Linear logic as a foundation for service-oriented computing

Hervé Grall, Nicolas Tabareau

Linear logic as a foundation for service-oriented computing

Hervé Grall Nicolas Tabareau

INRIA - École des mines de Nantes

Abstract. We present a calculus that provides formal and unified foundations to service-oriented computing. Service-oriented computing allows network-based software applications to be developed by resorting to services as primitive components. To date, there are two popular – and often antagonistic – models for service-oriented computing. On the first hand, the computation-oriented model, illustrated by WS* Web services, considers services as sets of operations. On the other hand, the resource-oriented model, illustrated by Restful Web services, considers services as interfaces to resources. The lack of unified models leads to adaptation, integration and coordination problems, three major concerns in this field. Our calculus restores unity to service-oriented computing, by reconciling both points of view. We give the operational semantics of the calculus using chemical solutions, and illustrate its expressive power not only as a query language over resources with support for recursion and aggregation, but also as a concurrent process language. Finally, we show that our calculus is also meaningful in logic programming since computation can be interpreted as proof search in focused linear logic with resource modalities: affine, contractible and exponential.

1 Introduction

Service-oriented computing is a solution to organize the exchange of messages in a network-based applications, by using services as primitive components. To date, there are two popular – and often antagonistic – models for service-oriented computing [14]. First, interoperability and integration issues has led to the development of the technology of WS* Web services, based on XML and Internet technologies. A service corresponds to a set of operations, implemented with any technical infrastructure, declared in a specific language, like the Web Services Description Language (WSDL), and accessed via a standard protocol like the Simple Object Access Protocol (SOAP). Processes, corresponding to the composition of operations, can be locally defined from an orchestration language like the Business Process Execution Language for Web Services (BPEL4WS), and globally specified with a choreography language like the Web Services Choreography Description Language (WS-CDL). Since the primitive entities are operations, we say that the model is computation-oriented. More recently, an alternative solution has been brought forward leading to RESTful web services. Since the primitive entities are resources, we say that the model is resource-oriented. The services correspond to four standard operations, allowing resources to be created, requested, updated and deleted. In other words, resources have CRUD (Create, Request, Update, Delete) interfaces. In this model, processes are CRUD scripts, very similar to database queries and updates. Thus, CRUD languages have been developed as variants of the SQL language: see for instance the language `YQL` from Yahoo. But currently, contrary to the computation-oriented model, there is no standard for CRUD languages like BPEL for orchestration languages, as shown by the current diversity of CRUD languages in use.

The absence of a unified model leads to adaptation, integration and coordination problems, which are major concerns in the field. For instance, assume there are two systems exporting services, which essentially have the same functionalities. Given a process defined to work with the first system, how to adapt the process in order to work with the second system? How to define a process integrating both systems? Given two processes working with one system, how to coordinate them?

Various calculi [4,12,17] have been proposed with the aim to capture aspects of service-oriented computing, from a verification or a modeling point of view but also from a formalization and programming point of view, which is related to our approach. However, these calculi are essentially computation-oriented and not resource-oriented. To solve these problems, we aim at using an abstract machine as a mediator between processes and services. The intent is to compile any process in the language of the abstract machine, which implies two strong requirements for the core calculus at the heart of the machine: it must be able to express query languages with support for recursion and aggregation, but also concurrent process languages. Our proposal is an edition calculus, that is a calculus that edits relational structures, which can abstractly represent resources and computations over resources. Our main contribution is to show that indeed our edition calculus is a good candidate for the abstract machine, since we can directly encode two paradigmatic languages, Datalog with negation on the query side and the π-calculus on the concurrent process side.

During the quest for a unified calculus, it appeared very fruitful to consider our design choice in the light of linear logic and of the notion of resource modalities. This should not come as a surprise as linear logic already appeared to be fruitful in the field of logic programming. Indeed, it is possible to overcome some limitations of Datalog, like the impossibility to express aggregations, by resorting to linear logic: see for instance the logic programming language used to program linear logical algorithms [15], or the language "Constraint Handling Rules" with a linear semantics [3], which can be seen as extensions of Datalog. Our edition calculus clearly extends these languages. Likewise, there is a strong connexion between concurrency and linear logic [6]. Thus, our calculus has also its roots in the chemical reaction model: It is a restriction of a coordination language with schedulers [7] for Gamma [2], whereas the main differences with the join-calculus [8] are the absence of the linearity constraint on messages, and the distinction between channels and messages.

Plan of the paper. In Section 2, we introduced the edition calculus and give its operational semantics using chemical solutions. In Section 3, we give a translation of Datalog with a negation, by implementing the alternating fixpoint construction. In Section 4, we provide a translation of the asynchronous π-calculus. Finally, we show in Section 5 that our calculus is also meaningful in logic programming since computation can be interpreted as proof search in focused linear logic with various notion of resource modalities: affine, contractible and exponential.

2 The edition calculus

2.1 Syntax: atoms, molecules, rules, scripts

The most primitive entities in the edition calculus are variables, multi-relations and relations. The set of variables is denoted by $\mathcal{V}$. There is no constant in the calculus as they can be easily encoded. When the constant is just a name, like the name of a node in a graph, it is simply

encoded by a new variable with a scope corresponding to the use of the constant. When the constant denotes a value, like a natural number, it is encoded by a variable and atoms describing the value. For instance, to denote zero, we can use x, specified by atom $\mathrm{zero}(x)$. Multi-relations are multi-sets: an element in a multi-relation may have multiple occurrences. On the contrary, relations are sets: an element in a relation has a unique occurrence. Exhaustive duplicate elimination transforms a multi-relation into a relation. Relations and multi-relations come from two infinite sets, $\mathcal{R}$ and $\mathcal{M}$ respectively. The sets $\mathcal{V}$, $\mathcal{R}$ and $\mathcal{M}$ are assumed to be infinite and disjoint. Whereas variables are just names, relations have also an arity, a natural number.

From multi-relations, relations and variables, atoms are built, by applying any relation R or multi-relation M with arity n to a sequence of n variables, $v_1, \ldots, v_n$, to get atoms $R(v_1, \ldots, v_n)$ and $M(v_1, \ldots, v_n)$ respectively. Atoms $a_1, \ldots a_p$ can be joined together to make a molecule $a_1 \,\&\, \ldots \,\&\, a_p$.

The heart of the edition calculus is the use of reactions that transform molecules into other molecules. A reaction is specified by a rule $j_1 \triangleright \overrightarrow{\nu v}.j_2$, transforming any molecule matching the molecule pattern j_1 to a new molecule matching the molecule pattern j_2, using new variables in $\overrightarrow{v}$. Any variable free in j_2, that is occurring in j_2 without being declared in $\overrightarrow{v}$, must be bound by the rule: it must occur in j_1. Here, and in the following, the notation $\overrightarrow{x}$ denotes a sequence of x, or depending on the context, a set or multiset of x, when the particular members of the collection do not matter; the sequence, set or multiset may be empty.

Finally, the edition calculus is a language of scripts, which can be seen as specifications of schedules for rules. There are basic scripts, the empty one, which contains no rule and does nothing, and the singleton one, which contains a unique rule that can be fired at most once. The parallel operator allows scripts to be concurrently active. For instance, the script $r \,,\, r$ allows the rule r to be fired twice, whereas the script $r \,,\, r'$ allows the rules r and r' to be fired exactly once each one, in any order. If a script needs to be executed an indefinite number of times, the replication operator can be used: for instance, the script r^ω means that the rule is always ready to be fired. There is also a sequential operator, allowing the transformations defined by scripts to be sequentially composed. Table 1 sums up the syntax of the edition calculus. As usual, we denote by $\mathrm{FV}(j)$ the set of free variables occurring in the molecule j.

2.2 Some simple examples

Consider a unary relation or multi-relation P. To define the initial state of P, we can use a simple rule:

$$\emptyset \triangleright \nu a.\nu b.P(a) \,\&\, P(b)\,;$$

In the following, we will call the multiset of atoms describing the relations and multi-relations the *solution*. After the unique firing of the rule, the relation or multi-relation is initialized, here containing two elements, a and b, two new variables. The rule can no longer be fired: actually, it will be removed from the active script, which we call the *reaction*.

To make a copy with a removal of P, we can then write:

$$(P(x) \triangleright Q(x))^\omega$$

We need to resort to the replication operator in order to allow multiple firings of the rule. Very often, rules are replicated. The rules that are not replicated are used to define solutions in particular states, or transitions between states. Note that without a replication operator, if all rules were implicitly replicated, it would be impossible to define an initial solution, as above. The

Script	$s ::= \emptyset$	Skip
	$\mid r$	Rule
	$\mid s \,,\, s$	Parallel
	$\mid s \,;\, s$	Sequence
	$\mid s^{\omega}$	Replication
Rule	$r ::= j_1 \triangleright \overrightarrow{\nu v}.j_2$	Transformation of j_1 into j_2 with new names $\overrightarrow{v}$ ($\overrightarrow{v} = \mathrm{FV}(j_2) - \mathrm{FV}(j_1)$)
Molecule	$j ::= \emptyset \mid a \mid j \,\&\, j$	Conjunction of atoms
Atom	$a ::= R\,(\overrightarrow{v}) \mid M\,(\overrightarrow{v})$	Relation or multi-relation applied to variables
Relation	$R ::= \ldots$	Names in $\mathcal{R}$
Multi-relation	$M ::= \ldots$	Names in $\mathcal{M}$
Variable	$v ::= \ldots$	Names in $\mathcal{V}$

Table 1. Edition calculus

preceding rule applied twice consumes $P(a)$ and $P(b)$ and produces $Q(a)$ and $Q(b)$. If P and Q are both relations, or are both multi-relations, the script just describes a renaming operation. It is more interesting when P is a multi-relation and Q a relation. Assume that the solution initially contains atoms $P(a), P(a), P(b)$. After the conversion of the multi-relation P into a relation Q, the solution contains two atoms: $Q(a), Q(b)$. Indeed, duplicates are automatically eliminated for relations.
To make a clone of P, we can try:

$$(P(x) \triangleright P(x) \,\&\, Q(x))^{\omega}$$

The script works when Q is a relation. But when Q is a multi-relation, it diverges: the script generates a solution containing an infinity of atom $Q(a)$, from some atom $P(a)$ initially in the solution. Here is a right script for cloning:

$$(P(x) \triangleright P'(x) \,\&\, Q(x))^{\omega} \,;\, (P'(x) \triangleright P(x))^{\omega}$$

Without the sequence operator, it would be impossible to define a clone operation for multi-relations. Indeed, assume that P and Q are multi-relations, and that the initial solution contains n atoms $P(a)$. Assume there exists a script that does not use the sequence operator and that implements the clone operation. Its execution terminates with a solution containing n atoms $P(a)$ and n atoms $Q(a)$, and nothing else. Now, add to the initial solution one atom $P(a)$, to get $n+1$ atoms $P(a)$. The execution can reach an intermediate solution, containing $n+1$ atoms $P(a)$ and n atoms $Q(a)$, by putting aside one atom $P(a)$ and following the first execution: this partial execution is possible because the script only contains rules in parallel. At this stage, no reaction can fire by only using n atoms $P(a)$ and n atoms $Q(a)$. It means that there is a rule that consumes the molecule $P(a) \& \ldots \& P(a)$ containing $n+1$ atoms. This is a contradiction: there

is no finite script implementing the clone operation. In other words, the sequence operator, like the replication operator, does increase the expressive power of our edition calculus.

We can now give a general form for scripts when they are used to edit relations or multi-relations. Suppose we want to add or remove Δ from the relation or multi-relation P. To slightly simplify, we assume that P and Δ are unary. We start by cloning P, then we compute Δ from the clone of P, and finally we realize the addition or subtraction of Δ.

$$(P(x) \triangleright P'(x)\ \&\ Q(x))^{\omega}\,;(P'(x) \triangleright P(x))^{\omega}\,;$$
$$\ldots;\ \text{(script computing } \Delta \text{ from clone } Q)$$
$$(\Delta(x) \triangleright P(x))^{\omega}\ \text{(addition)} \qquad (\Delta(x)\ \&\ P(x) \triangleright \emptyset)^{\omega}\ \text{(subtraction)}$$

Suppose that we want to assign the relation or multi-relation U to the relation or multi-relation P. We start by copying and removing P, then we compute U from the copy of P, and finally we realize the assignment of U to P.

$$(P(x) \triangleright Q(x))^{\omega}$$
$$\ldots;\ \text{(script computing } U \text{ from copy } Q)$$
$$(U(x) \triangleright P(x))^{\omega}\ \text{(assignment)}$$

2.3 A chemical abstract machine for executing edition scripts

The operational semantics of our calculus is given by a chemical abstract machine. A configuration γ is of the form

$$\rho \vdash \sigma$$

where ρ is a reaction – reactions coincide with scripts – and σ is a solution built on molecules, multiset formation and scope restriction: see Table 2.

Solution	$\sigma ::= j$	Molecule
	$\mid \sigma\,,\sigma$	Multiset
	$\mid \nu\, v.\sigma$	Scope restriction
Reaction	$\rho ::= s$	Script
Configuration	$\gamma ::= \rho \vdash \sigma$	Reaction in a solution

Table 2. Chemical abstract machine – Reaction and solution

Table 4 defines the structural congruence between configurations. All the equivalence are standard except for the replication law which indicates that a reaction ρ^{ω} can be seen as infinitely many copies of ρ executed in parallel.

The execution of a configuration is defined in three steps. First we define the *duplicate elimination* $\Rrightarrow_*$ (see Table 4) that eliminates every duplicated relational atom. This normalization process will be performed between each reduction step to ensure that relational atoms occur at most once in a configuration. Then we define the *chemical reduction* $\rightarrow$ that describes the

Fusion and fission	$\emptyset \vdash j_1 \,\&\, j_2 \equiv \emptyset \vdash j_1 \,, j_2$
Scope extrusion	$\emptyset \vdash \sigma_1 \,, \nu v.\sigma_2 \equiv \emptyset \vdash \nu v.(\sigma_1 \,, \sigma_2) \;\; (v \notin \mathrm{FV}(\sigma_1))$
Scope laws	$\emptyset \vdash \nu v.\emptyset \equiv \emptyset \vdash \emptyset$
	$\emptyset \vdash \nu v_1.\nu v_2.\sigma \equiv \emptyset \vdash \nu v_2.\nu v_1.\sigma$
Sequence laws	$s \,; \emptyset \vdash \emptyset \equiv \emptyset \,; s \vdash \emptyset \equiv s \vdash \emptyset$
	$(s_1 \,; s_2) \,; s_3 \vdash \emptyset \equiv s_1 \,; (s_2 \,; s_3) \vdash \emptyset$
Replication	$s^\omega \vdash \emptyset \equiv s^\omega \,, s \vdash \emptyset$
Replication law	$(s_1 \,, s_2)^\omega \vdash \emptyset \equiv {s_1}^\omega \,, {s_2}^\omega \vdash \emptyset$

Table 3. Chemical abstract machine – Structural congruence

Elimination $\quad \emptyset \vdash R(\overrightarrow{v}) \,, R(\overrightarrow{v}) \Rrightarrow \emptyset \vdash R(\overrightarrow{v})$ $\qquad$ Compatibility $\quad \dfrac{\gamma_1 \equiv \gamma_2 \quad \gamma_2 \Rrightarrow \gamma_3 \quad \gamma_3 \equiv \gamma_4}{\gamma_1 \Rrightarrow \gamma_4}$

Convergence $\quad \dfrac{\gamma_1 \equiv \gamma_2 \quad \neg(\gamma_2 \Rrightarrow)}{\gamma_1 \Rrightarrow_0 \gamma_2}$ $\qquad$ $\dfrac{\gamma_1 \Rrightarrow_0 \gamma_2}{\gamma_1 \Rrightarrow_* \gamma_2}$ $\qquad$ $\dfrac{\gamma_1 \Rrightarrow \gamma_2 \quad \gamma_2 \Rrightarrow_* \gamma_3}{\gamma_1 \Rrightarrow_* \gamma_3}$

Table 4. Chemical abstract machine – Duplicate elimination

basic reduction of the chemical abstract machine (see [8] for more explanations) . Finally, we define the *progression* $\Rightarrow$ that applies a chemical reduction to a configuration γ followed by duplicate elimination and check that the resulting configuration is not equivalent to γ to ensure progression: see Table 5.

An *execution* of a configuration γ is a sequence of progressions $\gamma \Rightarrow^* \gamma'$. We say that an execution $\gamma \Rightarrow^* \gamma'$ is *complete* when the configuration γ' can no longer progress.

3 The calculus as a query language : an encoding of Datalog with negation

3.1 The alternating fixpoint construction

A program in Datalog with negation is a set of inference rules. An inference rule is of the form $a \leftarrow l_1 \wedge \ldots l_n$, where the head a is an atom and where each literal l_i in the body is either a positive literal, that is an atom a_i, or a negative literal, that is the negation $\neg a_i$ of an atom a_i. It is a logical implication, asserting that from $l_1, \ldots, l_n$, you can deduce a. A fact is represented by a rule $a \leftarrow \mathrm{true}$, called an axiom. Atoms are defined from relations (and not multi-relations) applied to variables and constants. As usual, we assume that the rules are range-restricted: in any rule, each variable also occurs in a positive body literal. The condition ensures the existence of a finite set, such that each variable takes its value in this set. This finite set is a subset of the *universe*, which is the set of all constants occurring in the program.

Whereas Datalog has a univocal fixpoint semantics, there are different fixpoint semantics for Datalog with negation. Here, we will assign to each program its well-founded model [9], which

$$\frac{}{j_1 \triangleright \overrightarrow{\nu v}.j_2 \vdash j_1[\tau] \to \emptyset \vdash (\overrightarrow{\nu v}.j_2)[\tau]}$$

$$\frac{r \vdash \sigma_1 \to \emptyset \vdash \sigma_2}{r \vdash \sigma_1 \,, \sigma \to \emptyset \vdash \sigma_2 \,, \sigma} \qquad \frac{r \vdash \sigma_1 \to \emptyset \vdash \sigma_2}{r \vdash \nu v.\sigma_1 \to \emptyset \vdash \nu v.\sigma_2} \qquad \frac{\rho_1 \vdash \sigma_1 \to \rho_2 \vdash \sigma_2}{\rho_1 \,, \rho \vdash \sigma_1 \to \rho_2 \,, \rho \vdash \sigma_2}$$

$$\frac{\rho_1 \vdash \sigma_1 \to \rho_2 \vdash \sigma_2}{\rho_1 \,; \rho \vdash \sigma_1 \to \rho_2 \,; \rho \vdash \sigma_2} \qquad \frac{\rho_1 \vdash \sigma_1 \to \rho_2 \vdash \sigma_2 \quad \neg(\rho \vdash \sigma_1 \Rightarrow)}{\rho \,; \rho_1 \vdash \sigma_1 \to \rho_2 \vdash \sigma_2}$$

$$\frac{\gamma_1 \Rrightarrow_0 \gamma_2 \quad \gamma_2 \to \gamma_3 \quad \gamma_3 \Rrightarrow_* \gamma_4 \quad \neg(\gamma_1 \equiv \gamma_4)}{\gamma_1 \Rightarrow \gamma_4}$$

Table 5. Chemical abstract machine – Reduction and progression

can be characterized by an alternating fixpoint construction [16]. In this model, a ground atom is either true, false or unknown. In order to compute the set of true atoms, two approximations are computed. The first one computes the set of atoms that are certainly true: this is an under-approximation. The second one computes the set of atoms that are possibly true: this is an over-approximation. By complementation, we get the set of atoms that are possibly false and certainly false respectively: an atom is possibly false if and only if it is not certainly true, and certainly false if and only if it is not possibly true. More formally, given a program Λ in Datalog with negation, let $\mathcal{U}$ be the finite universe, and $\mathcal{H}$ be the Herbrand base, the finite set of all the ground atoms defined from the relations in Λ and the constants in $\mathcal{U}$. We define the immediate consequence operator as follows, for any set of negative ground literals N and positive ground literals A:

$$\Phi_\Lambda[N](A) \stackrel{\text{def}}{=} \{a[\tau] \mid \exists r \in \Lambda . r = a \leftarrow l_1 \wedge \ldots \wedge l_n, \forall i \in \{1, \ldots, n\} . l_i[\tau] \in N + A\}$$

We introduce four sequences $(C_i^+)_i, (C_i^-)_i, (P_i^+)_i, (P_i^-)_i$, with the following interpretation:

- C_i^+: set of ground atoms that are certainly true at rank i
- C_i^-: set of ground atoms that are certainly false at rank i
- P_i^+: set of ground atoms that are possibly true at rank i
- P_i^-: set of ground atoms that are possibly false at rank i

These sequences are inductively defined as follows. We use some usual notations: $\overline{X}$ denotes the complement of the set X in $\mathcal{H}$, and $\neg X$ the set of the negations of atoms in X; $\mathrm{lfp}(\varphi)$ denotes the least fixpoint of the operator φ defined over the powerset $2^{\mathcal{H}}$.

$$\begin{array}{ll} C_0^+ = C_0^- = \emptyset & P_0^+ = P_0^- = \mathcal{H} \\ C_{i+1}^+ = \mathrm{lfp}(\Phi_\Lambda[\neg C_i^-]) & P_{i+1}^+ = \mathrm{lfp}(\Phi_\Lambda[\neg P_i^-]) \\ C_{i+1}^- = \overline{P_{i+1}^+} & P_{i+1}^- = \overline{C_{i+1}^+} \end{array}$$

The computation halts when the four sequences become stationary. They are ultimately stationnary since the sequences $(C_i^+)_i$ and $(C_i^-)_i$ are increasing whereas the sequences $(P_i^+)_i$ and $(P_i^-)_i$ are decreasing. The well-founded model is derived from the limits C^+ and C^- of the sequences $(C_i^+)_i$ and $(C_i^-)_i$:

- C^+ is the set of ground atoms that are true in the model,
- C^- is the set of ground atoms that are false in the model.

3.2 Implementation in the edition calculus

To implement the alternating fixpoint construction in the edition calculus, we prefer to modify the inductive definition by adding difference sequences $(\Delta C_i^+)_i, (\Delta C_i^-)_i, (\Delta P_i^+)_i, (\Delta P_i^-)_i$. Indeed, they simplify the computations by allowing the sequences $(C_i^-)_i$ and $(P_i^-)_i$ to be incrementally computed. See Figure 1 for a visual representation of the following equations.

$$C_0^+ = C_0^- = \emptyset \qquad P_0^+ = P_0^- = \mathcal{H}$$

$$C_{i+1}^+ = \mathrm{lfp}(\Phi_\Lambda[\neg C_i^-]) \qquad \Delta C_{i+1}^+ = C_{i+1}^+ - C_i^+ \qquad \Delta P_{i+1}^- = \Delta C_{i+1}^+$$

$$P_{i+1}^+ = \mathrm{lfp}(\Phi_\Lambda[\neg P_i^-]) \qquad \Delta P_{i+1}^+ = P_i^+ - P_{i+1}^+ \qquad \Delta C_{i+1}^- = \Delta P_{i+1}^+$$

$$C_{i+1}^- = C_i^- + \Delta C_{i+1}^- \qquad P_{i+1}^- = P_i^- - \Delta P_{i+1}^-$$

Now, we come to the translation of a datalog program. Datalog rules use variables and constants whereas the rules in the edition calculus only use variables. To represent constants, we use unary relations, one per constant. In a rule we replace each constant c by a variable x, and add a premise $F(x)$ to the rule, if F is the relation associated to the constant c. With this transformation, the datalog program contains no axiom. Axioms are shared by all programs: they just assert that for any constant c, we have $F(c)$ if F is the associated relation. To translate the program in a script generating the well-founded model, we will use the following relations (or multi-relations):

- $F_1, \ldots, F_n$: n relations corresponding to the constants of the universe
- U: multi-relation representing the universe $\mathcal{U}$ (using multi-relations eases the generation of the Herbrand base as we will see)
- for each relation R occurring in the program, relations

  $$C^+[R], C^-[R], P^+[R], P^-[R], \Delta C^+[R], \Delta C^-[R], \Delta P^+[R], \Delta P^-[R]$$

  corresponding to the projection of the sets

  $$C_i^+, C_i^-, P_i^+, P_i^-, \Delta C_i^+, \Delta C_i^-, \Delta P_i^+, \Delta P_i^-$$

  over the relation R
- for each relation R occurring in the program, relations $C[R]$ and $P[R]$ corresponding to the result of the computation of the least fixpoint
- $\mathrm{start}, \mathrm{next}$: two relations with arity zero used to control the execution

We start by generating the axioms $F_i(x_i)$ and a description of the universe. We denote by m the greatest arity of the relations occurring in the program. If a is an atom, the notation a^m represents the conjunction of m atoms a.

$$\emptyset \triangleright \nu x_1 \ldots \nu x_n . F_1(x_1) \,\&\, \ldots \,\&\, F_n(x_n) \,\&\, U(x_1)^m \,\&\, \ldots \,\&\, U(x_n)^m \,;$$

We then initialize the relations $P^+[R]$ and $P^-[R]$ to the Herbrand base, for all relation R. The notation $(s_R)_R$ means that the scripts s_R are composed in parallel.

$$\left(\overrightarrow{U(x)} \triangleright P^+[R](\overrightarrow{x}), P^-[R](\overrightarrow{x}), \overrightarrow{U(x)}\right)^\omega_R \,;$$

Fig. 1. The alternating fixpoint construction

Note that we can use this simple script because U is a multi-relation. For any atom $U(x)$, the rule requires at most m occurrences of the atom, where m is the maximal arity. The computation can now begin:

$$\emptyset \triangleright \mathrm{start}\,;$$

The main part of the script computes the positive facts, certain and possible, by using two auxiliary scripts $\mathbf{C}$ and $\mathbf{P}$ that we will describe later, and then update the negative facts, certain and possible. If no update happens, then the computation stop. Otherwise, a new computation can start.

$$\Big(\mathrm{start} \triangleright \emptyset\,; \mathbf{C}\,, \mathbf{P}\,; \\ \big((\Delta C^+[R](\overrightarrow{x}) \triangleright \mathrm{next}\ \&\ \Delta P^-[R](\overrightarrow{x}))^\omega\big)_R\,, \big((\Delta P^+[R](\overrightarrow{x}) \triangleright \mathrm{next}\ \&\ \Delta C^-[R](\overrightarrow{x}))^\omega\big)_R\,; \\ \mathrm{next} \triangleright \mathrm{start}\Big)^\omega$$

Finally, we can describe the script $\mathbf{C}$ computing the certain facts. It starts by updating the negative facts, then generates the positive facts from the translation $[\![\Lambda]\!]_{\mathbf{C}}$ of the datalog program Λ, and finally updates the sets accordingly.
Script $\mathbf{C}$ for certain facts

$$\big((\Delta C^-[R](\overrightarrow{x}) \triangleright C^-[R](\overrightarrow{x}))^\omega\big)_R\,; [\![\Lambda]\!]_{\mathbf{C}}\,; \\ \big((C[R](\overrightarrow{x})\ \&\ C^+[R](\overrightarrow{x}) \triangleright C^+[R](\overrightarrow{x}))^\omega\big)_R\,; \\ \big((C[R](\overrightarrow{x}) \triangleright C^+[R](\overrightarrow{x})\ \&\ \Delta C^+[R](\overrightarrow{x}))^\omega\big)_R$$

The script $\mathbf{P}$ for possible facts is very similar. It generates the positive facts from the translation $[\![\Lambda]\!]_{\mathbf{P}}$ of the datalog program Λ.
Script $\mathbf{P}$ for possible facts

$$\big((\Delta P^-[R](\overrightarrow{x})\ \&\ P^-[R](\overrightarrow{x}) \triangleright \emptyset)^\omega\big)_R\,; \\ [\![\Lambda]\!]_{\mathbf{P}}\,; \\ \big((P[R](\overrightarrow{x})\ \&\ P^+[R](\overrightarrow{x}) \triangleright P[R](\overrightarrow{x}))^\omega\big)_R\,; \\ \big((P^+[R](\overrightarrow{x}) \triangleright \Delta P^+[R](\overrightarrow{x}))^\omega\big)_R\,; \\ \big((P[R](\overrightarrow{x}) \triangleright P^+[R](\overrightarrow{x}))^\omega\big)_R$$

It remains to give the direct translations of a datalog program in rules in the edition calculus, by a structural induction.

$$[\![R(\overrightarrow{x})]\!]_{\mathbf{C}} = C[R](\overrightarrow{x}) \qquad [\![R(\overrightarrow{x})]\!]_{\mathbf{P}} = P[R](\overrightarrow{x})$$

$$[\![\neg R(\overrightarrow{x})]\!]_{\mathbf{C}} = C^{-}[R](\overrightarrow{x}) \qquad [\![\neg R(\overrightarrow{x})]\!]_{\mathbf{P}} = P^{-}[R](\overrightarrow{x})$$

$$[\![a \leftarrow l_1 \wedge \ldots l_n]\!]_{-} = \left([\![l_1]\!]_{-} \,\&\, \ldots \,\&\, [\![l_n]\!]_{-} \triangleright [\![l_1]\!]_{-} \,\&\, \ldots \,\&\, [\![l_n]\!]_{-} \,\&\, [\![a]\!]_{-}\right)^{\omega}$$

$$[\![r_1, r_2]\!]_{-} = [\![r_1]\!]_{-} \,, [\![r_2]\!]_{-}$$

4 The calculus as a concurrent language

We consider in this paper a variant of the asynchronous π-calculus introduced by Honda and Tokoro [11] and independently by Boudol [5]. Without loss of generality, we allow only monadic messages and replicated input instead of more general recursion.

The syntax of the asynchronous π-calculus is given in Table 6. We do not recall here the structural congruence that makes the parallel composition a symmetric monoidal law, with 0 as neutral element and that deals with scope restriction. Let us just recall the two main rules of the reduction semantics which capture the ability of processes to communicate through channels:

$$x(y).P \mid \bar{x}\langle z\rangle \rightarrow P[z/y] \qquad \text{and} \qquad !\,x(y).P \mid \bar{x}\langle z\rangle \rightarrow !\,x(y).P \mid P[z/y].$$

The first rule is the standard communication on channel x that induces a substitution of y with z in P. The second rule is identical except that it keeps a copy of the reception process: a replication happens.

To every process P in the π-calculus, we associate the solution

$$\nu\, l.[\![P]\!]_{\pi}(l)$$

in the edition calculus inductively defined as

$$\begin{array}{rcl}
[\![\bar{x}\langle y\rangle]\!]_{\pi}(l) & = & \mathbf{E}(l, x, y) \\
[\![x(y).P]\!]_{\pi}(l) & = & \nu\, y.\nu\, l'.\mathbf{R}(l, x, y, l') \,\&\, [\![P]\!]_{\pi}(l') \\
[\![P_1 \mid P_2]\!]_{\pi}(l) & = & [\![P_1]\!]_{\pi}(l) \,\&\, [\![P_2]\!]_{\pi}(l) \\
[\![!\,x(y).P]\!]_{\pi}(l) & = & \nu\, y.\nu\, l'.\mathbf{RR}(l, x, y, l') \,\&\, [\![P]\!]_{\pi}(l') \\
[\![\nu\, x.P]\!]_{\pi}(l) & = & \nu\, x.\mathbf{NU}(l, x) \,\&\, [\![P]\!]_{\pi}(l) \\
[\![0]\!]_{\pi}(l) & = & \emptyset
\end{array}$$

Emission, reception and replicated reception, and the scope operator are encoded by new multi-relations. The null process and parallel composition are directly translating using the corresponding construction in the edition calculus. The operator ν of the edition calculus, which introduces new variables in the production of a rule and restricts scopes in a solution, allows the binders of the π-calculus to be encoded: it is used for reception and scope restriction. Thus, the structural congruence of the π-calculus are directly derived from the structural congruence of the edition calculus.

Note that in the definition of $[\![P]\!]_{\pi}(l)$, the variable l can be seen as a location that relates processes that are in parallel. In particular, two processes can communicate only if they are defined at the same location. The fresh variable l' in the translation of receptions can be seen as a continuation location. We assume that the set of variables used for locations is disjoint from

P	$::= \bar{x}\langle y\rangle$	Emission
	$\mid x(y).P$	Reception
	$\mid P \mid P$	Parallel composition
	$\mid \nu\, x.P$	Scope restriction
	$\mid\ !\, x(y).P$	Duplicated reception
	$\mid 0$	Null process

Table 6. Syntax of the asynchronous π-calculus

the set of variables used for channels. Note also that this translation could as well be performed directly in the edition calculus by defining a script that consists in the equalities above oriented from left to right and instantiated for each possible sub-process of the process to be translated (the size of the script remains finite in that case). For simplicity, we have chosen an equational presentation.

We now define the script s_π simulating communications in the asynchronous π-calculus. We first need to define some relations used to compute subtitutions and replications and to control execution:

- $\mathrm{SUB}(x, y), \mathrm{MOV}(l, l')$: relations respectively used to substitute x with y and to move processes at location l to location l'
- $\mathrm{CLO}(l, l'), \mathrm{SUBC}(l', x, y)$: relations respectively used to copy processes at location l to location l' and to substitute x with y in processes at location l'
- start: relation used as a token to start a reduction

The two reductions for communication are translated into the script s_{comm} that consists in the two following rules in parallel.

$$\begin{aligned}
&\mathrm{start} \,\&\, \mathbf{R}(l_0, x, y, l_1) \,\&\, \mathbf{E}(l_0, x, z) \quad \triangleright\ \mathrm{SUB}(y, z) \,\&\, \mathrm{MOV}(l_1, l_0)\\
&\mathrm{start} \,\&\, \mathbf{RR}(l_0, x, y, l_1) \,\&\, \mathbf{E}(l_0, x, z) \ \triangleright\ \nu l_1'.\nu y'.\mathrm{SUB}(y', z) \,\&\, \mathrm{MOV}(l_1', l_0)\\
&\qquad\qquad \&\, \mathbf{RR}(l_0, x, y, l_1) \,\&\, \mathrm{CLO}(l_1, l_1') \,\&\, \mathrm{SUBC}(l_1', y, y')
\end{aligned}$$

We now come to the technical details for substitutions and replications. We will resort to a generic garbage collector script, $\mathrm{GC}(R)$, which removes relation R from the solution.

$$(R(\overrightarrow{x}) \triangleright \emptyset)^\omega$$

Substitutions are defined by the following rules, defining the script r_{subst} when they are put in parallel.

$$\begin{aligned}
&\mathrm{SUB}(x, y) \,\&\, \mathbf{E}(l, x, z) \quad \triangleright\ \mathrm{SUB}(x, y) \,\&\, \mathbf{E}(l, y, z)\\
&\mathrm{SUB}(x, y) \,\&\, \mathbf{E}(l, z, x) \quad \triangleright\ \mathrm{SUB}(x, y) \,\&\, \mathbf{E}(l, z, y)\\
&\mathrm{SUB}(x, y) \,\&\, \mathbf{R}(l, x, z, l') \quad \triangleright\ \mathrm{SUB}(x, y) \,\&\, \mathbf{R}(l, y, z, l')\\
&\mathrm{SUB}(x, y) \,\&\, \mathbf{RR}(l, x, z, l') \ \triangleright\ \mathrm{SUB}(x, y) \,\&\, \mathbf{RR}(l, y, z, l')
\end{aligned}$$

Let us comment on these rules. For reception $\mathbf{R}(l, x, z, l')$ or $\mathbf{RR}(l, x, z, l')$, there is no rule for substituting the variable z as this variable is always fresh thanks to scope restriction. Likewise,

there is no rule for scope restriction $\mathbf{NU}(l,x)$. A communication generates also a relation $\mathrm{MOV}(l,l')$ that shifts one location up the sub-process involved in the communication. The shifting is defined by the following rules, defining the script r_{move} when they are put in parallel.

$$\begin{array}{ll}
\mathrm{MOV}(l,l') \,\&\, \mathbf{E}(l,x,y) & \triangleright\ \mathrm{MOV}(l,l') \,\&\, \mathbf{E}(l',x,y) \\
\mathrm{MOV}(l,l') \,\&\, \mathbf{R}(l,x,y,l'') & \triangleright\ \mathrm{MOV}(l,l') \,\&\, \mathbf{R}(l',x,y,l'') \\
\mathrm{MOV}(l,l') \,\&\, \mathbf{RR}(l,x,y,l'') & \triangleright\ \mathrm{MOV}(l,l') \,\&\, \mathbf{RR}(l',x,y,l'') \\
\mathrm{MOV}(l,l') \,\&\, \mathbf{NU}(l,x) & \triangleright\ \mathrm{MOV}(l,l') \,\&\, \mathbf{NU}(l',x)
\end{array}$$

Finally, we get the following script, s_{subst}, to perform substitutions and upward shifts:

$$(r_{\mathrm{subst}}, r_{\mathrm{move}})^{\omega} ;(\mathrm{GC}(\mathrm{SUB}), \mathrm{GC}(\mathrm{MOV}))$$

To implement replication, we need to resort to the same technique as the one used in Section 2.2 for cloning. Let $\mathbf{E_O}, \mathbf{R_O}, \mathbf{RR_O}$ and $\mathbf{NU_O}$ be the four multi-relations that will be used as original copies of $\mathbf{E}, \mathbf{R}, \mathbf{RR}$ and $\mathbf{NU}$ respectively. From each new multi-relation $M_{\mathbf{O}}$ that is the original copy of M, we will restore the original relation by using the following generic script, denoted $\mathrm{MV}(M)$.

$$(M_O(\overrightarrow{x}) \triangleright M(\overrightarrow{x}))^{\omega}$$

The replication is defined by the following rules, defining the script $r_{\mathrm{replication}}$ when they are put in parallel.

$$\begin{array}{ll}
\mathrm{CLO}(l,l') \,\&\, \mathbf{E}(l,x,y) & \triangleright\ \mathrm{CLO}(l,l') \,\&\, \mathbf{E_O}(l,x,y) \,\&\, \mathbf{E}(l',x,y) \\
\mathrm{CLO}(l_1,l_1') \,\&\, \mathbf{R}(l_1,x,y,l_2) & \triangleright\ \nu l_2'.\nu y'.\mathrm{CLO}(l_1,l_1') \,\&\, \mathbf{R_O}(l_1,x,y,l_2) \\
 & \quad \&\, \mathbf{R}(l_1',x,y',l_2') \,\&\, \mathrm{CLO}(l_2,l_2') \,\&\, \mathrm{SUBC}(l_2',y,y') \\
\mathrm{CLO}(l_1,l_1') \,\&\, \mathbf{RR}(l_1,x,y,l_2) & \triangleright\ \nu l_2'.\nu y'.\mathrm{CLO}(l_1,l_1') \,\&\, \mathbf{RR_O}(l_1,x,y,l_2) \\
 & \quad \&\, \mathbf{RR}(l_1',x,y',l_2') \,\&\, \mathrm{CLO}(l_2,l_2') \,\&\, \mathrm{SUBC}(l_2',y,y') \\
\mathrm{CLO}(l,l') \,\&\, \mathbf{NU}(l,x) & \triangleright\ \nu x'.\mathrm{CLO}(l,l') \,\&\, \mathbf{NU_O}(l,x) \\
 & \quad \&\mathbf{NU}(l',x') \,\&\, \mathrm{SUBC}(l',x,x') \\
\mathrm{SUBC}(l,x,y) \,\&\, \mathbf{E}(l,x,z) & \triangleright\ \mathrm{SUBC}(l,x,y) \,\&\, \mathbf{E}(l,y,z) \\
\mathrm{SUBC}(l,x,y) \,\&\, \mathbf{E}(l,z,x) & \triangleright\ \mathrm{SUBC}(l,x,y) \,\&\, \mathbf{E}(l,z,y) \\
\mathrm{SUBC}(l,x,y) \,\&\, \mathbf{R}(l,x,z,l') & \triangleright\ \mathrm{SUBC}(l,x,y) \,\&\, \mathbf{R}(l,y,z,l') \\
\mathrm{SUBC}(l,x,y) \,\&\, \mathbf{RR}(l,x,z,l') & \triangleright\ \mathrm{SUBC}(l,x,y) \,\&\, \mathbf{RR}(l,y,z,l') \\
\mathrm{SUBC}(l,x,x') \,\&\, \mathbf{R}(l,y,z,l') & \triangleright\ \mathrm{SUBC}(l,x,x') \,\&\, \mathbf{R}(l,y,z,l') \,\&\, \mathrm{SUBC}(l',x,x') \\
\mathrm{SUBC}(l,x,x') \,\&\, \mathbf{RR}(l,y,z,l') & \triangleright\ \mathrm{SUBC}(l,x,x') \,\&\, \mathbf{RR}(l,y,z,l') \,\&\, \mathrm{SUBC}(l',x,x')
\end{array}$$

The first subset of rules deals with cloning. A rule produces an original copy and a clone at a new location, propagates cloning in the case of reception, and introduces new variables when they are bound in the original atom, and the associated substitutions. The second subset deals with substitutions. Contrary to the preceding version of substitutions, the new version is localized: it only applies to a location and its sub-locations. Let us comment on the rules for substitutions. For reception $\mathbf{R}(l,x,z,l')$ or $\mathbf{RR}(l,x,z,l')$ or scope restriction $\mathbf{NU}(l,z)$. , there is no rule for substituting the variable z as this variable is always fresh thanks to scope restriction. For reception, the first rules perform the substitution for the reception channel whereas the last

rules propagate the substitution in processes that are below the reception. Finally, we get the following script, $s_{\text{replication}}$, to perform replications:

$$r_{\text{replication}}{}^{\omega} ; \big(\text{GC}(\text{CLO}) , \text{GC}(\text{SUBC}) , \text{MV}(\mathbf{E}) , \text{MV}(\mathbf{R}) , \text{MV}(\mathbf{RR}) , \text{MV}(\mathbf{NU})\big)$$

We can now conclude: the script s_π that simulates the reduction in the π-calculus is defined by

$$s_\pi = (\emptyset \triangleright \text{start}) ; \big(s_{\text{comm}} ; s_{\text{replication}} ; s_{\text{subst}} ; (\emptyset \triangleright \text{start})\big)^{\omega}.$$

We have left out the details that allow us to actually prove the following propositions, but we can still state the soundness and completeness of our translation.

Proposition 1 (Correctness of the translation of the π-calculus).
For any reduction $P_1 \rightarrow^* P_2$ of the π-calculus, there exists an execution

$$\emptyset \triangleright \nu l.[\![P_1]\!]_\pi(l) ; s_\pi \vdash \emptyset \quad \Rightarrow^* \quad s_\pi \vdash \nu l.[\![P_2]\!]_\pi(l).$$

Furthermore, if P_2 is in normal form, then the execution is complete.

Proposition 2 (Completeness of the translation of the π-calculus).
For any execution

$$\emptyset \triangleright \nu l.[\![P_1]\!]_\pi(l) ; s_\pi \vdash \emptyset \quad \Rightarrow^* \quad s_\pi \vdash \sigma,$$

there exists a process P_2 of the π-calculus such that $\sigma = \nu l.[\![P_2]\!]_\pi(l)$ and $P_1 \rightarrow^* P_2$. Furthermore, if the execution is complete, then P_2 is in normal form.

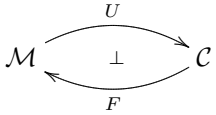
5 Logical foundations: computations as focalized proofs

5.1 Using resource modalities

As promised in the introduction, we now give an interpretation of complete execution in the edition calculus in terms of focused proof search in a fragment of linear logic. Actually, this is not exactly a fragment has we need to introduce other resource modalities than the exponential modality.

The exponential modality has been introduced by Girard [10] to make explicit the ability to erase or duplicate a formula in linear logic. It appears that a distinction between linear and classical formulas is not enough to interpret the edition calculus in linear logic. Indeed, every rule of a script s is basically linear but it can be affine when present in a replicated script s^ω, which corresponds to an infinite script that can be used as much as needed. Looking at molecules, a multi-relation is linearly consumes by a program, whereas a relation can be contracted (see the duplicate elimination of Table 4), but never erased or duplicated.

This indicates that we need various notions of resource modalities to make explicit those differences. Let us make a small categorical digression and explain how resource modalities are defined in [13]. A resource modality is defined in a categorical setting as a monoidal adjunction



between a linear (symmetric monoidal) category $\mathcal{C}$ that represents the linear part of the system and a (symmetric monoidal) category $\mathcal{M}$ that represents the structure induced by the resource modality $U \circ F$ on linear terms. Now, a resource modality is called

- *affine* when the unit of the tensor product of $\mathcal{M}$ is the terminal object of the category $\mathcal{M}$,
- *contractible* when each object A of $\mathcal{M}$ is equipped with a multiplication

$$A \otimes A \to A$$

  subject to coherence laws.
- *relevant* when each object A of $\mathcal{M}$ is equipped with a duplication

$$A \to A \otimes A$$

  subject to coherence laws.
- *exponential* when the tensor product of $\mathcal{M}$ is a cartesian product, and its unit is the terminal object of the category $\mathcal{M}$.

From a logical point of view, this categorical definition implies that every resource modality has a dereliction and promotion rule. Specifically, an affine modality $!_w$ supports weakening

$$\frac{\Gamma \vdash B}{\Gamma, !_w A \vdash B}$$

a contractible modality $!_d$ supports duplication

$$\frac{\Gamma, !_d A \vdash B}{\Gamma, !_d A, !_d A \vdash B}$$

and a relevant modality $!_c$ supports contraction

$$\frac{\Gamma, !_c A, !_c A \vdash B}{\Gamma, !_c A \vdash B}$$

An exponential modality can then be seen as the composition of an affine and a relevant modality. Note that we use Girard's terminology and read a rule from top to down (and name it accordingly) whereas the categorical interpretation is the other way around. This is why a contractible modality supports duplication and not contraction.

In the sequel, we will use the affine, contractible and relevant resource modalities to interpret reaction and solutions of our edition calculus.

5.2 The focused proof system

We now present a focused proof system inspired by Andreoli's original Σ_3 system for linear logic [1]. On the contrary to existing works on focusing for linear logic programing, we are not only interested in the simulation of executions by derivations, but we also want to characterized derivations that corresponds to complete executions. We thus have to introduce priorities between rules of the sequent calculus. Note that this notion of priorities breaks cut elimination. Therefore, we introduce a new connector

$$\alpha_1 \mathbin{©} \alpha_2$$

which explicitly introduces a cut where A appears in the context of the left premise of the $\mathbf{Cut}$ rule and B appears in the context of the right premise.

Formulas. We present bilateral formulation of focusing. Hence, we distinguish *positive formulas* which implies a commitment on the right (that is at a positive position) from *negative* formulas which implies a commitment on the left (that is at a negative position).

Positive connectives	$1, \otimes, \exists$
Negative connectives	$\multimap, \forall, ©$

All the connectors are standard except for resource modalities and © which explicitly introduces a cut in the proof. This is unusual in linear logic but is justified in our setting because we consider maximal proofs which do not support cut elimination. Note that we do not give a polarity to the resource modalities $\overset{!}{_w}$, $\overset{!}{_d}$ and $\overset{!}{_c}$.

In what follows, we consider sequent where the context is on the left. Thus, positive formulas will not imply commitment and will be immediately decomposed.

A focused sequent calculus. We now present the focused sequent calculus. We only introduce rules that will be relevant in the proof search corresponding to the execution of a program of the edition calculus. For example, we omit the right introduction of the existential connector as it will always appear on the left. For the same reason, we do not provide Promotion for affine and relevant modalities as a program only appears on the left of a sequent during the proof. We also omit Promotion and Dereliction for the contractible modality. Indeed, a relation – which is the only kind of formula that contains a contractible modality – can only be introduced by an axiom and the use of Dereliction would introduced a multi-relation whose name is in $\mathcal{R}$ and thus will never be consumed by any rule of the script.

The definition of the sequent calculus, given in Table 7, has been stratified in five layers. (1) The first layer corresponds to the introduction of a focused formula on the left and on the right. (2) The second layer corresponds to rules where there is a focused formula on the left. (3) The third layer corresponds to rules where there is a focused formula on the right. (4) The fourth layer corresponds to positive structural rules that must be applied as soon as possible. (5) The fifth layer corresponds to structural rules for affine and relevant polarity.

Maximal, contracted and irreversible proofs. To describe a complete execution of a script with our focused semantics, we need to restrict the proof search in three ways. First, to simulate the priority of duplicated elimination, we restrict the proof search to contracted derivation.

Definition 1 (Contracted derivation). *A derivation is* contracted *when the positive structural rules (defined in the fourth layer of Table 7) have been used primarily to any other rule.*

Second, we need to prevent the proof search from applying a rule that corresponds to a reduction *but not to a progression*. Such a undesirable rule is formalized in the definition of reversible derivation below.

Definition 2 (reversible derivation). *The rule* $\multimap \mathbf{G}$ *is said to be* reversible *if the sequent* $\Delta_1, \Delta_2; \vdash \sigma'$ *can be deduced from the sequent* $\Delta_2, \alpha; \vdash \sigma'$ *using only positive structural rules.*

This definition means that applying the rule $\sigma \multimap \alpha$ has not changed the solution. Indeed, the reaction consumes Δ_1 and produces α which, up-to structural rules is equivalent to Δ_1.

Finally, we need to introduce a notion of priority of the focus on the left with respect to the focus on the right. This means that the proof search must try to apply rules coming from the script as much as possible before focusing on the right and thus ending the proof with axioms.

Definition 3 (Maximal derivation). *A derivation is* maximal *when the rule* $\mathbf{Focus\text{-}D}$ *is applied only when the rule* $\mathbf{Focus\text{-}G}$ *would fail.*

$$\frac{\Delta;[\alpha]\vdash\sigma}{\Delta,\alpha;\vdash\sigma}\ [\mathbf{Focus\text{-}G}] \qquad \frac{\Delta;\vdash[\sigma]}{\Delta;\vdash\sigma}\ [\mathbf{Focus\text{-}D}]$$

$$\frac{\Delta_1;\vdash[\sigma] \qquad \Delta_2,\alpha;\vdash\sigma'}{\Delta_1,\Delta_2;[\sigma\multimap\alpha]\vdash\sigma'}\ [\multimap\mathbf{G}] \qquad \frac{\Delta;[\alpha]\vdash\sigma}{\Delta;[\forall x.\alpha]\vdash\sigma}\ [\forall\mathbf{G}]$$

$$\frac{\Delta_1,\alpha_1;\vdash\sigma_1 \qquad \Delta_2,\sigma_1,\alpha_2;\vdash\sigma}{\Delta_1,\Delta_2;[\alpha_1\copyright\alpha_2]\vdash\sigma}\ [\mathbf{Cut}]$$

$$\frac{\Delta_1;\vdash[\sigma_1] \qquad \Delta_2;\vdash[\sigma_2]}{\Delta_1,\Delta_2;\vdash[\sigma_1\otimes\sigma_2]}\ [\otimes\mathbf{D}] \qquad \frac{}{;\vdash[1]}\ [\mathbf{1D}] \qquad \frac{}{\alpha;\vdash[\alpha]}\ [\mathbf{Axiom}]$$

$$\frac{\Delta,\ !_d a;\vdash\sigma}{\Delta,\ !_d a,\ !_d a;\vdash\sigma}\ [\mathbf{Duplication}] \qquad \frac{\Delta;\sigma_1,\sigma_2;\vdash\sigma}{\Delta,\sigma_1\otimes\sigma_2;\vdash\sigma}\ [\otimes\mathbf{G}]$$

$$\frac{\Delta;\vdash\sigma}{\Delta,1;\vdash\sigma}\ [\mathbf{1G}] \qquad \frac{\Delta,\alpha;\vdash\sigma}{\Delta,\exists x.\alpha;\vdash\sigma}\ (x\notin \mathrm{FV}(\Delta))\ [\exists\mathbf{G}]$$

$$\frac{\Delta;\vdash\sigma}{\Delta,\ !_w a;\vdash\sigma}\ [\mathbf{Weakening}] \qquad \frac{\Delta,\ !_c a,\ !_c a;\vdash\sigma}{\Delta,\ !_c a;\vdash\sigma}\ [\mathbf{Contraction}]$$

$$\frac{\Delta,a;\vdash\sigma}{\Delta,\ !_w a;\vdash\sigma}\ [!_w\mathbf{Dereliction}] \qquad \frac{\Delta,a;\vdash\sigma}{\Delta,\ !_c a;\vdash\sigma}\ [!_c\mathbf{Dereliction}]$$

Table 7. The focused proof system

5.3 Translation of chemical configuration

Given a chemical configuration $\rho \vdash \sigma$, we define by induction a sequent $[\![\rho]\!] \vdash [\![\sigma]\!]$ of linear logic:

- Atom:
$$[\![R\,(\overrightarrow{v})]\!] = !_d R\,(\overrightarrow{v}) \quad \text{and} \quad [\![M\,(\overrightarrow{v})]\!] = M\,(\overrightarrow{v})$$
- Molecule:
$$[\![\emptyset]\!] = 1 \quad \text{and} \quad [\![a]\!] = a \quad \text{and} \quad [\![j_1\ \&\ j_2]\!] = [\![j_1]\!] \otimes [\![j_2]\!]$$
- Rule:
$$[\![j_1 \triangleright \overrightarrow{\nu v}.j_2]\!] = \left(\forall\,\overrightarrow{x}\,.\,[\![j_1]\!] \multimap \exists\,\overrightarrow{v}\,.\,[\![j_2]\!]\right) \qquad (\overrightarrow{x} = \mathrm{FV}(j_1))$$
- Script:
$$[\![s_1\,,\,s_2]\!] = [\![s_1]\!] \otimes [\![s_2]\!] \quad \text{and} \quad [\![s_1\,;\,s_2]\!] = [\![s_1]\!]\,©\,[\![s_2]\!] \quad \text{and} \quad [\![s^\omega]\!] = !_c[\![s]\!]_w$$
$$[\![s_1\,,\,s_2]\!]_w = !_w[\![s_1]\!]_w \otimes !_w[\![s_2]\!]_w \quad \text{and} \quad [\![s_1\,;\,s_2]\!]_w = !_w([\![s_1]\!]_w\,©\,[\![s_2]\!]_w) \quad \text{and} \quad [\![s^\omega]\!]_w = !_c[\![s]\!]_w$$
- Solution:
$$[\![\sigma_1\,,\,\sigma_2]\!] = [\![\sigma_1]\!] \otimes [\![\sigma_2]\!] \quad \text{and} \quad [\![\nu\,v.\sigma]\!] = \exists v\,.\,[\![\sigma]\!]$$

Remark that the different resource modalities are fully exploited in the translation. A multi-relation is translated by a linear formula, a relation by a contractible formula, a replicated reaction by an relevant formula and a reaction under a replicated reaction as an affine formula.

We can now state the soundness and (non-deterministic) completeness of the edition calculus with respect to our fragment of focused linear logic.

Proposition 3 (Soundness of operational semantics). *For any execution $\rho_1 \vdash \sigma_1 \Rightarrow^* \rho_2 \vdash \sigma_2$, there exists a contracted and irreversible proof of $[\![\rho_1]\!] \otimes [\![\sigma_1]\!] \vdash [\![\rho_2]\!] \otimes [\![\sigma_2]\!]$. Furthermore, if the execution is complete, this derivation is maximal.*

Proof. The proof goes by induction on the length of the execution. We show that each structural reduction rule gives rise to a contracted and irreversible derivation in our focused sequent calculus. For most of the rules, the interpretation is direct, as for example for Elimination which comes directly from rule $\mathbf{Duplication}$ of the contractible modality or for a reduction which comes directly from rule $\multimap \mathbf{G}$. Scope laws are standard properties of the existential. Let us just detail the validity of the replication law which corresponds to the logical equivalence

$$[\![\rho_1 \, , {\rho_2}^{\omega}]\!] = !_c([\![\rho_1]\!] \otimes [\![\rho_2]\!]) \equiv !_c[\![\rho_1]\!] \otimes !_c[\![\rho_2]\!] = [\![{\rho_1}^{\omega} \, , {\rho_2}^{\omega}]\!].$$

Such a rule is not valid in general in linear logic (for the same reason that it is not true for the exponential), but we combine the fact that the translation of a reaction under a replication is always an affine formula with the following equivalence

$$!_c(!_w A \otimes !_w B) \equiv !_{cw} A \otimes !_{cw} B,$$

this time is valid in linear logic. The fact that the derivation is contracted and irreversible comes from the definition of progression.

Proposition 4 (Non-deterministic completeness of operational semantics). *If there exists a contracted and irreversible proof of the sequent $[\![\rho_1]\!] \otimes [\![\sigma_1]\!] \vdash \Delta$, then there exists an execution of the form $\rho_1 \vdash \sigma_1 \Rightarrow^* \rho_2 \vdash \sigma_2$ with $\Delta \equiv [\![\rho_2]\!] \otimes [\![\sigma_2]\!]$. Furthermore, if the proof is maximal, the execution is complete.*

Proof. The proof goes by induction on the focused derivation.

References

1. J. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297, 1992.
2. Jean-Pierre Banâtre and Daniel Le Métayer. The gamma model and its discipline of programming. *Science of Computer Programing*, 15(1):55–77, 1990.
3. Hariolf Betz and Thom W. Frühwirth. A linear-logic semantics for constraint handling rules. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2005.
4. Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, and Gianluigi Zavattaro. SCC: A Service Centered Calculus. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

5. Gérard Boudol. Asynchrony and the Pi-calculus. Research Report RR-1702, INRIA, 1992.
6. Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation*, 207(10):1044–1077, 2009.
7. Michel R. V. Chaudron and Edwin D. de Jong. Towards a compositional method for coordinating gamma programs. In Paolo Ciancarini and Chris Hankin, editors, *COORDINATION*, volume 1061 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 1996.
8. C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proc. of the 23rd symposium on POPL*, pages 372–385. ACM, 1996.
9. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
10. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
11. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP '91*, pages 133–147, London, UK, 1991. Springer-Verlag.
12. Ivan Lanese, Francisco Martins, Vasco Thudichum Vasconcelos, and António Ravara. Disciplining orchestration and conversation in service-oriented computing. In *SEFM*, pages 305–314. IEEE Computer Society, 2007.
13. Paul-André Melliès and Nicolas Tabareau. Resource modalities in tensor logic. *Annals of Pure and Applied Logic*, 161(5):632–653, February 2010.
14. Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *WWW*, pages 805–814, 2008.
15. Robert J. Simmons and Frank Pfenning. Linear logical algorithms. In *Proc. of the 35th International Colloquium, ICALP*, pages 336–347, 2008.
16. Allen Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.
17. Hugo T. Vieira, Luís Caires, and João C. Seco. The conversation calculus: a model of service oriented computation. In *Proc. of ESOP'08, LNCS*. Springer, 2008.