtt{NodeName}). \\
\quad \overline{mobG}^{\uparrow}(m).\overline{mobG_1}^{\uparrow}(dest).\overline{mobGS}^{\uparrow}\{m\} \\
|\quad ¡\ mobG^{\uparrow}(m{:}\mathtt{ThreadName}).mobGS^{\uparrow}\{m\}.\overline{sincro}^{m}() \\
\textsc{ServerMobGen} \stackrel{def}{=} \\
\quad ¡\ mobG^{\downarrow}(m{:}\mathtt{ThreadName}).mobG_1^{\downarrow}(dest{:}\mathtt{NodeName}).mobGS^{\downarrow}\{m\}. \\
\quad \overline{mobG}^{dest}(m).\overline{mobGS}^{dest}\{m\}
\end{array}$$

$$[\![\, \mathsf{go}\ n.\,Th\,]\!]^m = \overline{mob}^{\uparrow}(m).\overline{mob\_1}^{\uparrow}(n).sincro^{\uparrow}().[\![\ Th\,]\!]^m$$

The `fork` and `spawn` operators are encoded much in the same way, up-to defining appropriate servers.

When all the servers are defined, the minimal types that can be assigned to the name of a node and to the name of a channel are:

$$\begin{array}{rcl}
\mathtt{NodeName} & = & \mathtt{Id}\ [mobG : \mathtt{ThreadName}\ ; mobGS : \mathtt{Thread}\ ; sincro : \mathtt{Shh}] \\
\mathtt{ChannelName} & = & \mathtt{Id}\ [sp : \mathtt{Packet}]
\end{array}$$

As they must be the same type, we define a common interface:

$$\begin{array}{rcl}
\mathtt{Resource} & = & \mathtt{NodeName}\ \oplus\ \mathtt{ChannelName} \\
\mathtt{Message} & = & \mathtt{Id\ Resource}
\end{array}$$

The full translation is reported below. We use as before type alias to simplify the reading of the code, and we report in a *symbol table* the list of the names declared as global.

Type Aliases

$$
\begin{array}{rcl}
\texttt{Resource} & = & [\mathit{mobG} : \texttt{ThreadName}\ ; \mathit{mobGS} : \texttt{Thread}\ ; \mathit{sincro} : \texttt{Shh}\ ; \mathit{sp} : \texttt{Packet}] \\
\texttt{Message} & = & \texttt{Id Resource} \\
\texttt{Thread} & = & [\mathit{im} : [\,]\ ; \mathit{sincro} : \texttt{Shh}] \\
\texttt{Node} & = & \texttt{Resource} \\
\texttt{ThreadName} & = & \texttt{Id Thread} \\
\texttt{NodeName} & = & \texttt{Id Node}\ (= \texttt{Message}) \\
\texttt{Packet} & = & [\,] \\
\texttt{Channel} & = & \texttt{Id Resource}\ (= \texttt{Message})
\end{array}
$$

Symbol Table

name	type	between		
sincro	`Ch Shh`			
op	`Ch ThreadName`	thread	$\rightarrow$	CS
oc	`Ch Channel`	thread	$\rightarrow$	CS
om	`Ch Packet`	thread	$\rightarrow$	CS
sp	`Ch Packet`	SC	$\rightarrow$	channel
ip	`Ch ThreadName`	thread	$\leftarrow$	CS
ic	`Ch Channel`	thread	$\leftarrow$	CS
im	`Ch Packet`	thread	$\leftarrow$	CS
gp	`Ch Packet`	SC	$\leftarrow$	channel
rp	`Ch Message`	packet	$\rightarrow$	thread
mob	`Ch ThreadName`	thread	$\rightarrow$	SML
mob_1	`Ch NodeName`	thread	$\rightarrow$	SML
mobG	`Ch ThreadName`	SML	$\leftrightarrow$	SMG
mobG_1	`Ch NodeName`	SML	$\leftrightarrow$	SMG
mobGS	`Ch Thread`	SML	$\leftrightarrow$	SMG
fork	`Ch ThreadName`	thread	$\rightarrow$	SF
forkS	`Ch Thread`	thread	$\rightarrow$	SF
spawn	`Ch NodeName`	thread	$\rightarrow$	SSL
spawnS	`Ch Node`	thread	$\rightarrow$	SSL
spawnG	`Ch NodeName`	SSL	$\rightarrow$	SSG
spawnGS	`Ch Node`	SSL	$\rightarrow$	SSG

Servers

$$\textsc{ServerMobLoc} \stackrel{def}{=}$$
$$\text{¡}\ mob^{\downarrow}(m{:}\mathtt{ThreadName}).mob\_1^{m}(dest{:}\mathtt{NodeName}).$$
$$\overline{mobG}^{\uparrow}(m).\overline{mobG\_1}^{\uparrow}(dest).\overline{mobGS}^{\uparrow}\{m\}$$
$$\mid\quad \text{¡}\ mobG^{\uparrow}(m{:}\mathtt{ThreadName}).mobGS^{\uparrow}\{m\}.\overline{sincro}^{m}()$$

$$\textsc{ServerMobGen} \stackrel{def}{=}$$
$$\text{¡}\ mobG^{\downarrow}(m{:}\mathtt{ThreadName}).mobG\_1^{\downarrow}(dest{:}\mathtt{NodeName}).mobGS^{\downarrow}\{m\}.$$
$$\overline{mobG}^{dest}(m).\overline{mobGS}^{dest}\{m\}$$

$$\textsc{ServerFork} \stackrel{def}{=}$$
$$\text{¡}\ fork^{\downarrow}(m{:}\mathtt{ThreadName}).forkS^{\downarrow}\{m\}.\overline{sincro}^{m}()$$

$$\textsc{ServerSpawnLoc} \stackrel{def}{=}$$
$$\text{¡}\ spawn^{\downarrow}(n{:}\mathtt{NodeName}).spawnS^{\downarrow}\{n\}.\overline{spawnG}^{\uparrow}(n).\overline{spawnGS}^{\uparrow}\{n\}$$

$$\textsc{ServerSpawnGen} \stackrel{def}{=}$$
$$\text{¡}\ spawnG^{\downarrow}(n{:}\mathtt{NodeName}).spawnGS^{\downarrow}\{n\}.\overline{sincro}^{n}()$$

$$\textsc{ChannelServer} \stackrel{def}{=}$$
$$!\ op^{\downarrow}(m{:}\mathtt{ThreadName}).oc^{m}(ch{:}\mathtt{Channel}).om^{m}\{p\}.\overline{sp}^{ch}\{p\}.\overline{sincro}^{m}()$$
$$\mid\quad !\ ip^{\downarrow}(m{:}\mathtt{ThreadName}).ic^{m}(ch{:}\mathtt{Channel}).gp^{ch}\{p\}.\overline{im}^{m}\{p\}$$

Encoding of the distributed Toy language

$\mathcal{T}(Net) = (\boldsymbol{\nu}\,\textsc{SymbolTable})\ (\ [\![Net]\!] \mid \textsc{ServerMobGen} \mid \textsc{ServerSpawnGen}\ )$

Translation of Types

$[\![Node]\!] = \mathtt{NodeName}$
$[\![Ch(W)]\!] = \mathtt{Channel}$

Translation of Nets

$[\![(\boldsymbol{\nu}\, n{:}W)Net]\!] = (\boldsymbol{\nu}\, n{:}[\![W]\!])[\![Net]\!]$
$[\![Net \mid Net]\!] = [\![Net]\!] \mid [\![Net]\!]$
$[\![\mathtt{node}\ n[Cro]]\!] = n[[\![Cro]\!] \mid \textsc{ServerMobLoc} \mid \textsc{ServerFork} \mid \textsc{ServerSpawnLoc} \mid \textsc{ChannelServer}]$

Translation of Crowds

$[\![(\boldsymbol{\nu}\, n{:}W)Cro]\!] = (\boldsymbol{\nu}\, n{:}[\![W]\!])[\![Cro]\!]$
$[\![Cro \mid Cro]\!] = [\![Cro]\!] \mid [\![Cro]\!]$
$[\![\mathtt{channel}\ c]\!] = c[!\ sp^{\uparrow}\{p\} \mid !\ \overline{gp}^{\uparrow}\{p\}]$
$[\![\mathtt{thread}\ [Th]]\!] = (\boldsymbol{\nu}\, m{:}\mathtt{ThreadName})\ m[[\![Th]\!]^{m}]$

Translation of Threads

$[\![\overline{c}(v)]\!]^{m} = \overline{op}^{\uparrow}(m).\overline{oc}^{\uparrow}(c).\overline{om}^{\uparrow}\{c\}.sincro^{m}() \mid c[\overline{rp}^{\uparrow}(v)]$
$[\![c(u{:}W).Th]\!]^{m} = (\boldsymbol{\nu}\, p{:}\mathtt{Id}\ [\,])\ \overline{ip}^{\uparrow}(m).\overline{ic}^{\uparrow}(c).im^{\uparrow}\{p\}.rp^{p}(u{:}[\![W]\!]).[\![Th]\!]^{m}$
$[\![\mathtt{go}\ n.Th]\!]^{m} = \overline{mob}^{\uparrow}(m).\overline{mob\_1}^{\uparrow}(n).sincro^{\uparrow}().[\![Th]\!]^{m}$
$[\![\mathtt{fork}\ [Th_1].Th_2]\!]^{m} = (\boldsymbol{\nu}\, m'{:}\mathtt{ThreadName})$
$\qquad (\ m'[sincro^{\uparrow}().[\![Th_1]\!]^{m'}] \mid \overline{fork}^{\uparrow}(m').\overline{forkS}^{\uparrow}\{m'\}.[\![Th_2]\!]^{m}\ )$
$[\![\mathtt{spawn}\ n[Cro].Th]\!]^{m} = n[sincro^{\uparrow}().[\![Cro]\!]^{m}] \mid \overline{spawn}^{\uparrow}(n).\overline{spawnS}^{\uparrow}\{n\}.[\![Th]\!]^{m}$

A whole bunch of properties and proofs related to this encoding can be found in [36].

This example shows that the Seal Calculus, like the Ambient Calculus, serves as a semantic meta-language for describing distributed computation. The mobility primitives of Seal follows for security reason the objective moves style, but are sufficiently expressive to encode subjective moves, at least limited to a flat structure of locations.

5.3 Immobile Seals

Seals have been designed with movement in mind. Nevertheless, in the programmer's mind some seals are immobile, mimicking the behaviour of immobile entities of the reality: for instance, in the encoding of the distributed Toy language we used seals to represent the immobile node of the networks and it would be appealing having the possibility of statically checking that these seal are immobile. Only a misuse, a programming error, could cause them to go somewhere. There is a little remark to be done here: nodes are immobile since they have been created. A `spawn` cause a new node to be brought into existence inside the thread and then moved to the network level, before becoming a fully-effect immobile node. So we are really interested in getting a fine control on the possibility of movement.

We enrich our type system extending the types with one attribute indicating wether a seal can be moved at all. The same extension has already been carried on in the Ambient Calculus framework, where it appears to be a restrictive extension. In particular an ambient which is declared immobile cannot accept inside himself any ambient which is declared mobile. This is a consequence of the semantic of the movement capabilities and of the presence of the `open` operation. In order to get a finer control on mobility, Cardelli, Ghelli and Gordon had to introduce what they call "objective moves": a sort of ambient inizialization sequence of movement capabilities, which are fired before the ambient become active.[2] They admit that an objectively mobile ambient can still enter a subjectively immobile ambient, a brilliant and elegant solution, even in despite the addition syntax introduced in the calculus.

In Seal Calculus, this extension does not have the dramatic consequences it has in Ambient Calculus, and we will still allow a seal enter or exit from a seal declared immobile. Here follows the new type syntax:

V	$::= M$	name
	$\mid A$	agent
M	$::= \mathtt{Shh}$	silent
	$\mid \mathtt{Ch}\ V$	channel name

[2] Be careful on terminology: objective moves à la Cardelli are quite unrelated from our distinction between objective and subjective moves.

$$
\begin{array}{lll}
 & \mid\ \mathtt{Id}^{Z}\, A & \text{agent name} \\[1em]
Z & ::= \curvearrowright & \text{mobile} \\
 & \mid\ \veebar & \text{immobile} \\[1em]
A & ::= [x_1{:}V_1; \cdots ; x_n{:}V_n] & \\[1em]
T & ::= M & \\
 & \mid\ \mathtt{Proc}\ A &
\end{array}
$$

The only modification is the little superscript Z in the type of the agent names, whose interpretation is quite intuitive:

$\vdash x : \mathtt{Id}^{\curvearrowright} A$ means that x is the name of a seal with interface A which *may* be moved.

$\vdash x : \mathtt{Id}^{\veebar} A$ means that x is the name of a seal with interface A which *cannot* be moved.

The type rules are modified according to our intuition: a process can try to send somewhere a seal only if this has been declared as a *mobile* one. For instance (Send Up and Send Down cases are straightforward):

(Snd Local)
$$\frac{\Gamma \vdash_{\Xi} x : \mathtt{Ch}\ A \qquad \Gamma \vdash_{\Xi} y : \mathtt{Id}^{\curvearrowright} A \qquad \Gamma \vdash_{\Xi} P : \mathtt{Proc}\ A'}{\Gamma \vdash_{\Xi} \overline{x}^{*}\{y\}.P : \mathtt{Proc}\ A'}$$

Notice that we require the seal we want to move to be declared as mobile ($\curvearrowright$), and this is the only modification we need.

In Seal's model of mobility, when a seal is received takes a new name, selected by the receiver process. We use this characteristic, together with the fact that the mobility attribute is tied to seals names, to turn a previously mobile seal into an immobile one. For instance,

$$(\boldsymbol{\nu}\, x{:}\mathtt{Ch}\ A)(\boldsymbol{\nu}\, a{:}\mathtt{Id}^{\curvearrowright} A)(\boldsymbol{\nu}\, b{:}\mathtt{Id}^{\veebar} A)\ \overline{x}^{*}\{a\} \mid x^{*}\{b\} \mid a[P] \ \rightarrow\ (\boldsymbol{\nu}\, b{:}\mathtt{Id}^{\veebar} A)\ b[P]$$

the mobile seal named a becomes magically an immobile seal named b. This is achieved by imposing no constraints on the mobility attribute of the receiving name in the receive typing rule:

(Rcv Local)
$$\frac{\Gamma \vdash_{\Xi} x : \mathtt{Ch}\ A \qquad \Gamma \vdash_{\Xi} y : \mathtt{Id}^{Z} A \qquad \Gamma \vdash_{\Xi} P : \mathtt{Proc}\ A'}{\Gamma \vdash_{\Xi} x^{*}\{y\}.P : \mathtt{Proc}\ A'} \quad x \in \mathrm{dom}(\Xi)$$

The extension to the non local cases are straightforward, so we report our second type system.

(Type Shh)
$$\frac{\Gamma \vdash_\Xi \diamond}{\Gamma \vdash_\Xi \mathtt{Shh}}$$

(Type Id)
$$\frac{\Gamma \vdash_\Xi A}{\Gamma \vdash_\Xi \mathtt{Id}^Z\, A}$$

(Type Ch)
$$\frac{\Gamma \vdash_\Xi V}{\Gamma \vdash_\Xi \mathtt{Ch}\; V}$$

(Type Proc)
$$\frac{\Gamma \vdash_\Xi A}{\Gamma \vdash_\Xi \mathtt{Proc}\; A}$$

(Type Interface)
$$\frac{\Gamma \vdash_\Xi x_i : \mathtt{Ch}\; V_i \qquad x_i \in \mathrm{dom}(\Xi)}{\Gamma \vdash_\Xi [x_1{:}V_1, \ldots, x_n{:}V_n]} \quad \forall i \in 1..n$$

(Proc Dead)
$$\frac{\Gamma \vdash_\Xi \diamond}{\Gamma \vdash_\Xi \mathbf{0} : \mathtt{Proc}\; [\,]}$$

(Proc Par)
$$\frac{\Gamma \vdash_\Xi P_1 : \mathtt{Proc}\; A \qquad \Gamma \vdash_\Xi P_2 : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi P_1 \mid P_2 : \mathtt{Proc}\; A}$$

(Proc Res Agent)
$$\frac{\Gamma, x{:}\mathtt{Id}^Z\, A \vdash_\Xi P : \mathtt{Proc}\; A'}{\Gamma \vdash_\Xi (\boldsymbol{\nu}\, x{:}\mathtt{Id}^Z\, A)P : \mathtt{Proc}\; A'}$$

(Proc Res Ch)
$$\frac{\Gamma, x{:}\mathtt{Ch}\; V \vdash_{\Xi,x} P : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi (\boldsymbol{\nu}\, x{:}\mathtt{Ch}\; V)P : \mathtt{Proc}\; A} \quad x \notin fn(A)$$

(Proc Seal)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Id}^Z\, A \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi x[P] : \mathtt{Proc}\; [\,]}$$

(Proc Bang)
$$\frac{\Gamma \vdash_\Xi P : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi !P : \mathtt{Proc}\; A}$$

(Output Local)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\; M \qquad \Gamma \vdash_\Xi y : M \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi \overline{x}^*(y).P : \mathtt{Proc}\; A}$$

(Input Local)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\; M \qquad \Gamma, y{:}M \vdash_\Xi P : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi x^*(y{:}M).P : \mathtt{Proc}\; A} \quad x \in \mathrm{dom}(\Xi)$$

(Output Local)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\; M \qquad \Gamma \vdash_\Xi y : M \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi \overline{x}^*(y).P : \mathtt{Proc}\; A}$$

(Output Up)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\; M \qquad \Gamma \vdash_\Xi y : M \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi \overline{x}^\uparrow(y).P : \mathtt{Proc}\; A}$$

(Input Up)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\; M \qquad \Gamma, y{:}M \vdash_\Xi P : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi x^\uparrow(y{:}M).P : \mathtt{Proc}\; (A \oplus [x{:}M])} \quad x \in \mathrm{dom}(\Xi)$$

(Output Down)
$$\frac{\Gamma \vdash_\Xi z : \mathtt{Id}^Z\, A' \qquad \Gamma \vdash y : M \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi \overline{x}^z(y).P : \mathtt{Proc}\; A} \quad (x{:}M) \in A'$$

(Input Down)
$$\frac{\Gamma \vdash_\Xi z : \mathtt{Id}^Z\, A' \qquad \Gamma \vdash_\Xi x : \mathtt{Ch}\; M \qquad \Gamma, y{:}M \vdash_\Xi P : \mathtt{Proc}\; A}{\Gamma \vdash_\Xi x^z(y{:}M).P : \mathtt{Proc}\; A} \quad x \in \mathrm{dom}(\Xi)$$

(Rcv Local)

$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^Z\ A \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\ A'}{\Gamma \vdash_\Xi x^*\{y\}.P : \mathtt{Proc}\ A'} \quad x \in \mathrm{dom}(\Xi)$$

(Snd Local)

$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^{\curvearrowright}\ A \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\ A'}{\Gamma \vdash_\Xi \overline{x}^*\{y\}.P : \mathtt{Proc}\ A'}$$

(Rcv Up)

$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^Z\ A \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\ A'}{\Gamma \vdash_\Xi x^\uparrow\{y\}.P : \mathtt{Proc}\ (A' \cup [x{:}A])} \quad x \in \mathrm{dom}(\Xi)$$

(Snd Up)

$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^{\curvearrowright}\ A \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\ A'}{\Gamma \vdash_\Xi \overline{x}^\uparrow\{y\}.P : \mathtt{Proc}\ A'}$$

(Rcv Down)

$$\frac{\Gamma \vdash_\Xi z : \mathtt{Id}^{Z'}\ A' \qquad \Gamma \vdash_\Xi x : \mathtt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^Z\ A \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\ A''}{\Gamma \vdash_\Xi x^z\{y\}.P : \mathtt{Proc}\ A''} \quad x \in \mathrm{dom}(\Xi)$$

(Snd Down)

$$\frac{\Gamma \vdash_\Xi z : \mathtt{Id}^Z\ A_1 \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^{\curvearrowright}\ A \qquad \Gamma \vdash_\Xi P : \mathtt{Proc}\ A_2}{\Gamma \vdash_\Xi \overline{x}^z\{y\}.P : \mathtt{Proc}\ A_1} \quad (x{:}A) \in A_1$$

Now we can revisit the distribuited Toy language encoding in order to declare nodes as immobile. Major hacks are to be done in the typing of nodes and threads names, and in the spawn code, in order to get able to initial move the newly created node up to the network level.

$$\begin{array}{rcl} \mathtt{ThreadName} & = & \mathtt{Id}^{\curvearrowright}\ \mathtt{Thread} \\ \mathtt{NodeName} & = & \mathtt{Id}^{\veebar}\ \mathtt{Message} \\ \mathtt{TempNodeName} & = & \mathtt{Id}^{\curvearrowright}\ \mathtt{Message} \\ \mathtt{Channel} & = & \mathtt{Id}^{\veebar}\ \mathtt{Message} \end{array}$$

We add a new type `TempNodeName` to denote the names of the node which have been created by a `spawn` but do not have reached the network level, which differs from `NodeName` in the mobility attribute. Threads are declared as mobile, while channels are declared immobile. Notice that when a thread is moved, all the channels which lies inside it are moved accordingly, even if they are declared immobile.

The new spawn server uses a temporary node name t (declared mobile) in order to move the new node up to the network level, where it takes the definitive node name n (declared immobile).

$\textsc{ServerSpawnLoc} \stackrel{def}{=}$

$\quad ¡\, spawn^{\downarrow}(n{:}\texttt{NodeName}).spawn1^{\downarrow}(t{:}\texttt{TempNodeName}).spawnS^{\downarrow}\{t\}.\overline{spawnG}^{\uparrow}(n).\overline{spawnGS}^{\uparrow}\{t\}$

$\textsc{ServerSpawnGen} \stackrel{def}{=}$

$\quad ¡\, spawnG^{\downarrow}(n{:}\texttt{NodeName}).spawnGS^{\downarrow}\{n\}.\overline{sincro}^{n}()$

We add a new name to the symbol table

spawn1	`Ch TempNodeName`	thread	$\rightarrow$	SSL

and we adapt slightly the `spawn` encoding in order to use the temporary node name to send the node to the spawn server.

$$\begin{array}{rcl} [\![\, \texttt{spawn}\ n[Cro].\,Th \,]\!]^m & = & (\boldsymbol{\nu}\, t{:}TempNodeName)\ (t[sincro^{\uparrow}().[\![\, Cro \,]\!]^m] \\ & & \mid\ \overline{spawn}^{\uparrow}(n).\overline{spawn1}^{\uparrow}(t).\overline{spawnS}^{\uparrow}\{t\}.[\![\, Th \,]\!]^m\ ) \end{array}$$

6

Conclusion

In this work we propose a new version of the Seal Calculus as a common framework for the study of distributed and mobile computation. The original calculus was quite untractable: the first phase of our work was to simplify its design without loosing much expressive power. The new calculus differs from the original in the existence of shared channels between seals in parent-child relationship, opposed to the remote interaction on channels local to seals, in the absence of portals, and in the impossibility of duplicating seals.

We define a type system that guarantees two main properties. Firstly, it ensures that only appropriate values will transit over channels of a given type, that is, it types communication. Secondly, it makes sure that the mobility operations do not invalidate the protocol to deal with an agent.

In order to achieve the latter result, which constitutes the main novelty of our work, we define a notion of *interface* that abstracts the protocol to interact with an agent. The definition of the interfaces is tightly linked to the model of computation of Seal Calculus: for instance, in no way it could be adapted to Ambient Calculus. Our type system appears to work quite well in capturing the possible effects an agent can have on its environment: interactions with the environment are essential in the world of distributed computation and being able to capture them statically is a fundamental step.

We obtain a standard subject reduction result: the invariance of types during reduction has been achieved by the development of three ideas:

1. Movement of seals takes place on typed channels, and the type of the channel ensures that when the seal is reactivated in the new location it has exactly the same type (interface) as when it started moving.

2. We introduced subtyping to guarantee that the interface of seal remains the same also in case the process inside the seal performs some actions.

3. The calculus has been restricted with the condition that a channel bound in an input operation is only used to perform output operation. This condition is not restrictive, as proved by the previous experiences with actual programming languages based on π-calculus.

We extend the type system with the possibility of declaring a seal as *mobile* or *immobile*, being this one of the simplest properties related to mobility one can imagine. The extension gives a fine-grained control on the movement of seals without raising the necessity of extending the syntax of the calculus. For instance, it is possible for an immobile seal to accept mobile seals willing to enter inside it, and it is possible to turn a previously mobile seal into an immobile one. The simplicity of the extension suggests that Seal Calculus and the proposed type system are well suited to be extended in many directions, resulting at the same time expressive and malleable.

In order to check the claimed expressivity of the new calculus, we give semantics to a typed distributed language via a translation into Seal Calculus. The same language made its apparition in [11]: in that case its semantics has been given via a translation into Ambient Calculus. The parallel between the two calculi appears interesting.

Further work There are two main directions along which we plan to complete this work.

First of all, we will terminate the parallel between Seal Calculus and Ambient Calculus by enriching our type system with *single threaded types*, in the style of the work on Safe Ambients by Levi and Sangiorgi. A term that respects the single threaded typing rules presents inside each location at most one process which has the control thread and which can consequently exercise some action. The type-system should allow us to prove equational laws relating the two sides of the reduction rules, in order to develop an algebraic theory of the calculus. In Ambient Calculus these laws are provided under restrictive conditions. We argue that these external conditions are not needed in Seal Calculus, and we hope that this will result into a more tractable algebraic theory.

Secondly, in highly distributed systems it is clearly unrealistic to rely on a static discipline of type system, and on a static type-checking of the whole network. It will be intriguing study how our notion of interface interacts with the requirements imposed by a distributed environment. In particular, it would be interesting being able of performing the typechecking of a term *just-in-time*, when a seal enters one another. Each seal could carry its interface and the interface should be expressive enough in order to typecheck the interactions the seal will perform by respect the newly enclosing seal. We argue that this should be possible by imposing a second restriction on the calculus: a seal performs output towards its enclosing seal only on channels which have been communicated to the seal by the parent seal. As all the channels a seal may read from its enclosing ambient appear in the interface, this second restriction might give to the interface a complete control on the interaction between the seal and its surrounding environment. The main novelty of this extension would be that in order to perform the just-in-time typechecking each seal should carry only its interface, and not the whole typing environment.

Bibliography

[1] M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the Spi calculus. *Lecture Notes in Computer Science*, 1243, 1997.

[2] Samson Abramsky. Interaction categories and typed concurrent programming. In *Proceedings of the International Summer School of Marktoberdorf*, NATO Advanced Science Institutes, series F. Springer-Verlag, 1994.

[3] Torben Amtoft, Assaf J. Kfoury, and Santiago M. Pericas-Geertsen. What are polymorphic ambients? Submitted for conference publication, July 2000.

[4] Gregory R. Andrews. *Concurrent Programming*. Benjamin/Cunnings, 1991.

[5] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[6] M. Bugliesi and G. Castagna. Mobile objects. In *FOOL*, 2000.

[7] L. Cardelli. Abstractions for mobile computations. *Lecture Notes in Computer Science*, 1603:51–94, 1999.

[8] L. Cardelli and G. Ghelli. A query language for semistructured data based on the ambient logic. DRAFT, 2000.

[9] L. Cardelli and A. D. Gordon. Mobile ambients. *Lecture Notes in Computer Science*, 1378, 1998.

[10] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

[11] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In Jiří Wiederman, Peter van Emde Boas, and Mogens Nielsen, editors, *26th Colloquium on Automata, Languages and Programming (ICALP) (Prague, Czech Republic)*, volume 1644 of *lncs*, pages 230–239. sv, July 1999.

[12] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In Catuscia Palamidessi, editor, *CONCUR 2000: Concurrency Theory (11th International Conference, University Park, PA, USA)*, volume 1877 of *lncs*, pages 365–379. sv, August 2000.

[13] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[14] U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. Technical Report DAIMI PB-208, Computer Science Department, University of Aarhus, 1986.

[15] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory (7th International Conference, Pisa, Italy, August 1996, Proceedings)*, volume 1119 of *lncs*, pages 406–421. sv, 1996.

[16] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc IJCAI '73*, pages 235–245, 1973.

[17] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. Reprinted in "Distributed Computing: Concepts and Implementations" edited by McEntire, O'Reilly and Larson, IEEE, 1984.

[18] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1984.

[19] C. A. R. Hoare and C. B. Jones. *Essays in Computing Science*. Prentice-Hall, 1989.

[20] Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 352–364, Boston, Massachusetts, January 19–21, 2000.

[21] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal System*, pages 33–69. North Holland, 1963.

[22] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[23] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[24] Robin Milner. Functions as processes. Research Report 1154, INRIA, Sofia Antipolis, 1990. Final version in *Journal of Mathematical Structures in Computer Science* 2(2):119–141, 1992.

[25] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[26] Robin Milner. Elements of interaction. *Communications of ACM*, 36(1):78–89, 1993.

[27] Eugenio Moggi. An abstract view of programming languages. LFCS Report ECS–LFCS–90–113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, April 1990.

[28] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 1996.

[29] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[30] Dana Scott. Outline of a mathematical theory of computation. In *4th Annual Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970.

[31] Peter Sewell. Global/local subtyping and capability inference for a distributed pi-calculus. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *25th Colloquium on Automata, Languages and Programming (ICALP) (Aalborg, Denmark)*, volume 1443 of *lncs*, pages 695–706. sv, July 1998. Full version as Technical Report 435, Computer Laboratory, University of Cambridge.

[32] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Wrappers and causality types. Technical Report 478, Computer Laboratory, University of Cambridge, 1999.

[33] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[34] David N. Turner. *The Polymorphic Pi-calulus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1995.

[35] Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. In *LICS*, 2000.

[36] Francesco Zappa Nardelli. Types for seal calculus. Forthcoming Master Thesis, October 2000.

Appendix A

Proof of subject reduction

Lemma 3 (Generation) *If* $\Gamma \vdash_{\Xi} P : \texttt{Proc}\ A$ *then* $\Gamma \vdash_{\Xi} \diamond$ *and* $\Gamma \vdash_{\Xi} A$*.*

Lemma 4 (Weakening) *If* $\Gamma \vdash_{\Xi} P : \texttt{Proc}\ A$*,* $x \notin dom(\Gamma)$ *and* $\Gamma, x{:}M \vdash_{\Xi,(x)} \diamond$ *then* $\Gamma, x{:}M \vdash_{\Xi,(x)} P : \texttt{Proc}\ A$*.*

Lemma 5 (Strengthening) *If* $\Gamma, x{:}M \vdash_{\Xi,(x)} P : \texttt{Proc}\ A$ *and* $x \notin fn(P)$ *then* $\Gamma \vdash_{\Xi} P : \texttt{Proc}\ A$*.*

Lemma 6 (Exchange) *If* $\Gamma, x{:}M, y{:}M' \vdash_{\Xi,(x),(y)} P : \texttt{Proc}\ A$*,* $x \notin fn(M')$ *and* $y \notin fn(M)$ *then* $\Gamma, y{:}M', x{:}M \vdash_{\Xi,(y),(x)} P : \texttt{Proc}\ A$*.*

Relations between type rules and typing algorithm.

Lemma 7 *If* $\Gamma \rhd_{\Xi} P : \texttt{Proc}\ A$ *then* $\exists A'$ *such that* $\texttt{Proc}\ A' \leq \texttt{Proc}\ A$ *and* $\Gamma \rhd P : \texttt{Proc}\ A'$*.*

Lemma 8 *If* $\Gamma \rhd_{\Xi} P : \texttt{Proc}\ A$ *then* $\Gamma \vdash_{\Xi} P : \texttt{Proc}\ A$*.*

We will work on the algorithmic derivation (noted $\rhd$).

NOTATION: We also define $A_1 \setminus [x{:}M]$ to represent the ensemble of agents in which we *may* have erased $[x{:}M]$ from A_1. This trivially extends to the case $A_1 \setminus A_2$. Notice that, if $A \in A_1 \setminus A_2$, $\texttt{Proc}\ A$ is always a subtype of $\texttt{Proc}\ A_1$ (the subtyping relation will be defined in Section 4.3.1). Also note that writing $A \setminus [x{:}V]$ implies $[x{:}V] \in A$. We will use this operator in the *(move in)* and *(write in)* case of the subject reduction proof.

Proposition 3 (Subject Congruence) *If* $\Gamma \rhd_{\Xi} P : \texttt{Proc}\ A$ *and* $P \equiv Q$ *then* $\Gamma \rhd_{\Xi} Q : \texttt{Proc}\ A$*.*

Proof The proposition follows by showing that $P \equiv Q$ implies:

1. If $\Gamma \rhd_{\Xi} P : \texttt{Proc}\ A$ then $\Gamma \rhd_{\Xi} Q : \texttt{Proc}\ A$.
2. If $\Gamma \rhd_{\Xi} Q : \texttt{Proc}\ A$ then $\Gamma \rhd_{\Xi} P : \texttt{Proc}\ A$.

We proceed by induction on the depth of the derivation of $P \equiv Q$. We perform a case analysis on the last applied rule.

(Struct Refl) Trivial.

(Struct Comm) Then $Q \equiv P$. For (1), assume $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$. By induction hypothesis (2), $Q \equiv P$ implies $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$. Part (2) is symmetric.

(Struct Trans) Then $P \equiv Q, Q \equiv R$. For (1), assume $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$. By the induction hypothesis (1) $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$. Again by the induction hypothesis (1) $\Gamma \rhd_\Xi R : \mathtt{Proc}\ A$. Part (2) is symmetric.

(Struct Dead Par) Then $P = P' \mid \mathbf{0}$ and $Q = P'$.

For (1), assume $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$. This must have been derived from *(Proc Par algo)* with $\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A'$, $\Gamma \rhd_\Xi \mathbf{0} : \mathtt{Proc}\ [\,]$, and $A = A' \oplus [\,]$. Since $[\,]$ is the unit of $\oplus$, $A = A'$. Hence $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$.

For (2), assume $\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A$. By *(Proc Dead)* $\Gamma \rhd_\Xi \mathbf{0} : \mathtt{Proc}\ [\,]$ and by *(Proc Par)* we have $\Gamma \rhd_\Xi P' \mid \mathbf{0} : \mathtt{Proc}\ (A \oplus [\,])$. Hence $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$.

(Struct Par Comm) Then $P = (P' \mid P'')$ and $Q = (P'' \mid P')$. For (1), assume $\Gamma \rhd_\Xi P' \mid P'' : \mathtt{Proc}\ A$. This must have been derived by *(Proc Par)* from $\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A'$ and $\Gamma \rhd_\Xi P'' : \mathtt{Proc}\ A''$, with $A = A' \oplus A''$. By *(Proc Par)* we can derive $\Gamma \rhd_\Xi P'' \mid P' : \mathtt{Proc}\ A'' \oplus A'$: since $\oplus$ is commutative we obtain $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$. Part (2) is symmetric.

(Struct Par Assoc) Then $P = ((P' \mid P'') \mid P''')$ and $Q = (P' \mid (P'' \mid P'''))$. For (1), assume $\Gamma \rhd_\Xi ((P' \mid P'') \mid P''') : \mathtt{Proc}\ A$. This must have been derived from *(Proc Par)* twice, with $\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A'$, $\Gamma \rhd_\Xi P'' : \mathtt{Proc}\ A''$ and $\Gamma \rhd_\Xi P''' : \mathtt{Proc}\ A'''$. Again by *(Proc Par)* twice, since $\oplus$ is associative, we can derive $\Gamma \rhd_\Xi (P' \mid (P'' \mid P''')) : \mathtt{Proc}\ A$. Hence $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$. Part (2) is symmetric.

(Struct Repl Par) Then $P = !P'$ and $Q = !P' \mid P'$.

For (1), assume $\Gamma \rhd_\Xi !P' : \mathtt{Proc}\ A$. This must have been derived by *(Proc Bang)* from $\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A$. Since $A \oplus A = A$ the *(Proc Par)* rule let us conclude $\Gamma \rhd_\Xi !P' \mid P' : \mathtt{Proc}\ A$.

For (2), assume $\Gamma \rhd_\Xi !P' \mid P' : \mathtt{Proc}\ A$. This must have been derived by *(Proc Par)* from $\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A'$ and $\Gamma \rhd_\Xi !P' : \mathtt{Proc}\ A''$. The latter must have been derived by the *(Proc Bang)* rule and this guarantees that $A' = A''$. Since $A' \oplus A'' = A$ we can easily deduce that $A = A' = A''$. Thus the *(Proc Bang)* rule let us conclude $\Gamma \rhd_\Xi !P' : \mathtt{Proc}\ A$.

(Struct Res Dead) Then $P = (\boldsymbol{\nu}\, x{:}M)\mathbf{0}$ and $Q = \mathbf{0}$.

For (1), assume $\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)\mathbf{0} : \mathtt{Proc}\ [\,]$. We can infer this by *(res ...)* from $\Gamma, x{:}M \rhd_{\Xi,(x)} \mathbf{0} : \mathtt{Proc}\ [\,]$. But $x \notin \mathit{fn}(\mathbf{0})$ and so, by Lemma 5, $\Gamma \rhd_\Xi \mathbf{0} : \mathtt{Proc}\ [\,]$.

For (2), assume $\Gamma \rhd_\Xi \mathbf{0} : \mathtt{Proc}\ [\,]$. We may assume that the name x does not occur in $\mathrm{dom}(\Gamma)$, specifically $\Gamma, x{:}M \rhd_{\Xi,(x)} \diamond$. Thus by the Lemma 4 $\Gamma, x{:}M \rhd_{\Xi,(x)} \mathbf{0} : \mathtt{Proc}\ [\,]$ and by *(res ...)* $\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)\mathbf{0} : \mathtt{Proc}\ [\,]$.

(Struct Res Res) Then $P = (\boldsymbol{\nu}\, x{:}M)(\boldsymbol{\nu}\, y{:}M')P'$ and $Q = (\boldsymbol{\nu}\, y{:}M')(\boldsymbol{\nu}\, x{:}M)P'$. For (1), assume $\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)(\boldsymbol{\nu}\, y{:}M')P' : \mathtt{Proc}\ A$. This must have been derived using *(res ...)* twice from $\Gamma, x{:}M, y{:}M' \rhd_{\Xi,(x),(y)} P : \mathtt{Proc}\ A$, with the condition that x and y do not appear in A. Since $x \notin fn(M')$ and $y \notin fn(M)$ by Lemma 6 we deduce $\Gamma, y{:}M', x{:}M \rhd_{\Xi,(y),(x)} P : \mathtt{Proc}\ A$ and by *(res ...)* twice $\Gamma \rhd_\Xi (\boldsymbol{\nu}\, y{:}M')(\boldsymbol{\nu}\, x{:}M)P' : \mathtt{Proc}\ A$. Hence $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$. Part (2) is symmetric.

(Struct Res Par) Then $P = (\boldsymbol{\nu}\, x{:}M)(P' \mid Q')$ and $Q = P' \mid (\boldsymbol{\nu}\, x{:}M)Q'$.

For (1), assume $\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)(P' \mid Q') : \mathtt{Proc}\ A$. This must have been deduced by the following derivation:

$$\dfrac{\dfrac{\Gamma, x{:}M \rhd_{\Xi,(x)} P' : \mathtt{Proc}\ A' \qquad \Gamma, x{:}M \rhd_{\Xi,(x)} Q' : \mathtt{Proc}\ A''}{\Gamma, x{:}M \rhd_{\Xi,(x)} P' \mid Q' : \mathtt{Proc}\ A}}{\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)(P' \mid Q') : \mathtt{Proc}\ A}$$

with the conditions $A = A' \oplus A''$ and $x \notin fn(A)$: the second condition is verified both in the case of $M = \mathtt{Id}\ A$, since seal names don't figure in the type, and of $M = \mathtt{Ch}\ V$, by the side condition on *(Proc Res Ch)*. Since $x \notin fn(A)$ and $A'' \subseteq A$, $x \notin fn(A'')$; beside $x \notin fn(P)$ and by Lemma 5 $\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A'$. So we can construct the following derivation:

$$\dfrac{\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A' \qquad \dfrac{\Gamma, x{:}M \rhd_{\Xi,(x)} Q' : \mathtt{Proc}\ A''}{\Gamma \rhd (\boldsymbol{\nu}\, x{:}M)Q' : \mathtt{Proc}\ A''}}{\Gamma \rhd_\Xi P' \mid (\boldsymbol{\nu}\, x{:}M)Q' : \mathtt{Proc}\ A}$$

Hence $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$.

For (2), assume $\Gamma \rhd_\Xi P' \mid (\boldsymbol{\nu}\, x{:}M)Q' : \mathtt{Proc}\ A$. This must have been deduced by the following derivation:

$$\dfrac{\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A' \qquad \dfrac{\Gamma, x{:}M \rhd_{\Xi,(x)} Q' : \mathtt{Proc}\ A''}{\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)Q' : \mathtt{Proc}\ A''}}{\Gamma \rhd_\Xi P' \mid (\boldsymbol{\nu}\, x{:}M)Q' : \mathtt{Proc}\ A}$$

But $x \notin fn(P)$ and $\Gamma, x{:}M \rhd_{\Xi,(x)} \diamond$, so by Lemma 4 we infer $\Gamma, x{:}M \rhd_{\Xi,(x)} P' : \mathtt{Proc}\ A'$. Thus we can derive:

$$\dfrac{\dfrac{\Gamma, x{:}M \rhd_{\Xi,(x)} P' : \mathtt{Proc}\ A' \qquad \Gamma, x{:}M \rhd_{\Xi,(x)} Q' : \mathtt{Proc}\ A''}{\Gamma, x{:}M \rhd_{\Xi,(x)} P' \mid Q' : \mathtt{Proc}\ A}}{\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)(P' \mid Q') : \mathtt{Proc}\ A}$$

Hence $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$.

(Struct Res Agent) Then $P = (\boldsymbol{\nu}\, x{:}M)y[P']$ and $Q = y[(\boldsymbol{\nu}\, x{:}M)P']$.

For (1), assume $\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)y[P'] : \mathtt{Proc}\ [\,]$. This must have been derived by:

$$\dfrac{\dfrac{\Gamma, x{:}M \rhd_{\Xi,(x)} P' : \mathtt{Proc}\ A_1 \qquad \Gamma, x{:}M \rhd_{\Xi,(x)} y : \mathtt{Id}\ A}{\Gamma, x{:}M \rhd_{\Xi,(x)} y[P'] : \mathtt{Proc}\ [\,]}}{\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)y[P'] : \mathtt{Proc}\ [\,]}$$

where $\mathtt{Proc}\ A_1 \leq \mathtt{Proc}\ A$. $\Gamma, x{:}M \rhd_{\Xi,(x)} y : \mathtt{Id}\ A$ implies $\Gamma \rhd_{\Xi} y : \mathtt{Id}\ A$ since $x \neq y$ (this condition is really important only if $M = \mathtt{Id}\ A'$, otherwise is trivially verified). If $M = \mathtt{Ch}\ V$ for some V then $\Gamma, x{:}\mathtt{Ch}\ V \rhd_{\Xi,x} y : \mathtt{Id}\ A$ also implies $x \notin fn(A)$ since the name x is not known when y is declared. But $\mathtt{Proc}\ A_1 \leq \mathtt{Proc}\ A$ and we have $x \notin fn(A_1)$ too. So we can construct the following derivation (independently from the type M):

$$\dfrac{\dfrac{\Gamma, x{:}M \rhd_{\Xi,(x)} P' : \mathtt{Proc}\ A_1 \qquad x \notin fn(A_1)}{\Gamma \rhd_{\Xi} (\boldsymbol{\nu}\, x{:}M)P' : \mathtt{Proc}\ A_1} \qquad \Gamma \rhd_{\Xi} y : \mathtt{Id}\ A}{\Gamma \rhd_{\Xi} y[(\boldsymbol{\nu}\, x{:}M)P'] : \mathtt{Proc}\ []}$$

Hence $\Gamma \rhd_{\Xi} Q : \mathtt{Proc}\ []$.

For (2), assume $\Gamma \rhd_{\Xi} y[(\boldsymbol{\nu}\, x{:}M)P'] : \mathtt{Proc}\ []$. This must have been derived by:

$$\dfrac{\dfrac{\Gamma, x{:}M \rhd_{\Xi,(x)} P' : \mathtt{Proc}\ A_1}{\Gamma \rhd_{\Xi} (\boldsymbol{\nu}\, x{:}M)P' : \mathtt{Proc}\ A_1}\, x \notin fn(A_1) \qquad \Gamma \rhd_{\Xi} y : \mathtt{Id}\ A}{\Gamma \rhd_{\Xi} y[(\boldsymbol{\nu}\, x{:}M)P'] : \mathtt{Proc}\ []}$$

where $\mathtt{Proc}\ A_1 \leq \mathtt{Proc}\ A$. By Lemma 4 we have $\Gamma, x{:}M \rhd_{\Xi,(x)} y : \mathtt{Id}\ A$ and so we can derive:

$$\dfrac{\dfrac{\Gamma, x{:}M \rhd_{\Xi,(x)} P' : \mathtt{Proc}\ A_1 \qquad \Gamma, x{:}M \rhd_{\Xi,(x)} y : \mathtt{Id}\ A}{\Gamma, x{:}M \rhd_{\Xi,(x)} y[P'] : \mathtt{Proc}\ []}}{\Gamma \rhd_{\Xi} (\boldsymbol{\nu}\, x{:}M)y[P'] : \mathtt{Proc}\ []}$$

Hence $\Gamma \rhd_{\Xi} P : \mathtt{Proc}\ []$.

Remark that this case doesn't work with the weak (erasing) version of *(Proc Res Ch)*.

(Struct Cong New) Then $P = (\boldsymbol{\nu}\, x{:}M)P'$ and $Q = (\boldsymbol{\nu}\, x{:}M)Q'$. Assume $\Gamma \rhd_{\Xi} (\boldsymbol{\nu}\, x{:}M)P : \mathtt{Proc}\ A$. This must have been derived by *(res ...)* from $\Gamma, x{:}M \rhd_{\Xi,(x)} P : \mathtt{Proc}\ A$ with the condition that $\Gamma, x{:}M$ is a well-formed environment. By the induction hypothesis $\Gamma, x{:}M \rhd_{\Xi,(x)} Q : \mathtt{Proc}\ A$ and by *(res ...)* $\Gamma \rhd_{\Xi} (\boldsymbol{\nu}\, x{:}M)Q' : \mathtt{Proc}\ A$. Hence $\Gamma \rhd_{\Xi} Q : \mathtt{Proc}\ A$. Part (2) is symmetric.

(Struct Cong Par) Then $P = P' \mid R$ and $Q = P'' \mid R$. Assume $\Gamma \rhd_{\Xi} P : \mathtt{Proc}\ A$. This must have been derived by *(Proc Par)* from $\Gamma \rhd_{\Xi} P' : \mathtt{Proc}\ A'$, $\Gamma \rhd_{\Xi} R : \mathtt{Proc}\ A''$ with $A = A' \oplus A''$. By the induction hypothesis $\Gamma \rhd_{\Xi} P'' : \mathtt{Proc}\ A'$ and by *(Proc Par)* $\Gamma \rhd_{\Xi} P'' \mid R : \mathtt{Proc}\ A' \oplus A''$. Hence $\Gamma \rhd_{\Xi} Q : \mathtt{Proc}\ A$. Part (2) is symmetric.

(Struct Cong Agent) Then $P = x[P']$ and $Q = x[P'']$. Assume $\Gamma \rhd_{\Xi} P : \mathtt{Proc}\ []$. This must have been derived by *(Proc Seal)* from the hypothesis $\Gamma \rhd_{\Xi} P' : \mathtt{Proc}\ A_1$ and $\Gamma \rhd_{\Xi} x : \mathtt{Id}\ A$, where $\mathtt{Proc}\ A_1 \leq \mathtt{Proc}\ A$. By the induction hypothesis $\Gamma \rhd_{\Xi} P'' : \mathtt{Proc}\ A_1$ and by *(Proc Seal)* we can conlude $\Gamma \rhd_{\Xi} x[P''] : \mathtt{Proc}\ []$. Hence $\Gamma \rhd_{\Xi} Q : \mathtt{Proc}\ []$. Part (2) is symmetric.

(Struct Cong Action) Then $P = \alpha.P'$ and $Q = \alpha.Q'$. By case on α.

$x^*(y{:}M)$ Assume $\Gamma \rhd_\Xi x^*(y{:}M).P' : \texttt{Proc}\ A$. This must have been derived by *(input local)* from $\Gamma \rhd_\Xi x : \texttt{Ch}\ M$ and $\Gamma, y{:}\texttt{Ch}\ M \rhd_\Xi P' : \texttt{Proc}\ A$, with the condition $x \in \mathrm{dom}(\Xi)$. By the induction hypothesis we have $\Gamma, y{:}\texttt{Ch}\ M \rhd_\Xi Q' : \texttt{Proc}\ A$; hence by *(input local)* $\Gamma \rhd_\Xi x^*(y{:}M).Q' : \texttt{Proc}\ A$.

$\overline{x}^*(y)$ Assume $\Gamma \rhd_\Xi \overline{x}^*(y).P' : \texttt{Proc}\ A$. This must have been derived by *(output local)* from $\Gamma \rhd_\Xi x : \texttt{Ch}\ M$, and $\Gamma \rhd_\Xi y : M$, and $\Gamma \rhd_\Xi P' : \texttt{Proc}\ A$. By the induction hypothesis we have $\Gamma \rhd_\Xi Q' : \texttt{Proc}\ A$; hence by *(output local)* $\Gamma \rhd_\Xi \overline{x}^*(y).Q' : \texttt{Proc}\ A$.

$x^\uparrow(y{:}M)$ Assume $\Gamma \rhd_\Xi x^\uparrow(y{:}M).P' : \texttt{Proc}\ A$. This must have been derived by *(input up)* from $\Gamma \rhd_\Xi x : \texttt{Ch}\ M$, and $\Gamma, y{:}\texttt{Ch}\ M \rhd_\Xi P' : \texttt{Proc}\ (A \setminus [x : M])$, with the condition $x \in \mathrm{dom}(\Xi)$. By the induction hypothesis we have $\Gamma, y{:}\texttt{Ch}\ M \rhd_\Xi Q' : \texttt{Proc}\ (A \setminus [x : M])$ and by *(input up)* $\Gamma \rhd_\Xi x^*(y{:}M).Q' : \texttt{Proc}\ A$.

$\overline{x}^\uparrow(y)$ Trivial, in the style of $\alpha = \overline{x}^*(y)$.

$x^z(y{:}M)$ Assume $\Gamma \rhd_\Xi x^z(y{:}M).P' : \texttt{Proc}\ A$. This must have been derived by *(input down)* from $\Gamma \rhd_\Xi z : \texttt{Id}\ A'$, and $\Gamma \rhd_\Xi x : \texttt{Ch}\ M$, and $\Gamma, y{:}\texttt{Ch}\ M \rhd_\Xi P' : \texttt{Proc}\ A$, with the condition $x \in \mathrm{dom}(\Xi)$. By the induction hypothesis we have $\Gamma, y{:}\texttt{Ch}\ M \rhd_\Xi Q' : \texttt{Proc}\ A$; hence by *(input down)* $\Gamma \rhd_\Xi x^*(y{:}M).Q' : \texttt{Proc}\ A$.

$\overline{x}^z(y)$ Assume $\Gamma \rhd_\Xi \overline{x}^z(y).P' : \texttt{Proc}\ A$. This must have been derived by *(output down)* from $\Gamma \rhd_\Xi z : \texttt{Id}\ A'$, and $\Gamma \rhd_\Xi y : M$, $\Gamma \rhd_\Xi P' : \texttt{Proc}\ A$ where $(x{:}M) \in A'$. By the induction hypothesis we have $\Gamma \rhd_\Xi Q' : \texttt{Proc}\ A$; hence by *(output down)* $\Gamma \rhd_\Xi \overline{x}^z(y).Q' : \texttt{Proc}\ A$

$x^*\{\!|y|\!\}$ Assume $\Gamma \rhd_\Xi x^*\{\!|y|\!\}.P' : \texttt{Proc}\ A$. This must have been derived by *(Rcv Local)* from $\Gamma \rhd_\Xi x : \texttt{Ch}\ A'$, and $\Gamma \rhd_\Xi y : \texttt{Id}\ A'$, and $\Gamma \rhd_\Xi P' : \texttt{Proc}\ A$, with the condition $x \in \mathrm{dom}(\Xi)$. By the induction hypothesis we have $\Gamma \rhd_\Xi Q' : \texttt{Proc}\ A$; hence by *(Rcv Local)* $\Gamma \rhd_\Xi x^*\{\!|y|\!\}.Q' : \texttt{Proc}\ A$.

$\overline{x}^*\{\!|y|\!\}$ Trivial, in the style of $\alpha = x^*\{\!|y|\!\}$.

$x^\uparrow\{\!|y|\!\}$ Assume $\Gamma \rhd_\Xi x^\uparrow\{\!|y|\!\}.P' : \texttt{Proc}\ A$. This must have been derived by *(Rcv Up)* from $\Gamma \rhd_\Xi x : \texttt{Ch}\ A'$, and $\Gamma \rhd_\Xi y : \texttt{Id}\ A'$, and $\Gamma \rhd_\Xi P' : \texttt{Proc}\ (A \setminus [x : A'])$, with the condition $x \in \mathrm{dom}(\Xi)$. By induction hypothesis we have $\Gamma \rhd_\Xi Q' : \texttt{Proc}\ (A \setminus [x : A'])$; hence by *(Rcv Up)* $\Gamma \rhd_\Xi x^\uparrow\{\!|y|\!\}.Q' : \texttt{Proc}\ A$.

$\overline{x}^\uparrow\{\!|y|\!\}$ Trivial, in the style of $\alpha = x^*\{\!|y|\!\}$.

$x^z\{\!|y|\!\}$ Trivial, in the style of $\alpha = x^*\{\!|y|\!\}$.

$\overline{x}^z\{\!|y|\!\}$ Assume $\Gamma \rhd_\Xi \overline{x}^z\{\!|y|\!\}.P' : \texttt{Proc}\ A$. This must have been derived by *(Snd Down)* from $\Gamma \rhd_\Xi z : \texttt{Id}\ A_1$, and $\Gamma \rhd_\Xi y : \texttt{Id}\ A$, and $\Gamma \rhd_\Xi P' : \texttt{Proc}\ A_2$ where $(x{:}A) \in A_1$. By the induction hypothesis we have $\Gamma \rhd_\Xi Q' : \texttt{Proc}\ A_2$; hence by *(Snd Down)* $\Gamma \rhd_\Xi \overline{x}^z\{\!|y|\!\}.Q' : \texttt{Proc}\ A$.

Part(2) is symmetric. $\square$

Now we prove a weak form of the standard Substitution Lemma. We will consider separately substitution of Channel Names and of Seal Names, since the former involves a substitution on the typing environment and some more complex machinery to deal with labels.

First of all some technical Lemmas to explain why we are using Ξ.

Lemma 9 *If $\Gamma', x : \mathtt{Ch}\ V, \Gamma'' \rhd_\Xi \diamond$ and $x \notin dom(\Xi)$ then $x \notin fn(\Gamma'')$.*

Proof By induction on the number of names defined in Γ'' (denoted $\mathtt{card}(\Gamma'')$).

- If $\mathtt{card}(\Gamma'') = \mathtt{0}$ then $x \notin fn(\Gamma'')$ is trivially verified.
- If $\mathtt{card}(\Gamma'') = \mathtt{n} + \mathtt{1}$ then let $y{:}M$ be the last element added to Γ'': we will write Γ'' as $\Gamma''', y{:}M$. The hypothesis guarantees that $\Gamma', x : \mathtt{Ch}\ V, \Gamma''', y{:}M \rhd_{\Xi,(y)} \diamond$. Both rules *(Env Add)* require that M must be well-formed: so we have $\Gamma', x : \mathtt{Ch}\ V, \Gamma''' \rhd_\Xi M$.

  By contradiction, assume that $x \in fn(M)$: by induction on the structure of M we can prove that M can not be well-formed in an environment where $x \notin \mathrm{dom}(\Xi)$. We limit ourself to sketch the idea, since in the whole induction there is only a lot of technical details to deal with the complex structure of M. If x appears in M then it can appear only as a label in an agent somewhere in the structure of M. But the agent type can not be well-formed, since the *(Type Interface)* rule requires that all labels x_i appear in $\mathrm{dom}(\Xi)$, and here we know that $x \notin \mathrm{dom}(\Xi)$.

$\square$

Corollary 2 *If $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_\Xi \diamond$ and $x \notin dom(\Xi)$ then $\Gamma', \Gamma'' \rhd_\Xi \diamond$.*

Lemma 10 *If $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_\Xi P : \mathtt{Proc}\ A$ and $x \notin dom(\Xi)$ then $x \notin fn(A)$.*

Proof By Lemma 3, if $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_\Xi P : \mathtt{Proc}\ A$ then $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_\Xi A$. From this we prove that $x \notin fn(A)$ by induction on the structure of A. The rule *(Type Interface)* guarantees that all labels that appear in A appears also in $\mathrm{dom}(\Xi)$. Since $x \notin \mathrm{dom}(\Xi)$, x can not appear in the labels of A. It remains to check that it does not appear in the type of some label. But all the names that appear in a type must appear as label in an agent type, possibly inside the structure of A: so the same reasonment can be applied for arbitrarly nested agent type. Thus $x \notin fn(A)$. $\square$

Lemma 11 *If $\Gamma', x{:}\mathtt{Ch}\ V', \Gamma'' \rhd_\Xi P : \mathtt{Proc}\ A$ and $x \notin dom(\Xi)$ then $x \notin fn(V)$, $\forall (\boldsymbol{\nu}\, v{:}V) \in P$.*

Proof During type-checking, the rules *(Proc Res Agent)* and *(Proc Res Ch)* are used to check every occurence of $(\boldsymbol{\nu}\, v{:}V)$. This implies that all $v{:}V$ met are added to the typing environment and all them will appear after the $x{:}\mathtt{Ch}\ V'$. Thus all v will be different names from x and x will never be added to Ξ. Since at the end of type-checking the rule *(dead)* requires to control if the environment constructed is well-formed, the lemma follows as a conseguence of Lemma 9. $\square$

Lemma 12 (Substitution Lemma: channel names) *If* $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} P : \mathtt{Proc}\ A$, $x \notin dom(\Xi', \Xi'')$ *and* $\Gamma' \triangleright_{\Xi'} y : \mathtt{Ch}\ V$ *then* $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} P\{^y/_x\} : \mathtt{Proc}\ A$.

Proof We proceed by induction on the structure of P.

0 Assume $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} \mathbf{0} : \mathtt{Proc}\ []$. Then $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} \diamond$ and by Corollary 2 $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} \diamond$. The *(dead)* rule permits to conclude $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} \mathbf{0} : \mathtt{Proc}\ []$.

$P' \mid P''$ Assume $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} P : \mathtt{Proc}\ A$. This must have been derived by *(Proc Par)* from $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} P' : \mathtt{Proc}\ A_1$ and $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} P'' : \mathtt{Proc}\ A_2$, where $A = A' \oplus A''$. By the induction hypothesis we have $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} P'\{^y/_x\} : \mathtt{Proc}\ A_1$ and $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} P''\{^y/_x\} : \mathtt{Proc}\ A_2$. By definition of substitution $P_1\{^y/_x\} \mid P_2\{^y/_x\} = (P_1 \mid P_2)\{^y/_x\}$. Hence by *(Proc Par)* we can conclude $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} P\{^y/_x\} : \mathtt{Proc}\ A$.

$u[P']$ Assume $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} u[P'] : \mathtt{Proc}\ A$. Remark that $A = []$. This must have been derived by *(Proc Seal)* from $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} u : \mathtt{Id}\ A_1$ and $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} P' : \mathtt{Proc}\ A_2$, with $\mathtt{Proc}\ A_2 \leq \mathtt{Proc}\ A_1$. Since x and u have different types they are different names: by Corollary 2 we obtain $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} \diamond$ and then $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} u : \mathtt{Id}\ A_1$. By the induction hypothesis we have $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} P'\{^y/_x\} : \mathtt{Proc}\ A_2$. Hence by *(Proc Seal)* we conclude $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} u[P'\{^y/_x\}] : \mathtt{Proc}\ A$. Since $x \neq u$, by definition of substitution, $u[P'\{^y/_x\}] = u[P']\{^y/_x\}$ and $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} u[P']\{^y/_x\} : \mathtt{Proc}\ A$, as desired.

$(\boldsymbol{\nu}\, u{:}M)P'$ Assume $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} (\boldsymbol{\nu}\, u{:}M)P' : \mathtt{Proc}\ A$. This must have been derived by *(res ...)* from $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'', u{:}M \triangleright_{\Xi',\Xi'',(u)} P' : \mathtt{Proc}\ A$. By the induction hypothesis we have $\Gamma', \Gamma'', u{:}M \triangleright_{\Xi',\Xi'',(u)} P'\{^y/_x\} : \mathtt{Proc}\ A$. By *(res ...)*, $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} (\boldsymbol{\nu}\, u{:}M)(P'\{^y/_x\}) : \mathtt{Proc}\ A$. The well-formed condition on $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'', u{:}M$ assures that $x \neq u$, Lemma 11 guarantees that $x \notin \mathit{fn}(M)$ since $x \notin \mathrm{dom}(\Xi', \Xi'')$. So $(\boldsymbol{\nu}\, u{:}M)(P'\{^y/_x\}) = ((\boldsymbol{\nu}\, u{:}M)P')\{^y/_x\}$. Hence we obtain $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} ((\boldsymbol{\nu}\, u{:}M)P')\{^y/_x\} : \mathtt{Proc}\ A$.

$\alpha.P'$ By cases on α. We detail only the local operations and the input from an up-channel. What is interessant to remark is the use of Ξ to guarantee that in the input and receive operations the name we are going to substitute is always different from the subject name.

$u^*(v{:}M)$ Assume $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} u^*(v{:}M).P' : \mathtt{Proc}\ A$. This must have been derived by *(input local)* from $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} u : \mathtt{Ch}\ M$ and $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'', v{:}M \triangleright_{\Xi',\Xi''} P' : \mathtt{Proc}\ A$, with the condition $u \in \mathrm{dom}(\Xi', \Xi'')$. By the induction hypothesis we obtain $\Gamma', \Gamma'', v{:}M \triangleright_{\Xi',\Xi''} P'\{^y/_x\} : \mathtt{Proc}\ A$. Since v can be α-converted, it is trivially different from x. To prove that $x \neq u$ is sufficient remark that $x \notin \mathrm{dom}(\Xi)$ while $u \in \mathrm{dom}(\Xi)$. By Corollary 2 we deduce $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} u : \mathtt{Ch}\ M$. Applying *(input local)*, we have $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} u^*(v{:}M).(P'\{^y/_x\}) : \mathtt{Proc}\ A$. By definition of substitution we finally get $\Gamma', \Gamma'' \triangleright_{\Xi',\Xi''} (u^*(v{:}M).P')\{^y/_x\} : \mathtt{Proc}\ A$.

$\overline{u}^*(v)$ Assume $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} \overline{u}^*(v).P' : \mathtt{Proc}\ A$. This must have been derived by *(output local)* from $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''} u : \mathtt{Ch}\ M$, and $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \triangleright_{\Xi',\Xi''}$

$v : M$, and $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} P' : \mathtt{Proc}\ A$. By the induction hypothesis we obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} P'\{^y/_x\} : \mathtt{Proc}\ A$. Now we have to deal with all the possible interactions that can take place between x and $\overline{u}^*(v)$.

$x = u \wedge x \neq v$ Then $M = V$: by hypothesis we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} y : \mathtt{Ch}\ M$ and since x and v are different we can derive $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} v : M$. By *(input local)* we can conclude $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} \overline{y}^*(v).(P'\{^y/_x\}) : \mathtt{Proc}\ A$, as desired.

$x \neq u \wedge x = v$ Then $M = \mathtt{Ch}\ V$: by hypothesis we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} y : M$ and since x and u are different we can derive $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u : \mathtt{Ch}\ M$. By *(input local)* we can conclude $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} \overline{u}^*(y).(P'\{^y/_x\}) : \mathtt{Proc}\ A$, as desired.

$x \neq u \wedge x \neq v$ Since x is different from both u and v we can derive $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u : \mathtt{Ch}\ M$ and $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} v : M$. For the same reason $\overline{u}^*(v).(P'\{^y/_x\}) = (\overline{u}^*(v).P')\{^y/_x\}$. By *(input local)* we can conlude $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} (\overline{u}^*(v).P')\{^y/_x\} : \mathtt{Proc}\ A$, as desired.

We remark that the case $x = u = v$ is impossible for a well-typed term because the type of u should be the same of the type of v: in our type system a V such that $\mathtt{Ch}\ V = V$ does not exist.

$u^*\{v\}$ Assume $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} u^*\{v\}.P' : \mathtt{Proc}\ A$. This must have been derived by *(Rcv Local)* from $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} u : \mathtt{Ch}\ A'$, $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} v : \mathtt{Id}\ A'$, and $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} P' : \mathtt{Proc}\ A$, with $u \in \mathrm{dom}(\Xi', \Xi'')$. By the induction hypothesis we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} P'\{^y/_x\} : \mathtt{Proc}\ A$. Since v and x have different types, they are different names; x is also different from u because $u \in \mathrm{dom}(\Xi', \Xi^second)$ and $x \notin \mathrm{dom}(\Xi', \Xi'')$. So by definition of substitution we have $u^*\{v\}.(P'\{^y/_x\}) = (u^*\{v\}.P')\{^y/_x\}$ and by *(mob local rec)* we conclude $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} (u^*\{v\}.P')\{^y/_x\} : \mathtt{Proc}\ A$, as desired.

$\overline{u}^*\{v\}$ Assume $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} \overline{u}^*\{v\}.P' : \mathtt{Proc}\ A$. This must have been derived by *(Snd Local)* from $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} u : \mathtt{Ch}\ A'$, $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} v : \mathtt{Id}\ A'$, and $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} P' : \mathtt{Proc}\ A$. By induction hypothesis we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} P'\{^y/_x\} : \mathtt{Proc}\ A$. Since v and x have different types, they are different names but we have still to consider separately the possible interactions between x and u.

$x = u$ Then $V = A'$. By hypothesis we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} y : \mathtt{Ch}\ A'$ and, since x is different from v, $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} v : \mathtt{Id}\ A'$. By *(mob local snd)* we conclude $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} \overline{y}^*\{v\}.(P'\{^y/_x\}) : \mathtt{Proc}\ A$, as desired.

$x \neq u$ We can derive $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u : \mathtt{Ch}\ A'$ and $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} v : \mathtt{Id}\ A'$. By definition of substitution we have $\overline{u}^*\{v\}.(P'\{^y/_x\}) = (\overline{u}^*\{v\}.P')\{^y/_x\}$. By *(Snd Local)* we conclude $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} (\overline{u}^*\{v\}.P')\{^y/_x\} : \mathtt{Proc}\ A$, as desired.

$u^\uparrow(v{:}M)$ Assume $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} u^\uparrow(v{:}M).P' : \mathtt{Proc}\ A$. This must have been derived by *(input up)* from $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'' \rhd_{\Xi',\Xi''} u : \mathtt{Ch}\ M$ and $\Gamma', x{:}\mathtt{Ch}\ V, \Gamma'', v{:}M \rhd_{\Xi',\Xi''} P' : \mathtt{Proc}\ A_1$, with the condition $u \in \mathrm{dom}(\Xi', \Xi'')$ and $A = A_1 \oplus [u{:}M]$. By the induction hypothesis we obtain $\Gamma', \Gamma'', v{:}M \rhd_{\Xi',\Xi''} P'\{{}^{y}/_{x}\} : \texttt{Proc}\ A_1$. Since v can be α-converted, it is trivially different from x. To prove that $x \neq u$ is sufficient remark that $u \in$ $dom(\Xi', \Xi'')$ and $x \notin \mathrm{dom}(\Xi', \Xi'')$. By Corollary 2 we deduce $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u : \texttt{Ch}\ M$. Applying *(input up)*, we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''}$ $u^*(v{:}M).(P'\{{}^{y}/_{x}\}) : \texttt{Proc}\ (A_1 \oplus [u{:}M])$. By definition of substitution and of A, we finally get $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} (u^*(v{:}M).P')\{{}^{y}/_{x}\} : \texttt{Proc}\ A$, as desired.

□

Corollary 3 *If* $\Gamma, x{:}\texttt{Ch}\ V \rhd_{\Xi} P : \texttt{Proc}\ A$ *and* $\Gamma \rhd_{\Xi} y : \texttt{Ch}\ V$ *then* $\Gamma \rhd_{\Xi} P\{{}^{y}/_{x}\} : \texttt{Proc}\ A$.

Proof From Lemma 12, taking Γ'' empty. □

Lemma 13 (Substitution Lemma: Agent Names) *If* $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} P : \texttt{Proc}\ A$ *and* $\Gamma' \rhd_{\Xi'} y : \texttt{Id}\ A'$ *then* $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} P\{{}^{y}/_{x}\} : \texttt{Proc}\ A$.

Proof We proceed by induction on the structure of P.

0 Assume $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} \mathbf{0} : \texttt{Proc}\ [\,]$. Then $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} \diamond$ and trivially $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} \diamond$. The *(dead)* rule permits to conclude $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} \mathbf{0} : \texttt{Proc}\ [\,]$.

$P' \mid P''$ Assume $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} P : \texttt{Proc}\ A$. This must have been derived by *(Proc Par)* from $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} P' : \texttt{Proc}\ A_1$ and $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} P'' : \texttt{Proc}\ A_2$, where $A = A_1 \oplus A_2$. By the induction hypothesis we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} P'\{{}^{y}/_{x}\}$: $\texttt{Proc}\ A_1$ and $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} P''\{{}^{y}/_{x}\} : \texttt{Proc}\ A_2$. Hence by *(Proc Par)* and by definition of substitution, we can conclude $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} P\{{}^{y}/_{x}\} : \texttt{Proc}\ A$.

$u[P']$ Assume $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} u[P'] : \texttt{Proc}\ A$. A must be exactly $[\,]$. This must have been derived from $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} u : \texttt{Id}\ A_1$ and $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} P'$: $\texttt{Proc}\ A_2$ where $\texttt{Proc}\ A_2 \leq \texttt{Proc}\ A_1$. By the induction hypothesis we can derive $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} P'\{{}^{y}/_{x}\} : \texttt{Proc}\ A_2$. There are two cases:

- $x \neq u$ We can safely derive $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u : \texttt{Id}\ A_1$, and apply the *(Proc Seal)* rule to get $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u[P'\{{}^{y}/_{x}\}] : \texttt{Proc}\ [\,]$. By definitions of substitution we conclude $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u[P']\{{}^{y}/_{x}\} : \texttt{Proc}\ [\,]$, as desired.
- $x = u$ Then $A' = A_1$. We have $\Gamma' \rhd_{\Xi'} y : \texttt{Id}\ A'$ and by repeated applications of Lemma 4 we can obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} y : \texttt{Id}\ A'$. By applying the rule *(Proc Seal)* we obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} y[P'\{{}^{y}/_{x}\}] : \texttt{Proc}\ [\,]$, and this is exactly what we were looking for, since $x = u$ implies $u[P']\{{}^{y}/_{x}\} = y[P'\{{}^{y}/_{x}\}]$.

$(\boldsymbol{\nu}\, u{:}M)P$ Assume $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} (\boldsymbol{\nu}\, u{:}M)P' : \texttt{Proc}\ A$. This must have been derived by *(res ...)* from $\Gamma', x{:}\texttt{Id}\ A', \Gamma'', u{:}M \rhd_{\Xi',\Xi'',(u)} P' : \texttt{Proc}\ A$. By the induction hypothesis we have $\Gamma', \Gamma'', u{:}M \rhd_{\Xi',\Xi'',(u)} P'\{{}^{y}/_{x}\} : \texttt{Proc}\ A$. By *(res ...)*, $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''}$ $(\boldsymbol{\nu}\, u{:}M)(P'\{{}^{y}/_{x}\})$: $\texttt{Proc}\ A$. The definition of Γ assure that in $\Gamma', x{:}\texttt{Id}\ A', \Gamma'', u{:}M$ we have $x \neq u$, and so $(\boldsymbol{\nu}\, u{:}M)(P'\{{}^{y}/_{x}\}) = ((\boldsymbol{\nu}\, u{:}M)P')\{{}^{y}/_{x}\}$. Hence we obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} ((\boldsymbol{\nu}\, u{:}M)P')\{{}^{y}/_{x}\} : \texttt{Proc}\ A$.

$\alpha.P$ By cases on α. We will point out only the case of a down communication —where the substitution can change the name of the seal we are writing on—, and receive from an up-channel —this operation adds a name to the interface of a process, and we will show that evil interaction on the new added name on the interface can not take place.

$u^z(v{:}M)$ Assume $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} u^z(v{:}M).P' : \mathtt{Proc}\ A$. This must have been derived by *(input down)* from $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} z : \mathtt{Id}\ A_2$, $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} u : \mathtt{Ch}\ M$ and $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'', v{:}M \rhd_{\Xi',\Xi''} P' : \mathtt{Proc}\ A$. By the induction hypothesis we have $\Gamma', \Gamma'', v{:}M \rhd_{\Xi',\Xi''} P'\{^y/_x\} : \mathtt{Proc}\ A$. Since z and u have different types they must be different names, and this let us derive $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u : \mathtt{Ch}\ M$. To get in account all the possible interactions between the substitution $\{^y/_x\}$ and α we need to consider the following two cases:

$x \neq z$ We can safely derive $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} z : \mathtt{Id}\ A_2$ and apply *(input down)* to get $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u^z(v{:}M).(P'\{^y/_x\}) : \mathtt{Proc}\ A$. In this case $u^z(v{:}M).(P'\{^y/_x\}) = (u^z(v{:}M).P')\{^y/_x\}$ and we obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} (u^z(v{:}M).P')\{^y/_x\} : \mathtt{Proc}\ A$, as desired.

$x = z$ Then $A' = A_2$. We have $\Gamma' \rhd_{\Xi'} y : \mathtt{Id}\ A'$, and by repeated applications of Lemma 4 we can obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} y : \mathtt{Id}\ A'$. Applying the rule *(input down)* we obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u^y(v{:}M).(P'\{^y/_x\}) : \mathtt{Proc}\ A$ and this is exactly what we were looking for since if $x = z$ then $(u^z(v{:}M).P')\{^y/_x\} = u^y(v{:}M).(P'\{^y/_x\})$.

$\overline{u}^z(v)$ Assume $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} \overline{u}^z(v).P' : \mathtt{Proc}\ A$. This must have been derived by *(output down)* from $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} z : \mathtt{Id}\ A_2$, $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} v : M$ and $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} P' : \mathtt{Proc}\ A$, with $(u{:}M) \in A_2$. By the induction hypothesis we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} P'\{^y/_x\} : \mathtt{Proc}\ A$. By hypothesis we know that $\Gamma' \rhd_{\Xi'} y : \mathtt{Id}\ A'$: by repeated applications of the Lemma 4 we get $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} y : \mathtt{Id}\ A'$. Considering the following cases we will get in account all the possible interactions between α and $\{^y/_x\}$:

$x \neq z \wedge x \neq v$ Since $x \neq v$ we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} v : M$ from $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} v : M$ and since $x \neq z$ we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} z : \mathtt{Id}\ A_2$ from $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} z : \mathtt{Id}\ A_2$. By *(output down)* we obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} \overline{u}^z(v).(P'\{^y/_x\}) : \mathtt{Proc}\ A$, as desired.

$x = z \wedge x \neq v$ Then $A' = A_2$ and $(u{:}M) \in A'$. Since $x \neq v$ we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} v : M$ from $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} v : M$. By *(output down)* we obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} \overline{u}^y(v).(P'\{^y/_x\}) : \mathtt{Proc}\ A$, as desired.

$x \neq z \wedge x = v$ Then $M = A'$ and $(u{:}A') \in A_2$. Since $x \neq z$ we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} z : \mathtt{Id}\ A_2$ from $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} z : \mathtt{Id}\ A_2$. By *(output down)* we obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} \overline{u}^z(y).(P'\{^y/_x\}) : \mathtt{Proc}\ A$, as desired.

We remark that the case $x = u = v$ is impossible for a well-typed term.

$u^{\uparrow}\{\!\{v\}\!\}$ Assume $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} u^{\uparrow}\{\!\{v\}\!\}.P' : \mathtt{Proc}\ A$. This must have been derived by *(mob up rec)* from $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} u : \mathtt{Ch}\ A_2$, $\Gamma', x{:}\mathtt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''}$

v : $\texttt{Id}\ A_2$ and $\Gamma', x{:}\texttt{Id}\ A', \Gamma'' \rhd_{\Xi',\Xi''} P'$: $\texttt{Proc}\ A_3$, where $A = A_3 \oplus [u{:}A_2]$. By the induction hypothesis we have $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} P'\{^y/_x\}$: $\texttt{Proc}\ A_3$. Since z and u have different types they must be different names, and this let us derive $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u$: $\texttt{Ch}\ M$. To get in account all the possible interactions between the substitution $\{^y/_x\}$ and α we need to consider the following two cases:

$x \neq v$ We can safely derive $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} v$: $\texttt{Id}\ A_2$ and apply *(Rcv Up)* to get $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u^{\uparrow}\{v\}.(P'\{^y/_x\})$: $\texttt{Proc}\ A$ as desired.

$x = v$ We have $\Gamma' \rhd_{\Xi'} y$: $\texttt{Id}\ A'$, with $A' = A_2$, and by repeated applications of Lemma 4 we can obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} y$: $\texttt{Id}\ A'$. By applying the rule *(Rcv Up)* we obtain $\Gamma', \Gamma'' \rhd_{\Xi',\Xi''} u^{\uparrow}\{y\}.(P'\{^y/_x\})$: $\texttt{Proc}\ A$, as desired.

□

Corollary 4 *If* $\Gamma, x{:}\texttt{Id}\ A' \rhd_{\Xi} P : \texttt{Proc}\ A$ *and* $\Gamma \rhd_{\Xi} y : \texttt{Id}\ A'$ *then* $\Gamma \rhd_{\Xi} P\{^y/_x\} : \texttt{Proc}\ A$.

Proof From Lemma 13, taking Γ'' empty. □

Corollary 5 (Substitution Lemma) *If* $\Gamma, x{:}M \rhd_{\Xi} P$: $\texttt{Proc}\ A$ *and* $\Gamma \rhd_{\Xi} y$: M *then* $\Gamma \rhd_{\Xi} P\{^y/_x\} : \texttt{Proc}\ A$.

Proof This is just syntactic sugar, and follows easily from Corollary 3 and Corollary 4. □

Theorem 1 (Subject Reduction) *If* $\Gamma \rhd_{\Xi} P : \texttt{Proc}\ A$ *and* $P \rightarrow Q$ *then* $\Gamma \rhd_{\Xi} Q : \texttt{Proc}\ A$.

Proof We proceed by induction on the depth of the derivation of $P \rightarrow Q$.

(write local) Then $P = x^*(u{:}M).P' \mid \overline{x}^*(v).P''$ and $Q = P'\{^v/_u\} \mid P''$. Assume $\Gamma \rhd_{\Xi} P$: $\texttt{Proc}\ A$. This must have been derived from (*Proc Par*), with $\Gamma \rhd_{\Xi} x^*(u{:}M).P'$: $\texttt{Proc}\ A'$ and $\Gamma \rhd_{\Xi} \overline{x}^*(v).P''$: $\texttt{Proc}\ A''$ where $A = A' \oplus A''$. The former must have been derived from (*input local*), with $\Gamma \rhd_{\Xi} x$: $\texttt{Ch}\ M$ and $\Gamma, u{:}M \rhd_{\Xi} P'$: $\texttt{Proc}\ A'$, and the conditions $x \in \mathrm{dom}(\Xi)$: remark that $u \notin \mathrm{dom}(\Xi)$. The latter must have been derived from (*output local*), with $\Gamma \rhd_{\Xi} x : \texttt{Ch}\ M$, $\Gamma \rhd_{\Xi} v : M$ and $\Gamma \rhd_{\Xi} P'' : \texttt{Proc}\ A''$, for some M'. This is depicted in fig. A.1. Γ behaves as a function, so $M = M'$ By application of Corollary 5 we get $\Gamma \rhd_{\Xi} P'\{^v/_u\} : \texttt{Proc}\ A'$. Hence, by (*Proc Par*), we deduce $\Gamma \rhd_{\Xi} P'\{^v/_u\} \mid P'' : \texttt{Proc}\ (A' \oplus A'')$, that is, $\Gamma \rhd_{\Xi} Q : \texttt{Proc}\ A$.

(write out) Then $P = x^y(u{:}M).P' \mid y[\overline{x}^{\uparrow}(v).P'' \mid R]$ and $Q = P'\{^v/_u\} \mid y[P'' \mid R]$. Assume $\Gamma \rhd_{\Xi} P$: $\texttt{Proc}\ A$. This must have been derived from (*Proc Par*), with $\Gamma \rhd_{\Xi} x^y(u{:}M).P'$: $\texttt{Proc}\ A$ and $\Gamma \rhd_{\Xi} y[\overline{x}^{\uparrow}(v).P'' \mid R]$: $\texttt{Proc}\ [\,]$. The former must have been derived from (*input down*) with $\Gamma \rhd_{\Xi} y$: $\texttt{Id}\ A_4$, $\Gamma \rhd_{\Xi} x$: $\texttt{Ch}\ M$, $\Gamma, u : M \rhd_{\Xi} P'$: $\texttt{Proc}\ A$, and $x \in \mathrm{dom}(\Xi)$: remark that $u \notin \mathrm{dom}(\Xi)$. The latter must have been derived from (*seal*) with $\Gamma \rhd_{\Xi} y$: $\texttt{Id}\ A_4$, $\Gamma \rhd \overline{x}^{\uparrow}(v).P'' \mid R : \texttt{Proc}\ A_3$ with $\texttt{Proc}\ A_3 \leq \texttt{Proc}\ A_4$, and this one by (*Proc Par*) from $\Gamma \rhd_{\Xi} \overline{x}^{\uparrow}(v).P''$: $\texttt{Proc}\ A_1$ and $\Gamma \rhd R : \texttt{Proc}\ A_2$, where $A_3 = A_1 \oplus A_2$. Lastly we point out that the hypothesis to derive $\Gamma \rhd_{\Xi} \overline{x}^{\uparrow}(v).P''$: $\texttt{Proc}\ A_1$ from (*output

$$\dfrac{\dfrac{\Gamma \rhd_\Xi x : \mathtt{Ch}\ M \qquad \Gamma, u{:}M \rhd_\Xi P' : \mathtt{Proc}\ A'}{\Gamma \rhd_\Xi x^*(u{:}M).P' : \mathtt{Proc}\ A'} \qquad \dfrac{\Gamma \rhd_\Xi x : \mathtt{Ch}\ M \qquad \Gamma \rhd_\Xi v : M \qquad \Gamma \rhd_\Xi P'' : \mathtt{Proc}\ A''}{\Gamma \rhd_\Xi \overline{x}^*(v).P'' : \mathtt{Proc}\ A''}}{\Gamma \rhd_\Xi x^*(u{:}M).P' \mid \overline{x}^*(v).P'' : \mathtt{Proc}\ A}$$

where $x \in \mathrm{dom}(\Xi)$,$u \notin \mathrm{dom}(\Xi)$ and $A = A' \oplus A''$.

Figure A.1: Derivation of $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$ in *(write_local)* *up*) are $\Gamma \rhd_\Xi x : \mathtt{Ch}\ M$, $\Gamma \rhd v : M'$ and $\Gamma \rhd_\Xi P'' : \mathtt{Proc}\ A_1$. This is depicted in fig. A.2. We can apply Corollary 5 with hypothesis $\Gamma, u{:}M \rhd_\Xi P' : \mathtt{Proc}\ A$ and $\Gamma \rhd_\Xi v : M$ to obtain $\Gamma \rhd_\Xi P'\{^v/_u\} : \mathtt{Proc}\ A$. Thus we can draw out the following derivation:

$$\dfrac{\Gamma \rhd_\Xi P'\{^v/_u\} : \mathtt{Proc}\ A \qquad \dfrac{\dfrac{\begin{array}{c}\Gamma \rhd_\Xi R : \mathtt{Proc}\ A_2 \\ \Gamma \rhd_\Xi P'' : \mathtt{Proc}\ A_1\end{array}}{\Gamma \rhd_\Xi P'' \mid R : \mathtt{Proc}\ A_3}\,(\mathit{Proc\ Par}) \qquad \Gamma \rhd_\Xi y : \mathtt{Id}\ A_4}{\Gamma \rhd_\Xi y[P'' \mid R] : \mathtt{Proc}\ [\,]}}{\Gamma \rhd_\Xi P'\{^v/_u\} \mid y[P'' \mid R] : \mathtt{Proc}\ A}$$

Hence $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$.

(write in) Then $P = \overline{x}^y(v)\,.\,P' \mid y[x^\uparrow(u{:}M)\,.\,P'' \mid R]$ and $Q = P' \mid y[P''\{^v/_u\} \mid R]$. Assume $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$. This must have been derived from *(Proc Par)* with $\Gamma \rhd_\Xi \overline{x}^y(v)\,.\,P' : \mathtt{Proc}\ A$ and $\Gamma \rhd_\Xi y[x^\uparrow(u{:}M)\,.\,P'' \mid R] : \mathtt{Proc}\ [\,]$. The former must have been derived from (*output down*) with $\Gamma \rhd_\Xi y : \mathtt{Id}\ A_1$, $\Gamma \rhd_\Xi v : M$, $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$ and $(x{:}M) \in A_1$. The latter must have been derived from (*seal*) with $\Gamma \rhd_\Xi y : \mathtt{Id}\ A_1$ and $\Gamma \rhd_\Xi x^\uparrow(u{:}M)\,.\,P'' \mid R : \mathtt{Proc}\ A_2$, where $\mathtt{Proc}\ A_2 \le \mathtt{Proc}\ A_1$, This one has been inferred by (*Proc Par*) from $\Gamma \rhd_\Xi x^\uparrow(u{:}M)\,.\,P'' : \mathtt{Proc}\ (A_3 \oplus [x{:}M])$ and $\Gamma \rhd_\Xi R : \mathtt{Proc}\ A_4$. To simplify the types, we used $A_2 = [x{:}M] \oplus A_3 \oplus A_4$. Lastly we point out that the hypothesis to derive $\Gamma \rhd_\Xi x^\uparrow(u{:}M)\,.\,P'' : \mathtt{Proc}\ (A_3 \oplus [x{:}M])$ from (*input up*) are $\Gamma \rhd_\Xi x : \mathtt{Ch}\ M$, and $\Gamma, u : M' \rhd_\Xi P'' : \mathtt{Proc}\ A_3$, with the condition $x \in \mathrm{dom}(\Xi)$; remark that $u \notin \mathrm{dom}(\Xi)$. This is depicted in fig. A.3. We can apply Corollary 5 with hypothesis $\Gamma, u : M \rhd_\Xi P'' : \mathtt{Proc}\ A_3$ and $\Gamma \rhd_\Xi v : M$ to obtain $\Gamma \rhd_\Xi P'\{^v/_u\} : \mathtt{Proc}\ A_3$. Thus we can draw out the following derivation:

$$\dfrac{\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A \qquad \dfrac{\dfrac{\begin{array}{c}\Gamma \rhd_\Xi R : \mathtt{Proc}\ A_4 \\ \Gamma \rhd_\Xi P''\{^v/_u\} : \mathtt{Proc}\ A_3\end{array}}{\Gamma \rhd_\Xi P''\{^v/_u\} \mid R : \mathtt{Proc}\ A_2} \qquad \Gamma \rhd_\Xi y : \mathtt{Id}\ A_1}{\Gamma \rhd_\Xi y[P''\{^v/_u\} \mid R] : \mathtt{Proc}\ [\,]}}{\Gamma \rhd_\Xi P' \mid y[P''\{^v/_u\} \mid R] : \mathtt{Proc}\ A}$$

Hence $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$.

(move local) Then $P = x^*\{\!|u|\!\}.P' \mid \overline{x}^*\{\!|v|\!\}.Q' \mid v[B]$ and $Q = P' \mid u[B] \mid Q'$. Assume $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$. The only possible derivation is:

$$\dfrac{\dfrac{\begin{array}{c}\Gamma \rhd_\Xi x : \mathtt{Ch}\ A_3 \\ \Gamma \rhd_\Xi u : \mathtt{Id}\ A_3 \\ \Gamma \rhd_\Xi P : \mathtt{Proc}\ A_1\end{array}}{\Gamma \rhd_\Xi x^*\{\!|u|\!\}.P' : \mathtt{Proc}\ A_1} \qquad \dfrac{\begin{array}{c}\Gamma \rhd_\Xi x : \mathtt{Ch}\ A_3 \\ \Gamma \rhd_\Xi v : \mathtt{Id}\ A_3 \\ \Gamma \rhd_\Xi Q' : \mathtt{Proc}\ A_2\end{array}}{\Gamma \rhd_\Xi \overline{x}^*\{\!|v|\!\}.Q' : \mathtt{Proc}\ A_2} \qquad \dfrac{\begin{array}{c}\Gamma \rhd_\Xi v : \mathtt{Id}\ A_3 \\ \Gamma \rhd_\Xi B : \mathtt{Proc}\ A_4\end{array}}{\Gamma \rhd_\Xi v[B] : \mathtt{Proc}\ [\,]}}{\Gamma \rhd_\Xi x^*\{\!|u|\!\}.P' \mid \overline{x}^*\{\!|v|\!\}.Q' \mid v[B] : \mathtt{Proc}\ (A_1 \oplus A_2)}$$

where $A = A_1 \oplus A_2$ and $\mathtt{Proc}\ A_4 \le \mathtt{Proc}\ A_3$.[1] Since we have $\Gamma \rhd_\Xi u : \mathtt{Id}\ A_3$ and $\Gamma \rhd_\Xi B : \mathtt{Proc}\ A_4$ with $\mathtt{Proc}\ A_4 \le \mathtt{Proc}\ A_3$, by *(Proc Seal)* we can conclude $\Gamma \rhd_\Xi u[B] : \mathtt{Proc}\ [\,]$.

[1] we used an immediate generalization of Proc Par rule.

$$
\dfrac{
\dfrac{
\begin{array}{c}
\Gamma \rhd_{\Xi} y : \mathtt{Id}\ A_4 \\
\Gamma \rhd_{\Xi} x : \mathtt{Ch}\ M \\
\Gamma, u{:}M \rhd_{\Xi} P' : \mathtt{Proc}\ A
\end{array}
}{
\Gamma \rhd_{\Xi} x^{y}(u{:}M)\,.\,P' : \mathtt{Proc}\ A
}
\qquad
\dfrac{
\dfrac{
\dfrac{
\Gamma \rhd_{\Xi} x : \mathtt{Ch}\ M' \qquad \Gamma \rhd_{\Xi} v : M' \qquad \Gamma \rhd_{\Xi} P'' : \mathtt{Proc}\ A_1
}{
\Gamma \rhd_{\Xi} \overline{x}^{\uparrow}(v)\,.\,P'' : \mathtt{Proc}\ A_1
}
\qquad
\Gamma \rhd_{\Xi} R : \mathtt{Proc}\ A_2
}{
\Gamma \rhd_{\Xi} \overline{x}^{\uparrow}(v)\,.\,P'' \mid R : \mathtt{Proc}\ A_3
}
\qquad
\Gamma \rhd_{\Xi} y : \mathtt{Id}\ A_4
}{
\Gamma \rhd_{\Xi} y[\overline{x}^{\uparrow}(v)\,.\,P'' \mid R] : \mathtt{Proc}\ [\,]
}
}{
\Gamma \rhd_{\Xi} x^{y}(u{:}M)\,.\,P' \mid y[\overline{x}^{\uparrow}(v)\,.\,P'' \mid R] : \mathtt{Proc}\ A
}
$$

where $\mathtt{Proc}\ A_3 \leq \mathtt{Proc}\ A_4$, $x \in \mathrm{dom}(\Xi)$ and $u \notin \mathrm{dom}(\Xi)$.

Figure A.2: Derivation of $\Gamma \rhd_{\Xi} P : \mathtt{Proc}\ A$ in *(write_out)*.

$$
\dfrac{
\dfrac{
\begin{array}{c}
\Gamma \rhd_{\Xi} v : M \\
\Gamma \rhd_{\Xi} y : \mathtt{Id}\ A_1 \\
\Gamma \rhd_{\Xi} P' : \mathtt{Proc}\ A
\end{array}
}{
\Gamma \rhd_{\Xi} \overline{x}^{y}(v)\,.\,P' : \mathtt{Proc}\ A
}\ ((x{:}M) \in A_1)
\qquad
\dfrac{
\dfrac{
\dfrac{
\Gamma \rhd_{\Xi} x : \mathtt{Ch}\ M \qquad \Gamma, u{:}M \rhd_{\Xi} P'' : \mathtt{Proc}\ A_3
}{
\Gamma \rhd_{\Xi} x^{\uparrow}(u{:}M)\,.\,P'' : \mathtt{Proc}\ (A_3 \oplus [x{:}M])
}
\qquad \Gamma \rhd_{\Xi} R : \mathtt{Proc}\ A_4
}{
\Gamma \rhd_{\Xi} x^{\uparrow}(u{:}M)\,.\,P'' \mid R : \mathtt{Proc}\ A_2
}
\qquad \Gamma \rhd_{\Xi} y : \mathtt{Id}\ A_1
}{
\Gamma \rhd_{\Xi} y[x^{\uparrow}(u{:}M)\,.\,P'' \mid R] : \mathtt{Proc}\ [\,]
}
}{
\Gamma \rhd_{\Xi} \overline{x}^{y}(v)\,.\,P' \mid y[x^{\uparrow}(u{:}M)\,.\,P'' \mid R] : \mathtt{Proc}\ A
}
$$

where $A_2 = [x{:}M] \oplus A_3 \oplus A_4$, $\mathtt{Proc}\ A_2 \leq \mathtt{Proc}\ A_1$, $x \in \mathrm{dom}(\Xi)$ and $u \notin \mathrm{dom}(\Xi)$.

Figure A.3: Derivation of $\Gamma \rhd_{\Xi} P : \mathtt{Proc}\ A$ in *(write_in)*

Thus the following derivation justifies $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$.

$$\dfrac{\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A_1 \qquad \Gamma \rhd_\Xi Q' : \mathtt{Proc}\ A_2 \qquad \dfrac{\begin{array}{c}\Gamma \rhd_\Xi u : \mathtt{Id}\ A_3 \\ \Gamma \rhd_\Xi B : \mathtt{Proc}\ A_4\end{array}}{\Gamma \rhd_\Xi u[B] : \mathtt{Proc}\ [\,]}}{\Gamma \rhd_\Xi P' \mid u[B] \mid Q' : \mathtt{Proc}\ (A_1 \oplus A_2)}$$

(move out) Then $P = x^y\lbrace\!\lbrace u\rbrace\!\rbrace.P' \mid y[\overline{x}^{\uparrow}\lbrace\!\lbrace v\rbrace\!\rbrace.Q' \mid v[B] \mid R]$ and $Q = P' \mid u[B] \mid y[Q' \mid R]$.

Assume $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$. This must have been derived by *(Proc Par)* from $\Gamma \rhd_\Xi x^y\lbrace\!\lbrace u\rbrace\!\rbrace.P' : \mathtt{Proc}\ A$ and $\Gamma \rhd_\Xi y[\overline{x}^{\uparrow}\lbrace\!\lbrace v\rbrace\!\rbrace.Q' \mid v[B] \mid R] : \mathtt{Proc}\ [\,]$. The former must have been derived by *(Rcv Down)* from $\Gamma \rhd_\Xi y : \mathtt{Id}\ A_5$, $\Gamma \rhd_\Xi x : \mathtt{Ch}\ A_1$, $\Gamma \rhd_\Xi u : \mathtt{Id}\ A_1$, and $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$. The latter must have been derived by *(Proc Seal)* from $\Gamma \rhd_\Xi y : \mathtt{Id}\ A_5$ and $\Gamma \rhd_\Xi \overline{x}^{\uparrow}\lbrace\!\lbrace v\rbrace\!\rbrace.Q' \mid v[B] \mid R : \mathtt{Proc}\ A_2$ with the condition $\mathtt{Proc}\ A_2 \leq \mathtt{Proc}\ A_5$. Again, the latter must have been derived by *(Proc Par)* from $\Gamma \rhd_\Xi \overline{x}^{\uparrow}\lbrace\!\lbrace v\rbrace\!\rbrace.Q' : \mathtt{Proc}\ A_3$, $\Gamma \rhd_\Xi v[B] : \mathtt{Proc}\ [\,]$, and $\Gamma \rhd_\Xi R : \mathtt{Proc}\ A_4$ with the condition $A_2 = A_3 \oplus A_4$. Lastly, we have to point out that $\Gamma \rhd_\Xi \overline{x}^{\uparrow}\lbrace\!\lbrace v\rbrace\!\rbrace.Q' : \mathtt{Proc}\ A_3$ must have been derived by *(Snd Up)* from $\Gamma \rhd_\Xi x : \mathtt{Ch}\ A_1$, $\Gamma \rhd_\Xi v : \mathtt{Id}\ A_1$, and $\Gamma \rhd_\Xi Q' : \mathtt{Proc}\ A_3$, that $\Gamma \rhd_\Xi v[B] : \mathtt{Proc}\ [\,]$ must have been derived from $\Gamma \rhd_\Xi v : \mathtt{Id}\ A_1$ and $\Gamma \rhd_\Xi B : \mathtt{Proc}\ A_6$ with the condition $\mathtt{Proc}\ A_6 \leq \mathtt{Proc}\ A_1$. This is depicted in Figure A.4. Since $\Gamma \rhd_\Xi u : \mathtt{Id}\ A_1$, $\Gamma \rhd_\Xi B : \mathtt{Id}\ A_6$, and $\mathtt{Proc}\ A_6 \leq \mathtt{Proc}\ A_1$ by *(Proc Seal)* we can conclude $\Gamma \rhd_\Xi u[B] : \mathtt{Proc}\ [\,]$. Thus we can draw out the following derivation.

$$\dfrac{\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A \qquad \dfrac{\begin{array}{c}\Gamma \rhd_\Xi u : \mathtt{Id}\ A_1 \\ \Gamma \rhd_\Xi B : \mathtt{Proc}\ A_6\end{array}}{\Gamma \rhd_\Xi u[B] : \mathtt{Proc}\ [\,]} \qquad \dfrac{\Gamma \rhd_\Xi y : \mathtt{Id}\ A_5 \qquad \dfrac{\begin{array}{c}\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A_3 \\ \Gamma \rhd_\Xi R : \mathtt{Proc}\ A_4\end{array}}{\Gamma \rhd_\Xi Q \mid R : \mathtt{Proc}\ A_2}}{\Gamma \rhd_\Xi y[Q \mid R] : \mathtt{Proc}\ [\,]}}{\Gamma \rhd_\Xi P' \mid u[B] \mid y[Q' \mid R] : \mathtt{Proc}\ A}$$

Hence $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$.

(move in) Then $P = \overline{x}^y\lbrace\!\lbrace v\rbrace\!\rbrace.P' \mid v[B] \mid y[x^{\uparrow}\lbrace\!\lbrace u\rbrace\!\rbrace.Q' \mid R]$ and $Q = P' \mid y[Q' \mid R \mid u[B]]$.

Assume $\Gamma \rhd_\Xi P : \mathtt{Proc}\ A$. This must have been derived by *(generalized Proc Par)* from the hypothesis $\Gamma \rhd_\Xi \overline{x}^y\lbrace\!\lbrace v\rbrace\!\rbrace.P' : \mathtt{Proc}\ A$, $\Gamma \rhd_\Xi v[B] : \mathtt{Proc}\ [\,]$, and $\Gamma \rhd_\Xi y[x^{\uparrow}\lbrace\!\lbrace u\rbrace\!\rbrace.Q' \mid R] : \mathtt{Proc}\ [\,]$. The former must have been derived by *(mob down send)* from $\Gamma \rhd_\Xi y : \mathtt{Id}\ A_2$, $\Gamma \rhd_\Xi v : \mathtt{Id}\ A_1$, and $\Gamma \rhd_\Xi P' : \mathtt{Proc}\ A$ with the condition that $(x{:}A_1) \in A_2$. The second must have been derived by *(Proc Seal)* from $\Gamma \rhd_\Xi v : \mathtt{Id}\ A_1$ and $\Gamma \rhd_\Xi B : \mathtt{Proc}\ A_6$, wtih $\mathtt{Proc}\ A_6 \leq \mathtt{Proc}\ A_1$. The latter by *(Proc Seal)* from $\Gamma \rhd_\Xi y : \mathtt{Id}\ A_2$ and $\Gamma \rhd_\Xi x^{\uparrow}\lbrace\!\lbrace u\rbrace\!\rbrace.Q' \mid R : \mathtt{Proc}\ A_3$, with the condition $\mathtt{Proc}\ A_3 \leq \mathtt{Proc}\ A_2$. In turn, this must have been derived by *(Proc Par)* from $\Gamma \rhd_\Xi R : \mathtt{Proc}\ A_5$ and $\Gamma \rhd_\Xi x^{\uparrow}\lbrace\!\lbrace u\rbrace\!\rbrace.Q'$ with the condition $A_3 = A_4 \oplus [x{:}A_1] \oplus A_5$. Lastly, to succesfully derive the last hypothesis we must have $\Gamma \rhd_\Xi x : \mathtt{Ch}\ A_1$, $\Gamma \rhd_\Xi u : \mathtt{Id}\ A_1$, and $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A_4$. This is depicted in Figure A.5. By properties of $\oplus$, we have $\mathtt{Proc}\ (A_4 \oplus A_5) \leq \mathtt{Proc}\ A_3 \leq \mathtt{Proc}\ A_2$. Hence we can write out the derivation represented in Figure A.6 and we obtain $\Gamma \rhd_\Xi Q : \mathtt{Proc}\ A$.

$$
\dfrac{
  \dfrac{\begin{array}{c}\Gamma \rhd_{\Xi} y : \mathtt{Id}\ A_5 \\ \Gamma \rhd_{\Xi} x : \mathtt{Ch}\ A_1 \\ \Gamma \rhd_{\Xi} u : \mathtt{Id}\ A_1 \\ \Gamma \rhd_{\Xi} P : \mathtt{Proc}\ A\end{array}}{\Gamma \rhd_{\Xi} x^{y}\{u\}.P' : \mathtt{Proc}\ A}
  \qquad
  \dfrac{\Gamma \rhd_{\Xi} y : \mathtt{Id}\ A_5 \qquad
    \dfrac{
      \dfrac{\begin{array}{c}\Gamma \rhd_{\Xi} x : \mathtt{Ch}\ A_1 \\ \Gamma \rhd_{\Xi} v : \mathtt{Id}\ A_1 \\ \Gamma \rhd_{\Xi} Q' : \mathtt{Proc}\ A_3\end{array}}{\Gamma \rhd_{\Xi} \overline{x}^{\uparrow}\{v\}.Q' : \mathtt{Proc}\ A_3}
      \quad
      \dfrac{\begin{array}{c}\Gamma \rhd_{\Xi} v : \mathtt{Id}\ A_1 \\ \Gamma \rhd_{\Xi} B : \mathtt{Proc}\ A_6\end{array}}{\Gamma \rhd_{\Xi} v[B] : \mathtt{Proc}\ [\,]}
      \quad
      \Gamma \rhd_{\Xi} R : \mathtt{Proc}\ A_4
    }{\Gamma \rhd_{\Xi} \overline{x}^{\uparrow}\{v\}.Q' \mid v[B] \mid R : \mathtt{Proc}\ A_2}
  }{\Gamma \rhd_{\Xi} y[\overline{x}^{\uparrow}\{v\}.Q' \mid v[B] \mid R] : \mathtt{Proc}\ [\,]}
}{\Gamma \rhd_{\Xi} x^{y}\{u\}.P' \mid y[\overline{x}^{\uparrow}\{v\}.Q' \mid v[B] \mid R] : \mathtt{Proc}\ [\,]}
$$

where $A_2 = A_3 \oplus A_4$, $\mathtt{Proc}\ A_6 \leq \mathtt{Proc}\ A_1$, and $\mathtt{Proc}\ A_2 \leq \mathtt{Proc}\ A_5$.

Figure A.4: Derivation of $\Gamma \rhd_{\Xi} P : \mathtt{Proc}\ A$ in *(move out)*.

$$
\dfrac{
\dfrac{\begin{array}{c}\Gamma \rhd_{\Xi} y : \mathtt{Id}\ A_2\\ \Gamma \rhd_{\Xi} v : \mathtt{Id}\ A_1\\ \Gamma \rhd_{\Xi} P' : \mathtt{Proc}\ A_1\end{array}}{\Gamma \rhd_{\Xi} \overline{x}^{y}\{\!|v|\!\}.P' : \mathtt{Proc}\ A}\ (x{:}A_1) \in A_2
\qquad
\dfrac{\begin{array}{c}\Gamma \rhd_{\Xi} v : \mathtt{Id}\ A_1\\ \Gamma \rhd_{\Xi} B : \mathtt{Proc}\ A_6\end{array}}{\Gamma \rhd_{\Xi} v[B] : \mathtt{Proc}\ [\,]}
\qquad
\dfrac{\Gamma \rhd_{\Xi} y : \mathtt{Id}\ A_2 \qquad \dfrac{\dfrac{\begin{array}{c}\Gamma \rhd_{\Xi} x : \mathtt{Ch}\ A_1\\ \Gamma \rhd_{\Xi} u : \mathtt{Id}\ A_1\\ \Gamma \rhd_{\Xi} Q : \mathtt{Proc}\ A_4\end{array}}{\Gamma \rhd_{\Xi} x^{\uparrow}\{\!|u|\!\}.Q' : \mathtt{Proc}\ (A_4 \oplus [x{:}A_1])} \qquad \Gamma \rhd_{\Xi} R : \mathtt{Proc}\ A_5}{\Gamma \rhd_{\Xi} x^{\uparrow}\{\!|u|\!\}.Q' \mid R : \mathtt{Proc}\ A_3}}{\Gamma \rhd_{\Xi} y[x^{\uparrow}\{\!|u|\!\}.Q' \mid R] : \mathtt{Proc}\ [\,]}
}{\Gamma \rhd_{\Xi} \overline{x}^{y}\{\!|v|\!\}.P' \mid v[B] \mid y[x^{\uparrow}\{\!|u|\!\}.Q' \mid R] : \mathtt{Proc}\ A}
$$

where $A_3 = A_4 \oplus A_5$, $\mathtt{Proc}\ A_6 \leq \mathtt{Proc}\ A_1$, and $\mathtt{Proc}\ A_3 \leq \mathtt{Proc}\ A_2$.

Figure A.5: Derivation of $\Gamma \rhd_{\Xi} P : \mathtt{Proc}\ A$ in *(move in)*.

$$\dfrac{\Gamma \rhd_{\Xi} P' : \texttt{Proc}\ A_1 \qquad \dfrac{\Gamma \rhd_{\Xi} y : \texttt{Id}\ A_2 \qquad \dfrac{\Gamma \rhd_{\Xi} Q : \texttt{Proc}\ A_4 \qquad \Gamma \rhd_{\Xi} R : \texttt{Proc}\ A_5 \qquad \dfrac{\begin{array}{c}\Gamma \rhd_{\Xi} u : \texttt{Id}\ A_1 \\ \Gamma \rhd_{\Xi} B : \texttt{Id}\ A_6\end{array}}{\Gamma \rhd_{\Xi} u[B] : \texttt{Proc}\ [\,]}}{\Gamma \rhd_{\Xi} Q' \mid R \mid u[B] : \texttt{Proc}\ (A_4 \oplus A_5)}}{\Gamma \rhd_{\Xi} y[Q' \mid R \mid u[B]] : \texttt{Proc}\ [\,]}}{\Gamma \rhd_{\Xi} P' \mid y[Q' \mid R \mid u[B]] : \texttt{Proc}\ A}$$

Figure A.6: Derivation of $\Gamma \rhd_{\Xi} Q : \texttt{Proc}\ A$ in *(move in)*.

(red par) Then $P = P' \mid R$ and $Q = Q' \mid R$. Assume $\Gamma \rhd_\Xi P' \mid R : \texttt{Proc}\ A$. This must have been derived by *(Proc Par)* from $\Gamma \rhd_\Xi P' : \texttt{Proc}\ A_1$ and $\Gamma \rhd_\Xi R : \texttt{Proc}\ A_2$ with $A = A_1 \oplus A_2$. By the induction hypothesis we have $\Gamma \rhd_\Xi Q' : \texttt{Proc}\ A_1$ and by *(Proc Par)* $\Gamma \rhd_\Xi Q' \mid R : \texttt{Proc}\ A$.

(red res) Then $P = (\boldsymbol{\nu}\, x{:}M)P'$ and $Q = (\boldsymbol{\nu}\, x{:}M)Q'$. Assume $\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)P' : \texttt{Proc}\ A$. This must have been derived by *(res ...)* from $\Gamma, x{:}M \rhd_{\Xi,(x)} P' : \texttt{Proc}\ A$ with the condition $x \notin fn(A)$. By the induction hypothesis we have $\Gamma, x{:}M \rhd_{\Xi,(x)} Q' : \texttt{Proc}\ A$ and by *(res ...)* $\Gamma \rhd_\Xi (\boldsymbol{\nu}\, x{:}M)Q' : \texttt{Proc}\ A$.

(red amb) Then $P = u[P']$ and $Q = u[Q']$. Assume $\Gamma \rhd_\Xi u[P'] : \texttt{Proc}\ [\,]$. This must have been derived by *(Proc Seal)* from $\Gamma \rhd_\Xi y : \texttt{Id}\ A_1$ and $\Gamma \rhd_\Xi P' : \texttt{Proc}\ A_2$ with $\texttt{Proc}\ A_2 \leq \texttt{Proc}\ A_3$. By the induction hypothesis we have $\Gamma \rhd_\Xi Q' : \texttt{Proc}\ A_3$ and by *(Proc Seal)* $\Gamma \rhd_\Xi u[Q'] : \texttt{Proc}\ A$.

(red $\equiv$) Here $P \equiv P'$, $P' \rightarrow Q'$, and $Q' \equiv Q$. Assume $\Gamma \rhd_\Xi P : \texttt{Proc}\ A$. By Proposition 3 $\Gamma \rhd_\Xi P' : \texttt{Proc}\ A$. By the induction hypothesis, $\Gamma \rhd_\Xi Q' : \texttt{Proc}\ A$ and again by Proposition 3 we can conclude $\Gamma \rhd_\Xi Q : \texttt{Proc}\ A$.

$\square$