Projet PiCoq

Deliverable D123

December 2011

This deliverable includes the following articles describing work done on tasks 1, 2, and 3 during the year 2011.

Task 1 Mathematical frameworks for local/abstract reasoning

- "Innocent strategies as presheaves and interactive equivalences for CCS" by Tom Hirschowitz and Damien Pous.
- "Untyping typed algebras and colouring cyclic linear logic" by Damien Pous.

Task 2 Process Calculi for Distributed Components

- "Revisiting glue expressiveness in component-based systems" by Cinzia Di Giusto and Jean-Bernard Stefani.
- "Controlling Reversibility in Higher-Order Pi" by Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani.

Task 3 Proof Techniques

- "Termination in a π-calculus with Subtyping" by Ioana Cristescu and Daniel Hirschkoff.
- "Strong Normalisation in λ-calculi with References" by Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi.

Innocent strategies as presheaves and interactive equivalences for CCS

Tom HIRSCHOWITZ[1] and Damien POUS[2]

Abstract

Seeking a general framework for reasoning about and comparing programming languages, we derive a new view of Milner's CCS [33]. We construct a category $\mathbb{E}$ of *plays*, and a subcategory $\mathbb{V}$ of *views*. We argue that presheaves on $\mathbb{V}$ adequately represent *innocent* strategies, in the sense of game semantics [20]. We equip innocent strategies with a simple notion of interaction.

We then prove decomposition results for innocent strategies, and, restricting to presheaves of finite ordinals, prove that innocent strategies are a final coalgebra for a polynomial functor [27] derived from the game. This leads to a translation of CCS with recursive equations.

Finally, we propose a notion of *interactive equivalence* for innocent strategies, which is close in spirit to Beffara's interpretation [1] of testing equivalences [7] in concurrency theory. In this framework, we consider analogues of *fair* testing and *must* testing. We show that must testing is strictly finer in our model than in CCS, since it avoids what we call 'spatial unfairness'. Still, it differs from fair testing, and we show that it coincides with a relaxed form of fair testing.

Note: This is an expanded version of our ICE '11 paper [19]. It notably simplifies a few aspects of the development, and corrects the mistaken statement that fair and must testing coincide in our semantic framework. Must testing only coincides with a relaxed variant of fair testing. This version also subsumes a previous preprint, providing more compact proofs.

1 Overview

Theories of programming languages Research in programming languages is mainly technological. Indeed, it heavily relies on techniques which

[1]CNRS, Université de Savoie, France, ttom.hirschowitz@univ-savoie.fr
[2]CNRS, Laboratoire d'Informatique de Grenoble, France, tdamien.pous@inria.fr

are ubiquitous in the field, but almost never formally made systematic. Typically, the definition of a language then quotiented by variable renaming (α-conversion) appears in many theoretical papers about functional programming languages. Why isn't there yet any abstract framework performing these systematic steps for you? Because the quest for a real *theory* of programming languages is not achieved yet, in the sense of a corpus of results that actually help developing them or reasoning about them. However, many attempts at such a theory do exist.

A problem for most of them is that they do not account for the dynamics of execution, which limits their range of application. This is for example the case of Fiore et al.'s second-order theories [10, 16, 17]. A problem for most of the other theories of programming languages is that they neglect denotational semantics, i.e., they do not provide a notion of model for a given language. This is for example the case of Milner et al.'s *bigraphs* [22], or of most approaches to *structural operational semantics* [36], with the notable exception of the *bialgebraic semantics* of Turi and Plotkin [40]. A recent, related, and promising approach is *Kleene coalgebra*, as advocated by Bonsangue et al. [2]. Finally, *higher-order rewriting* [35], and its semantics in double categories [12] or in cartesian closed 2-categories [18], is not currently known to adequately account for process calculi.

Towards a new approach The most relevant approaches to us are bialgebraic semantics and Kleene coalgebra, since the programme underlying the present paper concerns a possible alternative. A first difference, which is a bit technical but may be of importance, is that both bialgebraic semantics and Kleene coalgebra are based on labelled transition systems (LTSs), while our approach is based on reduction semantics. Reduction semantics is often considered more primitive than LTSs, and much work has been devoted to deriving the latter from the former [39, 22, 38, 37]. It might thus be relevant to propose a model based only on the more primitive reduction semantics.

More generally, our approach puts more emphasis on interaction between programs, and hence is less interesting in cases where there is no interaction. A sort of wild hope is that this might lead to unexpected models of programming languages, e.g., physical ones. This could also involve finding a good notion of morphism between languages, and possibly propose a notion of compilation. At any rate, the framework is not set up yet, so investigating the precise relationship with bialgebraic semantics and Kleene coalgebra is deferred to further work.

How will this new approach look like? Compared to such long-term goals, we only take a small step forward here, by considering a particular case, namely Milner's CCS [33], and providing a new view of it. This view borrows ideas from the following lines of research: game semantics [20], and in particular the notion of an *innocent strategy*, *graphical games* [8, 15], Krivine realisability [28], ludics [13], testing equivalences in concurrency [7, 1], the presheaf approach to concurrency [24, 25], and sheaves [31]. It is also, more remotely, related to graph rewriting [9] and computads [4].

From strategies to presheaves Game semantics [20] has provided fully complete models of programming languages. It is based on the notion of a *strategy*, i.e., a set of *plays* in some game, satisfying a few conditions. In concurrency theory, taking as a semantics the set of accepted plays, or 'traces', is known as *trace semantics*. Trace semantics is generally considered too coarse, since it equates, for a most famous example, the right and the wrong coffee machines, $a.(b+c)$ and $ab+ac$ [33].

An observation essentially due to Joyal, Nielsen, and Winskel is that strategies, i.e., prefix-closed sets of plays, are actually particular *presheaves of booleans* on the category $\mathbb{C}$ with plays as objects, and prefix inclusions as morphisms. By presheaves of booleans on $\mathbb{C}$ we here mean functors $\mathbb{C}^{op} \to 2$, where 2 is the preorder category $0 \leq 1$. If a play p is *accepted*, i.e., mapped to 1, then its prefix inclusions $q \hookrightarrow p$ are mapped to the unique morphism with domain 1, i.e., id_1, which entails that q is also accepted.

Following Joyal, Nielsen, and Winskel, we observe that considering instead presheaves (of sets) on $\mathbb{C}$ yields a much finer semantics. So, a play p is now mapped to a set $S(p)$, to be thought of as the set of ways for p to be accepted by the strategy S. Considering the set of players as a team, $S(p)$ may also be thought of as the set of possible *states* of the team after playing p – which is empty if the team never accepts to play p.

This presheaf semantics is fine enough to account for bisimilarity [24, 25]. Indeed, presheaves are essentially forests with edges labelled by moves. For example, in the setting where plays are finite words on an alphabet, the wrong coffee machine may be represented by the presheaf S defined by the equations on the left and pictured as on the right:

$$
\begin{array}{l}
S(\epsilon) = \{\star\}, \\
S(a) = \{x, x'\}, \\
S(ab) = \{y\}, \\
S(ac) = \{y'\},
\end{array}
\qquad
\begin{array}{l}
S(\epsilon \hookrightarrow a) = \{x \mapsto \star, x' \mapsto \star\}, \\
S(a \hookrightarrow ab) = \{y \mapsto x\}, \\
S(a \hookrightarrow ac) = \{y' \mapsto x'\} :
\end{array}
\qquad
\begin{array}{ccccc}
 & & a & \star & a & & \\
 & & x & & x' & & \\
b & | & & & & | & c \\
 & & y & & y'. & &
\end{array}
$$

So, in summary: the standard notion of strategy may be generalised to account for branching equivalences, by passing from presheaves of booleans to presheaves of sets.

Multiple players Traditional game semantics mostly emphasises two-player games. There is an implicit appearance of three-player games in the definition of composition of strategies, and of four-player games in the proof of its associativity, but these games are never given a proper status. A central idea of graphical games, and to a lesser extent of ludics, is the emphasis on multiple-player games.

Here, there first is a base category $\mathbb{B}$ of *positions*, whose objects represent configurations of players. Since the game represents CCS, it should be natural that players are related to each other via the knowledge of *communication channels*. So, roughly, positions are bipartite graphs with vertex sets *players* and *channels*, and edges from channels to players indicating when the former is known to the latter. As a first approximation, morphisms of positions may be thought of as just embeddings of such graphs.

Second, there is a category $\mathbb{E}$ of *plays*, with a functor to $\mathbb{B}$ sending each play to its initial position. Plays are represented in a more flexible way than just sequences of moves, namely using a kind of string diagrams. This echoes the idea [32] that two moves may be independent, and that plays should not depend on the order in which two independent moves are performed. Furthermore, our plays are a rather general notion, allowing, e.g., to focus on a given player. Morphisms of plays account both for:

- prefix inclusion, i.e., inclusion of a play into a longer play, and
- position enlargement, e.g., inclusion of information about some players into information about more players.

Now, restricting to plays above a given initial position X, and then taking presheaves on this category $\mathbb{E}_X$, we have a category of strategies on X.

Innocence A fundamental idea of game semantics is the notion of *innocence*, which says that players have a restricted *view* of the play, and that their actions may only depend on that view.

We implement this here by defining a subcategory $\mathbb{V}_X \hookrightarrow \mathbb{E}_X$ of *views* on X, and deeming a presheaf F on $\mathbb{E}_X$ *innocent* when it is determined by its restriction F' to $\mathbb{V}_X$, in the sense that it is isomorphic to the *right Kan extension* [30] of F' along $\mathbb{V}_X^{op} \hookrightarrow \mathbb{E}_X^{op}$.

Given this, it is sensible to define innocent strategies to be just presheaves on $\mathbb{V}_X$, and view them as strategies via the (essential) embedding $\widehat{\mathbb{V}_X} \hookrightarrow \widehat{\mathbb{E}_X}$ induced by right Kan extension.

Interaction For each position X, we thus have a category $\mathsf{S}_X = \widehat{\mathbb{V}_X}$ of innocent strategies. In game semantics, composition of strategies is achieved in two steps: *interaction* and *hiding*. Essentially, interaction amounts to considering the three-player game obtained by letting two two-player games interact at a common interface. Hiding then forgets what happens at that interface, to recover a proper two-player game.

We have not yet investigated hiding in our approach, but, thanks to the central status of multiple-player games, interaction is accounted for in a very streamlined way. For any position X with two subpositions $X_1 \hookrightarrow X$ and $X_2 \hookrightarrow X$ such that each player is in either X_1 or X_2, but none is in both, given innocent strategies $F_1 \in \mathsf{S}_{X_1}$ and $F_2 \in \mathsf{S}_{X_2}$, there is a unique innocent strategy, the *amalgamation* $[F_1, F_2]$ of F_1 and F_2, whose restrictions to X_1 and X_2 are F_1 and F_2.

Amalgamation in this sense models interaction in the sense of game semantics, and, using the correspondence with presheaves on $\mathbb{E}_X$ given by right Kan extension, it is the key to defining interactive equivalences.

CCS Next, we define a translation of CCS terms with recursive equations into innocent strategies. This rests on *spatial* and *temporal* decomposition results for innocent strategies. Spatial decomposition says that giving a strategy on a position X is the same as giving a strategy for each of its players. Temporal decomposition says that a strategy is determined up to isomorphism by its set of initial states, plus what remains of each of them after each basic move. Restricting to presheaves of finite ordinals, we also prove that innocent strategies form a final coalgebra for a polynomial functor (in the sense of Kock [27]) derived from the game, thus hinting at links with Kleene coalgebra. It is then easy to translate finite CCS into the language

induced by our polynomial functor, and to finally extend the translation to CCS with recursive equations via infinite unfolding.

A natural question is then: which equivalence does this translation induce on CCS terms? As explained in the following paragraph, we provide some preliminary results about interactive equivalences, but essentially leave the question open.

Interactive equivalences Returning to the development of our approach, we then define a notion of *interactive equivalence*, which is close in spirit to both testing equivalences in concurrency theory and Krivine realisability and ludics.

The game, as sketched above, allows interacting with players which are not part of the considered position. E.g., a player in the considered position X may perform an input which is not part of any synchronisation. A *test* for an innocent strategy F on X is then, roughly, an innocent strategy G on a position X' with the same channels as X. To decide whether F *passes* the test G, we consider a restricted variant of the game on the 'union' $X \cup X'$, forbidding any interaction with the outside. We call that variant the *closed-world* game.

Then F passes G iff the amalgamation $[F, G]$, right Kan extended to $\mathbb{E}_{X \cup X'}$ and then restricted to the closed-world game, belongs to some initially fixed class of strategies, $\perp\!\!\!\perp_{X \cup X'}$. Finally, two innocent strategies F and F' on X are equivalent when they pass the same tests.

Here are two examples for $\perp\!\!\!\perp$. Consider a *tick* move, fixed in advance. Then call *successful* all plays containing at least one tick, and accordingly call successful all states reached after a successful play. One may consider:

- $\perp\!\!\!\perp^m$, consisting of strategies whose maximal states (those that admit no strict extensions) are all successful; the tick move plays a rôle analogous to the daimon in ludics: it is the only move which is observable from the outside;
- $\perp\!\!\!\perp^f$, consisting of strategies in which all states on *finite* plays admit a successful extension.

From the classical concurrency theory point of view on behavioural equivalences, the first choice clearly mimicks *must* testing equivalence, while the second mimicks *fair* testing equivalence [34, 3].

Consider the processes Ω and $\Omega|\overline{a}$, where Ω is a process doing infinitely many silent transitions. These processes are intuitively quite different: the

latter can do an output on the channel a, while the former cannot. They are however equated by standard must testing equivalence: the infinite trace provided by Ω may prevent the output prefix from being performed. In fact, must testing equivalence heavily relies on the potential unfairness of the scheduler. In the literature, this peculiar behaviour actually motivates the introduction of fair testing equivalence.

In contrast, our notion of play is more flexible than standard traces, so that our counterpart to must testing equivalence actually distinguishes these two processes: the infinite play where the output prefix is not performed is not maximal, so that the corresponding unfair behaviour is not taken into account. In other words, thanks to our notion of play, the rather natural notion of must testing already avoids what we call 'spatial unfairness'. However, must testing does not coincide with fair testing in our setting, because there are other sources of unfairness, that are not handled properly. Technically, we prove that $\perp\!\!\!\perp^m$ coincides with the set of strategies whose states all admit a successful extension. However, the restriction to finite plays in the definition of $\perp\!\!\!\perp^f$ is required to rule out other sources of unfairness.

Summary In summary, our approach emphasises a flexible notion of multiple-player play, encompassing both views in the sense of game semantics, closed-world plays, and intermediate notions. Strategies are then described as presheaves on plays, while innocent strategies are presheaves on views. Innocent strategies admit a notion of interaction, or amalgamation, and are embedded into strategies via right Kan extension. This allows a notion of testing, or interactive equivalence by amalgamation with the test, right Kan extension, and finally restriction to closed-world.

Our main technical contributions are then a translation of CCS terms with recursive equations into innocent strategies, and the study of fair and must equivalences in our setting.

Perspectives Our next task is clearly to tighten the link with CCS. Namely, we should explore which equivalence on CCS is induced via our translation, for a given interactive equivalence. We will start with $\perp\!\!\!\perp^f$. Furthermore, the very notion of interactive equivalence might deserve closer consideration. Its current form is rather *ad hoc*, and one could hope to see it emerge more naturally from the game. For instance, the fixed class $\perp\!\!\!\perp$ of 'successful' strategies should probably be more constrained than is done here. Also, the paradigm of observing via the set of successful tests might

admit sensible refinements, e.g., probabilistic ones.

Another possible research direction is to tighten the link with 'graphical' approaches to rewriting, such as graph rewriting or computads. E.g., our plays might be presented by a computad [14], or be the bicategory of rewrite sequences up to shift equivalence, generated by a graph grammar in the sense of Gadducci et al. [11]. Both goals might require some technical adjustments, however. For computads, we would need the usual yoga of U-turns to flexibly model our positions; e.g., zigzags of U-turns are usually only equal up to a higher-dimensional cell, while they would map to equal positions in our setting. For graph rewriting, the problem is that our positions are not exactly graphs (e.g., the channels known to a player are linearly ordered).

Other perspectives include the treatment of more complicated calculi like π or λ. In particular, calculi with duplication of terms will pose a serious challenge. An even longer-term hope is to be able to abstract over our approach. Is it possible to systematise the process starting from a calculus as studied in programming language theory, and generating its strategies modulo interactive equivalence? If this is ever understood, the next question is: when does a translation between two such calculi preserve a given interactive equivalence? Finding general criteria for this might have useful implications in programming languages, especially compilation.

Notation Throughout the paper, we abusively identify n with $\{1 \dots n\}$, for readability. So, e.g., $i \in n$ means $i \in \{1, \dots, n\}$.

The various categories and functors constructed in the development are summed up with a short description in Table 1. There, given two functors $\mathbb{C} \xrightarrow{F} \mathbb{E} \xleftarrow{G} \mathbb{D}$, we denote (slightly abusively) by $\mathbb{C} \downarrow_{\mathbb{E}} \mathbb{D}$ the *comma* category: it has as objects triples (C, D, u) with $C \in \mathbb{C}$, $D \in \mathbb{D}$, and $u \colon F(C) \to G(D)$ in $\mathbb{E}$, and as morphisms $(C, D, u) \to (C', D', u')$ pairs (f, g) making the square above commute. Also, when F is the identity on $\mathbb{C}$ and $G \colon 1 \to \mathbb{C}$ is an object C of $\mathbb{C}$, this yields the usual *slice* category, which we abbreviate as $\mathbb{C}/C$. Finally, the category of presheaves on any category $\mathbb{C}$ is denoted by $\widehat{\mathbb{C}} = [\mathbb{C}^{op}, \mathsf{Set}]$.

$$\begin{array}{ccc} FC & \xrightarrow{F(f)} & FC' \\ u\downarrow & & \downarrow u' \\ GD & \xrightarrow[G(g)]{} & GD' \end{array}$$

Furthermore, for any category $\mathbb{C}$, we denote by $\mathrm{ob}(\mathbb{C})$ its set of objects. For any functor $F \colon \mathbb{C} \to \mathbb{D}$, we denote by $\mathbb{F}^{op} \colon \mathbb{C}^{op} \to \mathbb{D}^{op}$ the functor induced on opposite categories, defined exactly as F on both objects and morphisms. Also, recall that an *embedding* of categories is an injective-on-objects, faithful functor. This admits the following generalisation: a functor

Category	Description of its objects
$\widehat{\mathbb{C}}$	'diagrams'
$\mathbb{B} \hookrightarrow \widehat{\mathbb{C}}$	positions
$\mathbb{E} \hookrightarrow (\mathbb{B} \downarrow_{\widehat{\mathbb{C}}} \widehat{\mathbb{C}})$	plays
$\mathbb{E}_X = (\mathbb{E} \downarrow_{\mathbb{B}} (\mathbb{B}/X))$	plays on a position X
$\mathbb{V}_X \hookrightarrow \mathbb{E}_X$	views on X
$\mathsf{S}_X = \widehat{\mathbb{V}_X}$	innocent strategies on X
$\mathbb{W} \hookrightarrow \mathbb{E}$	closed-world plays
$\mathbb{W}(X)$	closed-world plays on X

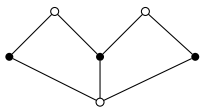
Table 1: Summary of categories and functors

$F\colon \mathbb{C} \to \mathbb{D}$ is *essentially injective on objects* when $FC \cong FC'$ implies $C \cong C'$. Any faithful, essentially injective on objects functor is called an *essential embedding*.

2 Plays as string diagrams

We now describe our approach more precisely, starting with the category of multiple-player plays. For the sake of clarity, we first describe this category in an informal way, before giving the precise definition (Section 3).

2.1 Positions



Since the game represents CCS, it should be natural that players are related to each other via the knowledge of *communication channels*. This is represented by a kind of[3] finite, bipartite graph: an example position is on the right. Bullets represent players, circles represent channels, and edges indicate when a player knows a channel. The channels known by a player are linearly ordered. Formally, as explained in Section 3, positions are presheaves over a certain category $\mathbb{C}_1$. Morphisms of positions are natural transformations, which are roughly morphisms of graphs, mapping players to players and channels to channels. In full generality, morphisms thus do not have to be

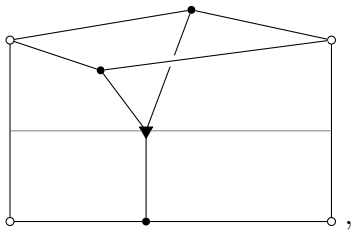
[3]Only 'a kind of', because, as mentioned above, the channels known to a player are linearly ordered.

injective, but include in particular embeddings of positions in the intuitive sense. Positions and morphisms between them form a category $\mathbb{B}$.

2.2 Moves

Plays will be defined as glueings of *moves* between positions. Moves are derived from the very definition of CCS, as we now sketch. The diagrams we draw in this section will be given a very precise combinatorial definition in Section 3.

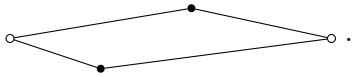
Let us start with the *forking* move, which corresponds to parallel composition in CCS: a process (the player) forks into two sub-processes. In the case of a player knowing two channels, the forking move is represented by the diagram



(1)

to be thought of as a move from the bottom position X

(with one player p) to the top position Y



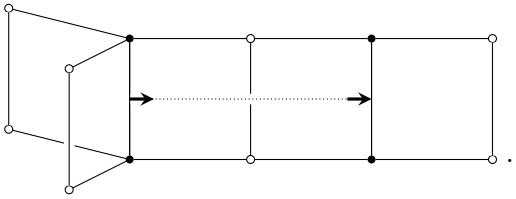
(with two players, which we call the 'avatars' of p). The left- and right-hand borders are just channels evolving in time, not noticing that the represented player forks into two. The surfaces spread between those vertical lines represent links (edges in the involved positions) evolving in time. For example, each link here divides into two when the player forks, thus representing the fact that both of the avatars retain knowledge of the corresponding channel. There is of course an instance π_n of forking for each n, according to the number of channels known to the player. As for channels known to a player, the players and channels touching the black triangle are ordered: there are different 'ports' for the initial player and its two avatars.

We then have a *tick* move $\heartsuit_n$, whose role is to define successful plays, and a move for the *channel creation* or *restriction* of CCS, here ν_n. In the case where the player knows two channels, they are graphically represented as

and

respectively. As expected, there is an instance of each of these two moves for each number n of channels known to the player.

We also need a move to model CCS-like synchronisation, between two players. For all n and m, representing the numbers of channels known to the players involved in the synchronisation, and for all $i \in n$, $j \in m$, there is a *synchronisation* $\tau_{n,i,m,j}$, represented, in the case where one player outputs on channel $3 \in 3$ and the other inputs on channel $1 \in 2$, by



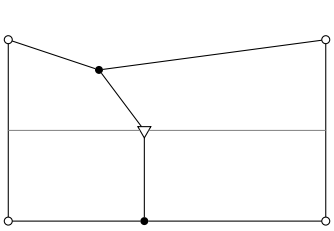
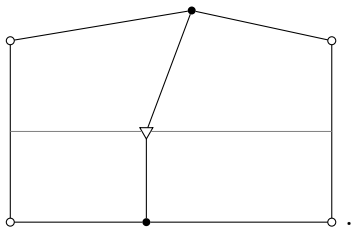
As we shall see in Section 3, the dotted wire in the picture is actually a point in the formal representation (i.e., an element of the corresponding presheaf).

The above four kinds of moves (forking, tick, channel creation, and synchronisation) come from the reduction semantics of CCS. We classify these as *closed-world* moves, since they correspond to the evolution of a group of players in isolation.

We however need a more fine-grained structure for moves: moves whose final position has more than one player (forking and synchronisation) must be decomposed into *basic* moves, to get an appropriate notion of view.

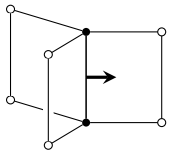
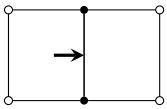
We introduce two sub-moves for forking: *left* and *right half-forking*. In the case where the player knows two channels, they are represented by the

following diagrams, respectively:


and

. (2)

These sub-moves represent what each of the 'avatars' of the forking player sees of the move. We call π_n^l and π_n^r the respective instances of the left-hand and right-hand basic moves for a player knowing n channels. Formally, there will be injections from the left and right half-forking moves to the corresponding forking moves.

We finally decompose synchronisation into an input move and an output move: $a.P$ and $\overline{a}.P$ in CCS become $\iota_{n,i}^+$ and $\iota_{n,i}^-$ here (where n is the number of known channels, $i \in \{1 \dots n\}$ is the index of the channel bearing the synchronisation). Here, output on the right-hand channel and input on the left-hand channel respectively look like


and

. (3)

Like with forking, there will be injections from the input and output moves to the corresponding synchronisation moves.

All in all, there are three classes of moves, which we summarise in Table 2.

- Tick, channel creation, half-forking, and input/output moves are *basic* moves: they evolve from a position with exactly one player to another position with exactly one player. These moves are used to define views later on.
- Forking, synchronisation, tick and channel creation moves are *closed-world* moves: they correspond to the case where a group of players evolves on its own, in isolation; they are central to the notion of interactive equivalence.

Basic	Full	Closed-world
Left half-forking Right half-forking	Forking	Forking
Input Output	Input Output	Synchronisation
Channel creation	Channel creation	Channel creation
Tick	Tick	Tick

Table 2: Summary of classes of moves

- We need a third class of moves, called *full* and consisting of forking, input, output, tick and channel creation. They allow us to focus on a single player and all of its avatars. They appear, e.g., in the statement of Lemma 12, which is a partial correctness criterion for closed-world plays.

Formally, we define moves as cospans $X \hookrightarrow P \hookleftarrow Y$ in the category of diagrams (technically a presheaf category $\widehat{\mathbb{C}}$—see Section 3), where X is the initial position and Y the final one. Both legs of the cospan are actually monic arrows in $\widehat{\mathbb{C}}$, as will be the case for all cospans considered here.

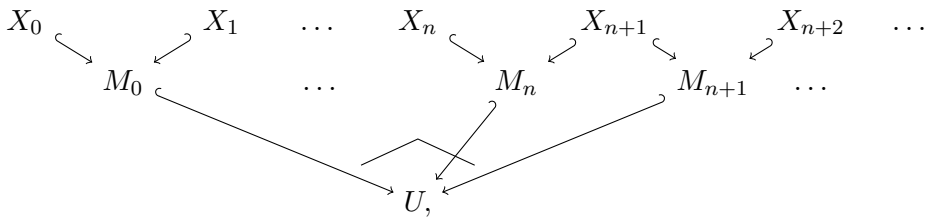
2.3 Plays

We now sketch how plays are defined as glueings of moves. We start with the following example, depicted in Figure 1. The initial position consists of two players p_1 and p_2 sharing knowledge of a channel a, each of them knowing another channel, resp. a_1 and a_2. The play consists of four moves: first p_1 forks into $p_{1,1}$ and $p_{1,2}$, then p_2 forks into $p_{2,1}$ and $p_{2,2}$, and then $p_{1,1}$ does a left half-fork into $p_{1,1,1}$; finally $p_{1,1,1}$ synchronises (as the sender) with $p_{2,1}$. Now, we reach the limits of the graphical representation, but the order in which p_1 and p_2 fork is irrelevant: if p_2 forks before p_1, we obtain the same play. This means that glueing the various parts of the picture in Figure 1 in different orders formally yields the same result (although there are subtle issues in representing this result graphically in a canonical way).

Let us now sketch a definition of plays. Recall that moves may be seen as cospans $X \hookrightarrow M \hookleftarrow Y$, and consider an *extended* notion of move, which may occur in a position not limited to players involved in the move. For

example, the moves in Figure 1 are extended moves in this sense. We may now state:

Definition 1. *A* play *is an embedding $X_0 \hookrightarrow U$ in the category $\widehat{\mathbb{C}}$ of diagrams, isomorphic to a possibly denumerable 'composition' of moves in the (bi)category $\mathsf{Cospan}(\widehat{\mathbb{C}})$ of cospans in $\widehat{\mathbb{C}}$, i.e., obtained as a colimit:*



where each $X_i \hookrightarrow M_i \hookleftarrow X_{i+1}$ is an extended move.

Notation: we often denote plays just by U, leaving the embedding $X \hookrightarrow U$ implicit.

Remark 1. *For finite plays, one might want to keep track not only of the initial position, but also of the final position. This indeed makes sense. Finite plays then compose 'vertically', and form a double category. But infinite plays do not really have any final position, which explains our definition.*

Let a morphism $(X \hookrightarrow U) \to (Y \hookrightarrow V)$ of plays be a pair (h, k) making the diagram on the right commute in $\widehat{\mathbb{C}}$. This permits both inclusion 'in width' and 'in height'. E.g., the play consisting of the left-hand basic move in (2) embeds in exactly two ways into the play of Figure 1. (Only two because the image of the base position must lie in the base position of the codomain.) We have:

$$\begin{array}{ccc} U & \xrightarrow{k} & V \\ \uparrow & & \uparrow \\ X & \xrightarrow{h} & Y. \end{array}$$

Proposition 1. *Plays and morphisms between them form a category $\mathbb{E}$.*

There is a projection functor $\mathbb{E} \to \mathbb{B}$ mapping each play $X \hookrightarrow U$ to its base position X. This functor has a section, which is an embedding $\mathbb{B} \hookrightarrow \mathbb{E}$, mapping each position X to the 'identity' play $X \hookrightarrow X$ on X.

Remark 2 (Size)**.** *The category $\mathbb{E}$ is only locally small. Since presheaves on a locally small category are less well-behaved than on a small category, we will actually consider a skeleton of $\mathbb{E}$. Because $\mathbb{E}$ consists only of denumerable presheaves, this skeleton is a small category. Thus, our presheaves in the next section may be understood as taken on a small category.*

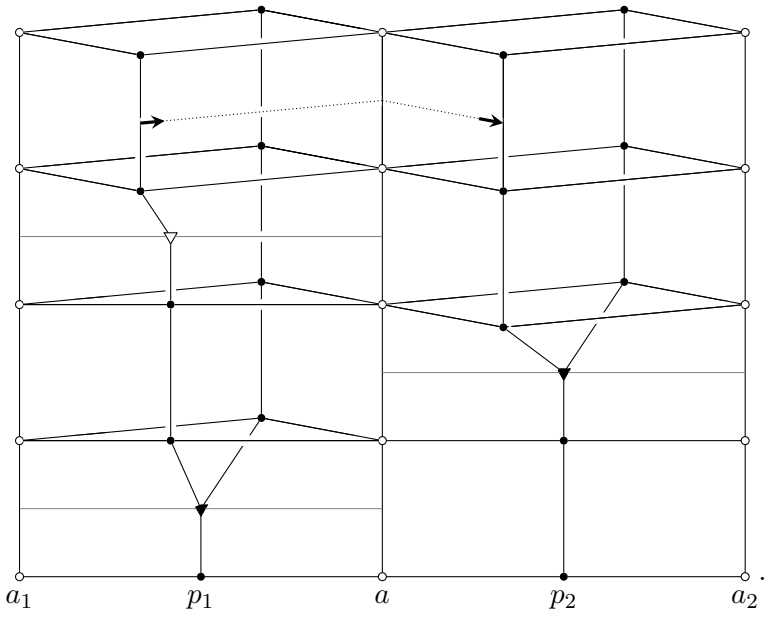


Figure 1: An example play

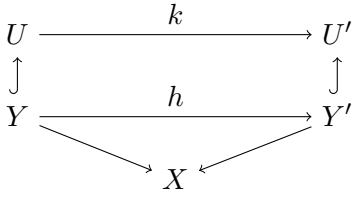
Remark 3. *Plays are not very far from being just (infinite) abstract syntax trees (or forests) 'glued together along channels'.*

2.4 Relativisation

If we now want to restrict to plays over a given base position X, we may consider

Definition 2. *Let the category $\mathbb{E}_X$ have*

- *as objects pairs of a play $Y \hookrightarrow U$ and a morphism $Y \to X$,*
- *as morphisms $(Y \hookrightarrow U) \to (Y' \hookrightarrow U')$ all pairs (h, k) making the diagram*



commute in $\widehat{\mathbb{C}}$.

We will usually abbreviate $U \hookleftarrow Y \to X$ as just U when no ambiguity arises. As for morphisms of positions, in full generality, h and k, as well as the morphisms $Y \to X$, do not have to be injective.

Example 1. *Let X be the position* *. The play in Figure 1, say $Y \hookrightarrow U$, equipped with the injection $Y \hookrightarrow X$ mapping the two players of Y to the two leftmost players of X, is an object of $\mathbb{E}_X$.*

One naively could imagine that the objects $\mathbb{E}_X$ could just consist of plays $X \hookrightarrow U$ on X. However, spatial decomposition, Theorem 1, relies on our slightly more complex definition. E.g., still in Figure 1, this allows us to distinguish between the identity view $[2] = [2] \overset{p_1}{\hookrightarrow} X$ on p_1 from the identity view $[2] = [2] \overset{p_2}{\hookrightarrow}$ on p_2, which would otherwise not be possible.

3 Diagrams

In this section, we define the category on which the string diagrams of the previous section are presheaves. The techniques used here date back at least to Carboni and Johnstone [5, 6].

3.1 First steps

Let us first consider two small examples. It is well-known that directed graphs form a presheaf category: consider the category $\mathbb{C}$ freely generated by the graph with two vertices, say $\star$ and $[1]$, and two edges $d, c \colon \star \to [1]$ between them. One way to visualise this is to compute the *category of elements* of a few presheaves on $\mathbb{C}$. Recall that the category of elements of a presheaf F on $\mathbb{C}$ is the comma category $y \downarrow_{\widehat{\mathbb{C}}} F$, where y is the Yoneda embedding. Via Yoneda, it has as elements pairs (C, x) with $C \in \mathrm{ob}(\mathbb{C})$ and $x \in F(C)$, and morphisms $(C, x) \to (D, y)$ morphisms $f \colon C \to D$ in $\mathbb{C}$ such that $F(f)(y) = x$ (which we abbreviate as $y \cdot f = x$ when the context is clear).

Example 2. *Consider the graph*

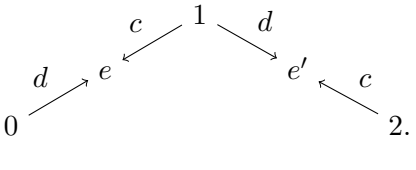
$$0 \xrightarrow{\ e\ } 1 \xrightarrow{\ e'\ } 2$$

with three vertices $0, 1$*, and* 2*, and two edges* e *and* e'*.*

This graph is represented by the presheaf F *defined by the following equations, whose category of elements is actually freely generated by the graph on the right:*

- $F(\star) = \{0, 1, 2\}$,
- $F([1]) = \{e, e'\}$,

- $e \cdot d = 0$,
- $e \cdot c = 1$,
- $e' \cdot d = 1$,
- $e' \cdot c = 2$,



This latter graph is not exactly the original one, but it does represent it. Indeed, for each vertex we know whether it is in $F(\star)$ *or* $F([1])$*, hence whether it represents a 'vertex' or an 'edge'. The arrows all go from a 'vertex'* v *to an 'edge'* e*. They lie above* d *when* v *is the domain of* e*, and above* c *when* v *is the codomain of* e*.*

Multigraphs, i.e., graphs whose edges have a list of sources instead of just one, may also be seen as a presheaves on the category freely generated by the graph with

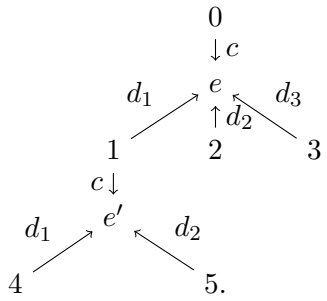
- as vertices: one special vertex $\star$, plus for each natural number n a vertex, say, $[n]$; and
- $n + 1$ edges $\star \to [n]$, say $d_1, \ldots, d_n$, and c.

It should be natural for presheaves on this category to look like multigraphs: the elements of a presheaf F above $\star$ are the vertices in the multigraph, the elements above $[n]$ are the n-ary multiedges, and the action of the d_i's give the ith source of a multiedge, while the action of c gives its target.

Example 3. *Similarly, computing a few categories of elements might help visualising. As above, consider* F *defined by*

- $F(\star) = \{0, 1, 2, 3, 4, 5\}$,
- $F([1]) = F([0]) = \emptyset$,
- $F([2]) = \{e'\}$,
- $F([3]) = \{e\}$,
- $F([n+4]) = \emptyset$,
- $e \cdot c = 0$,
- $e \cdot d_1 = 1$,
- $e \cdot d_2 = 2$,
- $e \cdot d_3 = 3$,
- $e' \cdot c = 1$,
- $e' \cdot d_1 = 4$,
- $e' \cdot d_2 = 5$,

whose category of elements is freely generated by the graph:



Now, this pattern may be extended to higher dimensions. Consider for example extending the previous base graph with a vertex $[m_1, \ldots, m_n; p]$ for all natural numbers $n, p, m_1, \ldots, m_n$, plus edges

$$
\begin{array}{l}
s_1 \colon [m_1] \to [m_1, \ldots, m_n; p], \\
\ldots, \\
s_n \colon [m_n] \to [m_1, \ldots, m_n; p], \mathit{and} \\
t \colon [p] \to [m_1, \ldots, m_n; p].
\end{array}
$$

Let now $\mathbb{C}$ be the free category on this extended graph. Presheaves on $\mathbb{C}$ are a kind of 2-multigraphs: they have vertices, multiedges, and multiedges between multiedges.

We could continue this in higher dimensions.

3.2 Constructing the base category

Our base category follows a very similar pattern. We start from a slightly different graph: let $\mathbb{G}_0$ have just one vertex $\star$; let $\mathbb{G}_1$, have one vertex $\star$, plus a vertex $[n]$ for each natural number n, plus n edges $d_1, \ldots, d_n \colon \star \to [n]$. Let $\mathbb{C}_0$ and $\mathbb{C}_1$ be the categories freely generated by $\mathbb{G}_0$ and $\mathbb{G}_1$, respectively.

So, presheaves on $\mathbb{C}_1$ are a kind of hypergraphs with arity (since vertices incident to a hyperedge are numbered). This is enough to model positions.

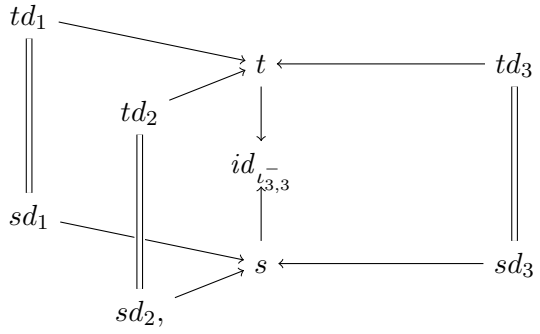
Now, consider the graph $\mathbb{G}_2$, which is $\mathbb{G}_1$ augmented with:

- for all n, vertices $\heartsuit_n$, π^l_n, π^r_n, ν_n,
- for all n and $1 \leq i \leq n$, vertices $\iota^+_{n,i}$ and $\iota^-_{n,i}$,
- for all n, edges $s, t \colon [n] \to \heartsuit_n$, $s, t \colon [n] \to \pi^l_n$, $s, t \colon [n] \to \pi^r_n$, $s \colon [n] \to \nu_n$, $t \colon [n+1] \to \nu_n$,
- for all n and $1 \leq i \leq n$, edges $s, t \colon [n] \to \iota^+_{n,i}$, $s, t \colon [n] \to \iota^-_{n,i}$.

We slightly abuse language here by calling all these t's and s's the same. We could label them with their codomain, but we refrain from doing so for the sake of readability.

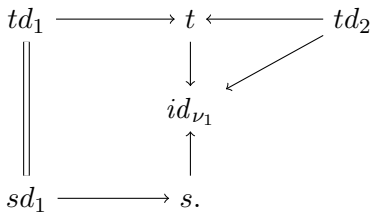
Now, let $\mathbb{C}_2$ be the category generated by $\mathbb{G}_2$ and the relations $s \circ d_i = t \circ d_i$ for all n and $1 \leq i \leq n$ (for all sensible—common—codomains).

Example 4. *Again, computing a few categories of elements is in order. For example, the category of elements of (the representable presheaf on) $\iota^-_{3,3}$ is the poset freely generated by the graph*



to be compared with the corresponding pictures (3).

Example 5. *Similarly, the category of elements of ν_1 is the poset freely generated by the graph*



Note that only channel creation changes the number of channels known to the player, and accordingly the corresponding morphism t has domain $[n+1]$.

Presheaves on $\mathbb{C}_2$ are enough to model views, but since we want more, we continue, as follows.

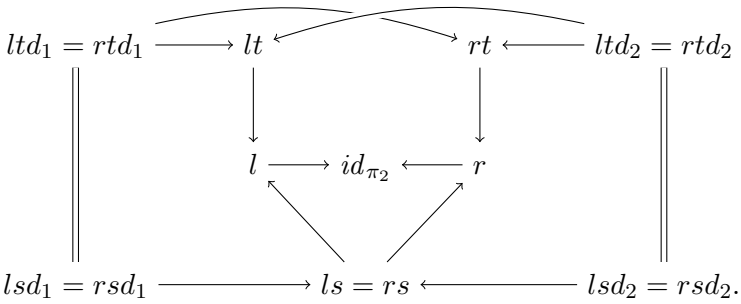
Let $\mathbb{G}_3$ be $\mathbb{G}_2$, augmented with:

- for all n, a vertex π_n, and
- edges $l\colon \pi_n^l \to \pi_n$ and $r\colon \pi_n^r \to \pi_n$.

Definition 3. *Let $\mathbb{C}_3$ be the category generated by $\mathbb{G}_3$, the previous relations, plus the relations $l \circ s = r \circ s$.*

The equation models the fact that a forking move should be played by just one player. We also call $s = l \circ s = r \circ s$ the common composite, which gives a uniform notation for the initial player of full moves.

Example 6. *The category of elements of π_2 is the poset freely generated by the graph*



The two views corresponding to left and right half-forking are subcategories, and the object id_{π_2} 'ties them together'.

Presheaves on $\mathbb{C}_3$ are enough to model full moves; to model closed-world moves, and in particular synchronisation, we continue as follows.

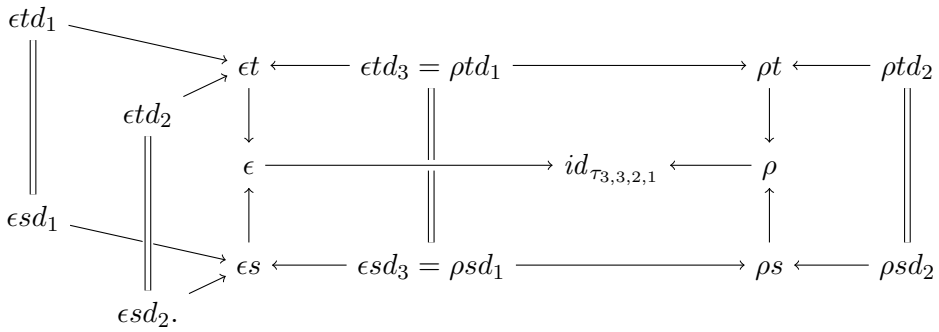
Let $\mathbb{G}_4$ be $\mathbb{G}_3$, augmented with, for all n, m, $1 \leq i \leq n$, and $1 \leq j \leq m$,

- a vertex $\tau_{n,i,m,j}$, and
- edges $\epsilon \colon \iota^+_{n,i} \to \tau_{n,i,m,j}$ and $\rho \colon \iota^-_{m,j} \to \tau_{n,i,m,j}$ (ϵ and ρ respectively stand for 'emission' and 'reception').

Definition 4. *Let $\mathbb{C}_4$ be the category generated by $\mathbb{G}_4$, the previous relations, plus, for each $\iota^+_{n,i} \xrightarrow{\epsilon} \tau_{n,i,m,j} \xleftarrow{\rho} \iota^-_{m,j}$, the relation $\epsilon \circ s \circ d_i = \rho \circ s \circ d_j$.*

This equation is the exact point where we enforce that a synchronisation involves an input and an output on the same channel, as announced in Example 4.

Example 7. *The category of elements of $\tau_{3,3,1,1}$ is the preorder freely generated by the graph*



Again, the two views corresponding to $\iota^+_{3,3}$ and $\iota^-_{2,1}$ are subcategories, and the new object $\tau_{3,3,2,1}$ ties them together.

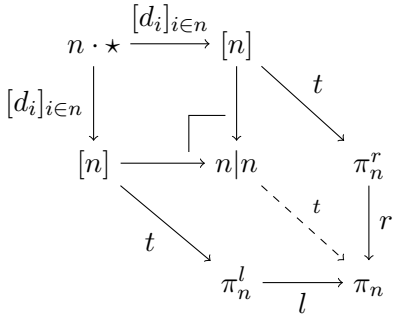
3.3 Positions and moves

We have now defined the base category $\mathbb{C} = \mathbb{C}_4$ on which the string diagrams of Section 2 are presheaves. More accurately we have defined a sequence $\mathbb{C}_0 \hookrightarrow \ldots \hookrightarrow \mathbb{C}_4$ of subcategories.

Positions Positions are finite presheaves on $\mathbb{C}_1$, or equivalently, finite presheaves on $\mathbb{C}_4$ empty except above $\mathbb{C}_1$.

Moves Basic moves should be essentially representable presheaves on objects in $\mathrm{ob}(\mathbb{C}_2) \setminus \mathrm{ob}(\mathbb{C}_1)$. Recall however that basic moves are defined as particular cospans in $\widehat{\mathbb{C}}$. This is also easy: in the generating graph $\mathbb{G}_2$, each such object c has exactly two morphisms s and t into it, from objects, say, $[n_s]$ and $[n_t]$, respectively. By Yoneda, these induce a cospan $[n_s] \xrightarrow{s} c \xleftarrow{t} [n_t]$ in $\widehat{\mathbb{C}}$, which is the desired cospan. (Observe, again, that only ν_n has $n_s \neq n_t$.)

Similarly, full moves either are basic moves, or are essentially representable presheaves on objects in $\mathrm{ob}(\mathbb{C}_3) \setminus \mathrm{ob}(\mathbb{C}_1)$, i.e., representables on some π_n. To define the expected cospan, first observe that by the equation $ls = rs$, we obtain an arrow $[n] \xrightarrow{s} \pi_n^l \xrightarrow{l} \pi_n$, equal to rs, in $\widehat{\mathbb{C}}$. This will form the first leg of the cospan. For the other, observe that for each n and $i \in n$, we obtain, by the equations $ltd_i = lsd_i = rsd_i = rtd_i$ and by Yoneda, that the outermost part of



(4)

commutes in $\widehat{\mathbb{C}}$, where $n \cdot \star$ denotes an n-fold coproduct of $\star$. Letting $n|n$ be the induced pushout, and the dashed arrow t be obtained by its universal property, we obtain the desired cospan $[n] \xrightarrow{ls} \pi_n \xleftarrow{t} n|n$.

Finally, closed-world moves either are full moves, or are essentially representable presheaves on some $\tau_{n,i,m,j}$. To define the expected cospan, we proceed as in Figure 2: compute the pushout $n\,{}_i|_j\,m$, and infer the dashed arrows s' and t' to obtain the desired cospan $n\,{}_i|_j\,m \xrightarrow{s'} \tau_{n,i,m,j} \xleftarrow{t'} n\,{}_i|_j\,m$.

Remark 4 (Isomorphisms)**.** *Moves are particular cospans in $\widehat{\mathbb{C}}$. For certain moves, the involved objects are representable, but not for others, like forking or synchronisation, whose final position is not representable. In the latter cases, our definition thus relies on a choice, e.g., of pushout in* (4)*. Thus, let us be completely accurate: a move is a cospan which is isomorphic to one of the cospans chosen above, in $\widehat{\mathbb{C}}^{\cdot \leftarrow \cdot \rightarrow \cdot}$, i.e., the category of functors*

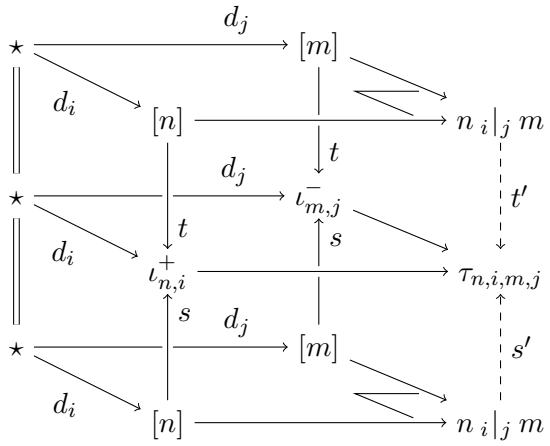


Figure 2: Construction of the synchronisation move

from the category $\cdot \leftarrow \cdot \rightarrow \cdot$ *(generated by the graph with three objects and an arrow from one to each of the other two) to* $\widehat{\mathbb{C}}$.

3.4 Extended moves, plays, and relativisation

The most delicate part of our formalisation of Section 3 is perhaps the passage from moves to extended moves. Recall from the paragraph above Definition 1 that an extended move should be like a move occurring in a larger position.

Moves with interfaces To formalise this idea, we first equip moves with interfaces, as standard in graph rewriting [23]. Since moves are cospans, one might expect that interfaces be cospans too. This may be done, but there is a simpler, equivalent presentation. The route we follow here might have to be generalised in order to handle more complex calculi than CCS, but let us save the complications for later work.

Here, we define an *interface* for a cospan $X \to M \leftarrow Y$ to consist of a

presheaf I and morphisms $X \leftarrow I \rightarrow Y$ such that

$$\begin{array}{ccc} I & \longrightarrow & Y \\ \downarrow & & \downarrow \\ X & \longrightarrow & M \end{array} \tag{5}$$

commutes, and I has dimension 0, i.e., is empty except above $\mathbb{C}_0$, i.e., consists only of channels.

Definition 5. *A cospan equipped with an interface is called a* cospan with interface*.*

Moves are particular cospans, and we now equip them with canonical interfaces: all moves except channel creation preserve the set of channels, the interface is then $n \cdot \star$, with the obvious inclusion. For example, the less obvious case is π_n: we choose

$$\begin{array}{ccc} n \cdot \star & \longrightarrow & n|n \\ \downarrow & & \downarrow \\ [n] & \longrightarrow & \pi_n, \end{array}$$

where the upper map is as in (4). For channel creation, we naturally choose

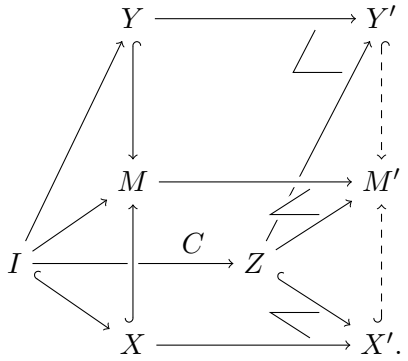
$$\begin{array}{ccc} n \cdot \star & \xrightarrow{[d_i]_{i \in n}} & [n+1] \\ \downarrow & & \downarrow \\ [n] & \longrightarrow & \nu_n. \end{array}$$

Definition 6. *A* move with interface *is one of these cospans with interface. The basic, full, or closed-world character is retained from the underlying move.*

Extended moves We now plug moves with interfaces into contexts, in the following sense.

Definition 7. *A* context *for a cospan with interface* (5) *is a position Z, equipped with a morphism $I \rightarrow Z$.*

From any cospan with interface μ as in (5) and context $C \colon I \rightarrow Z$, we construct the cospan $C[\mu]$ as in:



I.e., we push the available morphisms out of I along C, and infer the dashed arrows, which form the desired cospan.

Definition 8. *An* extended move *is a cospan of the shape $C[\mu]$, for any move with interface μ and context C as above.*

Example 8. *Recall that* $[2]$ *is a position with one player knowing two channels. Recall from Figure 2 the pushout*

$$\begin{array}{ccc} \star & \xrightarrow{d_1} & [2] \\ {\scriptstyle d_2}\downarrow & & \downarrow{\scriptstyle p_2} \\ [2] & \xrightarrow[p_1]{} & 2\ {}_2|_1\ 2, \end{array}$$

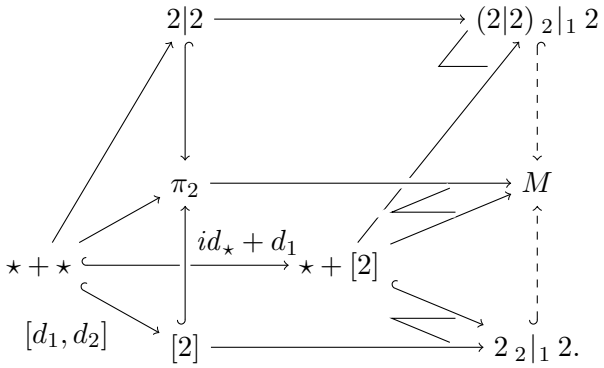
equivalently obtained as the pushout

$$\begin{array}{ccc} \star+\star & \xrightarrow{id_\star + d_1} & \star + [2] \\ {\scriptstyle [d_1,d_2]}\downarrow & & \downarrow{\scriptstyle [a_1,p_2]} \\ [2] & \xrightarrow[p_1]{} & 2\ {}_2|_1\ 2. \end{array}$$

The base position of Figure 1 is thus $2\ {}_2|_1\ 2$*. Recall also from* (4) *that* $2|2$ *denotes the position with two players both knowing two channels. Now, we have the forking move* $[2] \hookrightarrow \pi_2 \hookleftarrow 2|2$*. Equipping it with the interface*

$$[d_1, d_2] \colon \star + \star \to [2],$$

and putting it in the context $id_\star + d_1 \colon \star + \star \to \star + [2]$*, (which happens to be the same as the interface), we obtain*



This formally constructs the first layer of Figure 1. Constructing the whole play would be a little too verbose to be included here, but essentially straightforward.

Plays and relativisation We may now read Definition 1 again, this time in the formal setting, to define plays. Similarly, the definition of morphisms now makes rigorous sense, as well as Proposition 1.

Proof of Proposition 1: $\mathbb{E}$ is the full subcategory of the arrow category of $\widehat{\mathbb{C}}$ whose objects are plays. □

Similarly, Section 2.4 now makes rigorous sense.

4 Innocent strategies as sheaves

Now that the category of plays is defined, we move on to defining innocent strategies. There is a notion of a Grothendieck *site* [31], which consists of a category equipped with a (generalised) topology. On such sites, one may define a category of sheaves, which are very roughly the presheaves that are determined locally w.r.t. the generalised topology. We claim that there is a topology on each $\mathbb{E}_X$, for which sheaves adequately model innocent strategies. Fortunately, in our setting, sheaves admit a simple description, so that we can avoid the whole machinery. But sheaves were the way we arrived at the main ideas presented here, because they convey the right intuition: plays form a Grothendieck site, and the states of innocent strategies should be determined locally.

In this section, we first define innocent strategies, and state the spatial and temporal decomposition theorems. We then present our coalgebraic interpretation of innocent strategies, i.e., we define a polynomial endofunctor

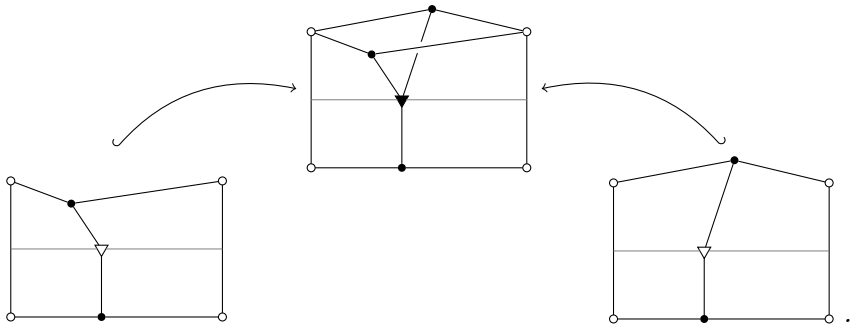
F, and show that presheaves of finite ordinals on views form a final F-coalgebra. We then derive from this a formal language and its interpretation in terms of innocent strategies. We finally use this language to translate CCS with recursive equations into innocent strategies.

4.1 Innocent strategies

Definition 9. *A* view *is a finite, possibly empty 'composition'* $[n] \hookrightarrow V$ *of (extended) basic moves in* $\mathsf{Cospan}(\widehat{\mathbb{C}})$*, i.e., a play in which all the cospans are basic moves.*

The empty case yields the view $[n] \hookrightarrow [n]$; but note that empty presheaves (with not even an initial position) are not views.

Example 9. *Forking* (1) *has two non-trivial views, namely the (left legs of) basic moves* (2)*. Each of them embeds into forking:*



Example 10. *In Figure 1, the leftmost branch contains a view consisting of three basic moves: two* π_2^l *and an output.*

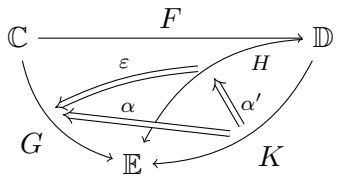
Definition 10. *For any position* X*, let* $\mathbb{V}_X$ *be the full subcategory of* $\mathbb{E}_X$ *consisting of views.*

More precisely, $\mathbb{V}_X$ consists of spans $U \hookleftarrow Y \to X$ where $Y \hookrightarrow U$ is a view.

Definition 11. *Let the category* S_X *of* innocent strategies *on* X *be the category* $\widehat{\mathbb{V}_X}$ *of presheaves on* $\mathbb{V}_X$*.*

A possible interpretation is that for a presheaf $F \in \widehat{\mathbb{V}_X}$ and view $V \in \mathbb{V}_X$, $F(V)$ is the set of possible *states* of the strategy F after playing V.

It might thus seem that we could content ourselves with defining only views, as opposed to plays. However, in order to define interactive equivalences in Section 5, we need to view innocent strategies as (particular) presheaves on the whole of $\mathbb{E}_X$.



The connection is as follows. Recall from MacLane [30] the notion of *right Kan extension*. Given functors F and G as on the right, a right Kan extension $\mathrm{Ran}_F(G)$ of G along F is a functor $H\colon \mathbb{D} \to \mathbb{E}$, equipped with a natural transformation $\epsilon\colon HF \to G$, such that for all functors $K\colon \mathbb{D} \to \mathbb{E}$ and transformations $\alpha\colon KF \to G$, there is a unique $\alpha'\colon K \to H$ such that $\alpha = \epsilon \bullet (\alpha' \circ id_F)$, where $\bullet$ is vertical composition of natural transformations. Now, precomposition with F induces a functor $\mathsf{Cat}(F, \mathbb{E})\colon \mathsf{Cat}(\mathbb{D}, \mathbb{E}) \to \mathsf{Cat}(\mathbb{C}, \mathbb{E})$, where $\mathsf{Cat}(\mathbb{D}, \mathbb{E})$ is the category of functors $\mathbb{D} \to \mathbb{E}$ and natural transformations between them. When $\mathbb{E}$ is complete, right Kan extensions always exist (and an explicit formula for our setting is given below), and choosing one of them for each functor $\mathbb{C} \to \mathbb{E}$ induces a right adjoint to $\mathsf{Cat}(F, \mathbb{E})$. Furthermore, it is known that when F is full and faithful, then ϵ is a natural isomorphism, i.e., $HF \cong G$.

Proposition 2. *If F is full and faithful, then Ran_F is a full essential embedding.*

Proof: For essential injectivity on objects, assume $H = \mathrm{Ran}_F(G)$, $\mathrm{Ran}_F(G') = H'$, and $i\colon H \to H'$ is an isomorphism with inverse k. We must construct an isomorphism $G \cong G'$. Let $j\colon G \to G'$ be $\epsilon_{G'} \bullet (iF) \bullet \epsilon_G^{-1}$. Similarly, let $l\colon G' \to G$ be $\epsilon_G \bullet (kF) \bullet \epsilon_{G'}^{-1}$. We have

$$\begin{array}{rcl} l \bullet j & = & \epsilon_G \bullet (kF) \bullet \epsilon_{G'}^{-1} \bullet \epsilon_{G'} \bullet (iF) \bullet \epsilon_G^{-1} \\ & = & \epsilon_G \bullet (kF) \bullet (iF) \bullet \epsilon_G^{-1} \\ & = & \epsilon_G \bullet ((k \bullet i) \circ F) \bullet \epsilon_G^{-1} \\ & = & \epsilon_G \bullet \epsilon_G^{-1} \\ & = & id_G. \end{array}$$

Similarly, $j \bullet l = id_{G'}$ and we have $G \cong G'$.

To see that Ran_F is full, observe that for any $i\colon H \to H'$, with $H = \mathrm{Ran}_F(G)$ and $H' = \mathrm{Ran}_F(G')$, $j = \epsilon_{G'} \bullet (iF) \bullet \epsilon_G^{-1}$ is an antecedent of i by Ran_F. Indeed, by definition, $\mathrm{Ran}_F(j)$ is the unique $i'\colon H \to H'$ such that $\epsilon_{G'} \bullet (i'F) = j \bullet \epsilon_G$. But the latter is equal to $\epsilon_{G'} \bullet (iF)$, so $i' = i$.

Finally, to show that Ran_F is faithful, consider $G, G' \colon \mathbb{C} \to \mathbb{E}$ and two natural transformations $i, j \colon G \to G'$ such that $\mathrm{Ran}_F(i) = \mathrm{Ran}_F(j) = k$. Then, by construction of k, we have

$$i \bullet \epsilon_G = \epsilon_{G'} \bullet (kF) = j \bullet \epsilon_G.$$

But, ϵ_G being an isomorphism, this implies $i = j$ as desired. □

Returning to views and plays, the embedding $i_X \colon \mathbb{V}_X \hookrightarrow \mathbb{E}_X$ is full, so right Kan extension along $i_X^{op} \colon \mathbb{V}_X^{op} \to \mathbb{E}_X^{op}$ induces a full essential embedding $\mathrm{Ran}_{i_X^{op}} \colon \widehat{\mathbb{V}_X} \to \widehat{\mathbb{E}_X}$. The (co)restriction of this essential embedding to its essential image thus yields an essentially surjective, fully faithful functor, i.e., an equivalence of categories:

Proposition 3. *The category S_X is equivalent to the essential image of $\mathrm{Ran}_{i_X^{op}}$.*

The standard characterisation of right Kan extensions as ends [30] yields, for any $F \in \widehat{\mathbb{V}_X}$ and $U \in \mathbb{E}_X$:

$$\mathrm{Ran}_{i_X^{op}}(F)(U) = \int_{V \in \mathbb{V}_X} F(V)^{\mathbb{E}_X(V,U)},$$

i.e., giving an element of $\mathrm{Ran}_{i_X^{op}}(F)$ on a play U amounts to giving, for each view V and morphism $V \to U$, an element of $F(V)$, satisfying some compatibility conditions. In Example 11 below, we compute an example right Kan extension.

The interpretation of strategies in terms of states extends: for any presheaf $F \in \widehat{\mathbb{E}_X}$ and play $U \in \mathbb{E}_X$, $F(U)$ is the set of possible *states* of the strategy F after playing U. That F is in the image of $\mathrm{Ran}_{i_X^{op}}$ amounts to $F(U)$ being a compatible tuple of states of F after playing each view of U.

Example 11. *Here is an example of a presheaf $F \in \widehat{\mathbb{E}_X}$ which is not innocent, i.e., not in the image of $\mathrm{Ran}_{i_X^{op}}$. Consider the position X consisting of three players, say x, y, z, sharing a channel, say a. Let X_x be the subposition with only x and a, and similarly for X_y, X_z, $X_{x,y}$, and $X_{x,z}$. Let $I_x = (\iota_{1,1}^- \hookleftarrow X_x \hookrightarrow X)$ be the play where x inputs on a, and similarly let O_y and O_z be the plays where y and z output on a, respectively. Let now $S_{x,y} = (\tau_{1,1,1,1} \hookleftarrow X_{x,y} \hookrightarrow X)$ be the play where x and y synchronise on a (x inputs and y outputs), and similarly let $S_{x,z}$ be the play where x and z synchronise on a.*

Finally, let $F(S_{x,y}) = 2$ be a two-element set, and $F(S_{x,z}) = \emptyset$. To define F on other plays, the idea is to map any subplay of $S_{x,y}$ and $S_{x,z}$ to a one-element set 1, and other plays to $\emptyset$. But if U is a subplay of, say, $S_{x,y}$, then, for any epic $e\colon U \twoheadrightarrow U'$, U' has the same views as U, so we choose to also map U' to 1. Formally, beyond $F(S_{x,y}) = 2$ and $F(S_{x,z}) = \emptyset$, define F for any play U' by:

- *if there exists a player $t \in \{y, z\}$, a play U, and arrows $U' \overset{e}{\twoheadleftarrow} U \overset{i}{\hookrightarrow} S_{x,t}$, with e epic and i monic but not epic, then let $F(U') = 1$;*
- *otherwise let $F(U') = \emptyset$.*

In particular, for any strict superplay U of $S_{x,y}$ or $S_{x,z}$, $F(U) = \emptyset$, and we have $F(I_x) = F(O_y) = F(O_z) = F(id_x) = F(id_y) = F(id_z) = 1$.

This F fails to be innocent on two counts. First, since x and y accept to input and output in only one way, it is non-innocent to accept that they synchronise in more than one way. Formally, $S_{x,y}$ has two non-trivial views, I_x and O_y, so since F maps identity views to a singleton, $F(S_{x,y})$ should be isomorphic to $F(I_x) \times F(O_y) = 1 \times 1 = 1$. Second, since x and z accept to input and output, it is non-innocent to not accept that they synchronise. Formally, $F(S_{x,z})$ should also be a singleton. This altogether models the fact that in CCS, processes do not get to know with which other processes they synchronise.

The restriction of F to $\mathbb{V}_X$, i.e., $F' = F \circ i_X^{op}$, in turn has a right Kan extension F'', which is innocent. (In passing, the unit of the adjunction $\mathsf{Cat}(i_X^{op}, \mathsf{Set}) \dashv \mathrm{Ran}_{i_X^{op}}$ is a natural transformation $F \to F''$.) To conclude this example, let us compute F''. First, F' only retains from F its values on views. So, if X_x denotes the empty view on X_x, $F'(X_x) = 1$, and similarly $F'(X_y) = F'(X_z) = 1$. Furthermore, $F'(I_x) = F'(O_y) = F'(O_z) = 1$. Finally, for any view V not isomorphic to any of the previous ones, $F'(V) = \emptyset$. So, recall that F'' maps any play $U \hookleftarrow Y \hookrightarrow X$ to $\int_{V \in \mathbb{V}_X} F'(V)^{\mathbb{E}_X(V,U)}$. So, e.g., since the views of $S_{x,y}$ are subviews of I_x and O_y, we have $F''(S_{x,y}) = F'(I_x) \times F'(O_y) = 1$. Similarly, $F''(S_{x,z}) = 1$. But also, for any play U such that all views $V \to U$ are subviews of either of I_x, O_y, or O_z, we have $F''(U) = 1$. Finally, for any play U such that there exists a view $V \to U$ which is not a subview of any of I_x, O_y, or O_z, we have $F''(U) = \emptyset$.

One way to understand Proposition 3 is to view $\widehat{\mathbb{V}_X}$ as the syntax for innocent strategies: presheaves on views are (almost) infinite terms in a certain syntax (see Section 4.4 below). On the other hand, seeing them as

presheaves on plays will allow us to consider their global behaviour: see Section 5 when we restrict to the closed-world game. Thus, right Kan extension followed by restriction to closed-world will associate a semantics to innocent strategies.

Remark 5. *The relevant Grothendieck topology on $\mathbb{E}_X$ says, roughly, that a play is covered by its views. Any sheaf for this topology is determined by its restriction to $\mathbb{V}_X$, for its elements on any non-view play U are precisely amalgamations of its elements on views of U. Right Kan extension just computes these amalgamations in the particular case of a topology derived from a full subcategory, here views.*

So, we have defined for each X the category S_X of innocent strategies on X. This assignment is actually functorial $\mathbb{B}^{op} \to \mathsf{CAT}$, as follows (where CAT is the large category of locally small categories). Any morphism $f : Y \to X$ induces a functor $f_! : \mathbb{V}_Y \to \mathbb{V}_X$ mapping $(V \hookleftarrow Z \to Y)$ to $(V \hookleftarrow Z \to Y \to X)$. Precomposition with $(f_!)^{op}$ thus induces a functor $\mathsf{S}_f : \widehat{\mathbb{V}_X} \to \widehat{\mathbb{V}_Y}$.

Proposition 4. *This defines a functor $\mathsf{S} : \mathbb{B}^{op} \to \mathsf{CAT}$.*

Proof: A straightforward verification. □

But there is more: for any position, giving a strategy for each player in it easily yields a strategy on the whole position. We call this *amalgamation* of innocent strategies (because the functor S is indeed a *stack* [42], and this is a particular case of amalgamation in that stack). Formally, consider any subpositions X_1 and X_2 of a given position X, inducing a partition of the players of X, i.e., such that $X_1 \cup X_2$ contains all players of X, and $X_1 \cap X_2$ contains none. Then $\mathbb{V}_X$ is isomorphic to the coproduct $\mathbb{V}_{X_1} + \mathbb{V}_{X_2}$. (Indeed, a view contains in particular an initial player in X, which forces it to belong either in $\mathbb{V}_{X_1}$ or in $\mathbb{V}_{X_2}$.)

Definition 12. *Given innocent strategies F_1 on X_1 and F_2 on X_2, let their* amalgamation *be their copairing*

$$[F_1, F_2] : \mathbb{V}_X^{op} \cong (\mathbb{V}_{X_1} + \mathbb{V}_{X_2})^{op} \cong \mathbb{V}_{X_1}^{op} + \mathbb{V}_{X_2}^{op} \to \mathsf{Set}.$$

By universal property of coproduct:

Proposition 5. *Amalgamation yields an isomorphism of categories $\widehat{\mathbb{V}_X} \cong \widehat{\mathbb{V}_{X_1}} \times \widehat{\mathbb{V}_{X_2}}$.*

Example 12. *Consider again the position X from Example 11, and let $X_{y,z}$ be the subposition with only y and z. We have $\mathbb{V}_X \simeq (\mathbb{V}_{X_x} + \mathbb{V}_{X_{y,z}})$, which we may explain by hand as follows. A view on X has a base player, x, y, or z, and so belongs either in $\mathbb{V}_{X_x}$ or in $\mathbb{V}_{X_{y,z}}$. Furthermore, if V is a view on x and W is a view on y, then $\mathbb{V}_X(V, W) = \emptyset$ (and similarly for any pair of distinct players in X).*

Now, recall F', the restriction of F to $\mathbb{V}_X$. We may define $F_x \colon \mathbb{V}_{X_x}^{op} \to \mathsf{Set}$ to be the restriction of F' along the (opposite of the) embedding $\mathbb{V}_{X_x} \hookrightarrow \mathbb{V}_X$, and similarly $F_{y,z}$ to be the restriction of F' along $\mathbb{V}_{X_{y,z}} \hookrightarrow \mathbb{V}_X$. We have obviously $F' = [F_x, F_{y,z}]$.

Analogous reasoning leads to what we call spatial decomposition. For any X, let $\mathrm{Pl}(X) = \sum_n X([n])$, i.e., the set of pairs (n, x), where x is a player in X, knowing n channels.

Theorem 1. *We have $\widehat{\mathbb{V}_X} \cong \prod_{(n,x)\in \mathrm{Pl}(X)} \widehat{\mathbb{V}_{[n]}}$.*

Again, this is a particular case of amalgamation in the stack S, but we do not need to spell out the definition here.

4.2 Temporal decomposition

Let us now describe *temporal* decomposition. Recall that basic moves are left and right half-forking (2), input, output, tick, and channel creation.

Definition 13. *Let $\mathcal{M}$ be the graph with vertices all natural numbers n, and with edges $n \to n'$ all (isomorphism classes of) basic moves $M \colon [n] \to [n']$.*

Recall from Remark 4 that the notion of isomorphism considered here is that of an isomorphism of cospans in $\widehat{\mathbb{C}}$.

Definition 14. *Let $\mathcal{M}_n$ be the set of edges from n in $\mathcal{M}$.*

For stating the temporal decomposition theorem, we need a standard [21] categorical construction, the category of families on a given category $\mathbb{C}$. First, given a set X, consider the category $\mathrm{Fam}(X)$ with as objects X-indexed families of sets $Y = (Y_x)_{x\in X}$, and as morphisms $Y \to Z$ families $(f_x \colon Y_x \to Z_x)_{x\in X}$ of maps. This category is equivalently described as the slice category Set/X. To see the correspondence, consider any family $(Y_x)_{x\in X}$, and map it to the projection function $\sum_{x\in X} Y_x \to X$ sending (x, y) to x. Conversely, given $f \colon Y \to X$, let, for any $x \in X$, Y_x be the fibre of f above x, i.e., $f^{-1}(x)$.

Generalising from sets X to small categories $\mathbb{C}$, $\mathrm{Fam}(\mathbb{C})$ has as objects families $p\colon Y \to \mathrm{ob}(\mathbb{C})$ indexed by the objects of $\mathbb{C}$. Morphisms $(Y,p) \to (Z,q)$ are pairs of $u\colon Y \to Z$ and $v\colon Y \to \mathrm{mor}(\mathbb{C})$, where $\mathrm{mor}(\mathbb{C})$ is the set of morphisms of $\mathbb{C}$, such that $\mathrm{dom} \circ v = p$, and $\mathrm{cod} \circ v = q \circ u$. Thus, any element $y \in Y$ above $C \in \mathbb{C}$ is mapped to some $u(y) \in Z$ above $C' \in \mathbb{C}$, and this mapping is labelled by a morphism $v(y)\colon C \to C'$ in $\mathbb{C}$. The obtained category is locally small.

Further generalising, for $\mathbb{C}$ a locally small category, we may define $\mathrm{Fam}(\mathbb{C})$ in exactly the same way (with Y still a set), and the obtained category remains locally small.

The temporal decomposition theorem is:

Theorem 2. *There is an equivalence of categories*

$$\mathsf{S}_n \simeq \mathrm{Fam}\left(\prod_{M \in \mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)}\right).$$

The main intuition is that an innocent strategy is determined up to isomorphism by (i) its initial states, and (ii) what remains of them after each possible basic move. The family construction is what permits innocent strategies with several possible states over the identity play.

Proof sketch: For general reasons, we have:

$$\begin{array}{rcl} \mathrm{Fam}\left(\prod_{M \in \mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)}\right) & = & \mathrm{Fam}\left(\prod_{M \in \mathcal{M}_n} [\mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}]\right) \\ & \cong & \mathrm{Fam}\left(\left[\sum_{M \in \mathcal{M}_n} \mathbb{V}_{\mathrm{cod}(M)}{}^{op}, \mathsf{Set}\right]\right) \\ & \simeq & \left[\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}\right] \downarrow \Delta, \end{array}$$

where $\Delta\colon \mathsf{Set} \to [\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}]$ maps any set X to the constant presheaf mapping any object to X and any arrow to the identity.

By definition, the last category is a lax pullback

$$\begin{array}{ccc} \left[\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}\right] & = & \left[\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}\right] \\ \Delta \uparrow & & \uparrow \\ \mathsf{Set} & \longleftarrow & \left[\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}\right] \downarrow \Delta \end{array}$$

in CAT.

Now, any basic move $M\colon n \to n'$ induces a functor $(-\circ M)\colon \mathbb{V}_{[n']} \to \mathbb{V}_{[n]}$, mapping any view $V \in \mathbb{V}_{[n']}$ to $V \circ M$ (with composition in $\mathsf{Cospan}(\widehat{\mathbb{C}})$). We show that the square

$$\begin{array}{ccc} \sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)} & = & \sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)} \\ {\scriptstyle !}\downarrow & \overset{\lambda}{\swarrow} & \downarrow{\scriptstyle [-\circ M]_{M\in\mathcal{M}_n}} \\ 1 & \xrightarrow[\ulcorner id_{[n]} \urcorner]{} & \mathbb{V}^{op}_{[n]} \end{array} \tag{6}$$

is a lax pushout in Cat, where $\lambda_{M,V}\colon id_{[n]} \to M \circ V$, seen in $\mathbb{V}_{[n]}$, is the obvious inclusion, which for general reasons is mapped by the hom-2-functor $\mathsf{CAT}(-, \mathsf{Set})$ to a lax pullback. But $\mathsf{CAT}(!, \mathsf{Set}) = \Delta$ and $\mathsf{CAT}(id, \mathsf{Set}) = id$, so we obtain a canonical isomorphism of lax pullbacks

$$\mathsf{S}_{[n]} = [\mathbb{V}^{op}_{[n]}, \mathsf{Set}] \cong \left[\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set} \right] \downarrow \Delta.$$

More detail is in Appendix A. □

Remark 6. *The theorem almost makes innocent strategies into a* sketch *(on the category with positions as objects, finite compositions of extended moves as morphisms, and the $\mathcal{M}_X$'s as distinguished cones). Briefly, being a sketch would require a bijection of sets $\mathsf{S}_n \cong \prod_{M\in\mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)}$. Here, the bijection becomes an equivalence of categories, and the family construction sneaks in.*

4.3 Innocent strategies as a terminal coalgebra

Temporal decomposition gives

$$\mathsf{S}_n \simeq \mathrm{Fam}\left(\prod_{M\in\mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)} \right),$$

for all n. Considering a variant of this formula as a system of equations will lead to our interpretation of CCS. The first step is to replace Set with FinOrd, the category of finite ordinals and monotone functions. The proof applies *mutatis mutandis* and we obtain an equivalence, which, because both categories are skeletal, is an isomorphism:

$$\widehat{\mathbb{V}_{[n]}} \cong \mathrm{Fam}_f\left(\prod_{M\in\mathcal{M}_n} \widehat{\mathbb{V}_{\mathrm{cod}(M)}} \right), \tag{7}$$

where

- Fam_f is the same as Fam but with finite families, i.e., for any category $\mathbb{C}$, $\mathrm{ob}(\mathrm{Fam}_f(\mathbb{C})) = \sum_{I \in \mathsf{FinOrd}} (\mathrm{ob}(\mathbb{C}))^I = (\mathrm{ob}(\mathbb{C}))^*$ is the set of finite words over objects of $\mathbb{C}$, also known as the free monoid on $\mathrm{ob}(\mathbb{C})$;

- and for any category $\mathbb{C}$, $\widehat{\mathbb{C}}$ denotes the functor category $[\mathbb{C}^{op}, \mathsf{FinOrd}]$.

Remark 7. *Recall that in the proof of Theorem 2,* Fam *arises from the 'constant presheaf' functor* $\Delta \colon \mathsf{Set} \to \widehat{-}$*, with* $-$ *a complicated category. This functor itself is equal to restriction along* $- \to 1$*, via* $\widehat{1} \cong \mathsf{Set}$*. Replacing* Set *with* FinOrd *thus replaces* Δ *with the analogous functor* $\mathsf{FinOrd} \to \widehat{-}$*, via* $\widehat{1} \cong \mathsf{FinOrd}$*, and thus* Fam *with* Fam_f*.*

Furthermore, because FinOrd *embeds into* Set*, the special strategies of* $\widehat{\mathbb{V}_{[n]}}$ *embed into* $\mathsf{S}_{[n]}$*.*

Then, taking advantage of the fact that FinOrd is a small category, we consider its set FinOrd_0 of objects, i.e., finite ordinals, and the endofunctor F on $\mathsf{Set}/\mathsf{FinOrd}_0$ defined on any family of sets $X = (X_i)_{i \in \mathsf{FinOrd}_0}$ by:

$$(F(X))_n = \sum_{I \in \mathsf{FinOrd}_0} \left(\prod_{M \in \mathcal{M}_n} X_{\mathrm{cod}(M)} \right)^I,$$

where we abusively confuse $[n'] = \mathrm{cod}(M)$ and the natural number n' itself. The isomorphism (7) becomes

$$\mathrm{ob}(\widehat{\mathbb{V}_{[n]}}) \cong (F(\mathrm{ob}(\widehat{\mathbb{V}_{-}})))_n.$$

We may decompose F as follows. Consider the endofunctor on $\mathsf{Set}/\mathsf{FinOrd}_0$ defined by $(\partial X)_n = \prod_{M \in \mathcal{M}_n} X_{\mathrm{cod}(M)}$, for any family $X \in \mathsf{Set}/\mathsf{FinOrd}_0$. We obviously have:

Lemma 1. *F is equal to the composite $(\partial -)^*$.*

This endofunctor is polynomial [27] and we now give a characterisation of its final coalgebra. The rest of this subsection is devoted to proving:

Theorem 3. *The family* $\mathrm{ob}(\widehat{\mathbb{V}_n})$ *formed for each* n *by (the objects of)* $\widehat{\mathbb{V}_n}$ *is a terminal coalgebra for* F*.*

Consider any F-coalgebra $a\colon X \to FX$.

We define by induction on N a sequence of maps $f_N\colon X \to \widehat{\mathbb{V}_{[-]}}$, such that for any view V of length less than N, and any $N' > N$, $f_{N'}(x)(V) = f_N(x)(V)$, and similarly the action of $f_N(x)$ on morphisms is the same as that of $f_{N'}(x)$.

To start the induction, take $f_0(x)$ to be the strategy mapping $id_{[n]}$ to $\pi(a(x))$, i.e., the length of $a(x) \in \sum_{I \in \mathsf{FinOrd}_0}((\partial X)_n)^I$, and all other views to 0.

Furthermore, given f_N, define f_{N+1} to be

$$X \xrightarrow{a} FX \xrightarrow{F(f_N)} F(\widehat{\mathbb{V}_{[-]}}) \xrightarrow{\cong} \widehat{\mathbb{V}_{[-]}},$$

where the equivalence is by temporal decomposition.

Unfolding the definitions yields:

Lemma 2. *Consider any $x \in X_n$, and $a(x) = (z_1, \ldots, z_k)$. For any move $M\colon n \to n'$ and view $V\colon n' \to n''$ of length at most N, and for any $i \in k$, $f_{N+1}(x)(V \circ M) = \sum_{i \in k} f_N(z_i(M))(V)$.*

For any $x \in X_n$, we have a sequence $f_0(x) \hookrightarrow f_1(x) \hookrightarrow \ldots f_N(x) \hookrightarrow f_{N+1}(x) \hookrightarrow \ldots$ which is pointwise stationary. This sequence thus has a colimit in $\widehat{\mathbb{V}_{[n]}}$, the presheaf mapping any view V of length N to $f_N(V)$ (or equivalently $f_{N'}(x)$ for any $N' \geq N$), which allows us to define:

Definition 15. *Let $f\colon X \to \widehat{\mathbb{V}_{[-]}}$ map any $x \in X_n$ to the colimit of the $f_N(x)$'s.*

By construction, we have

Lemma 3. *The following diagram commutes:*

$$\begin{array}{ccc} X & \xrightarrow{\quad a \quad} & FX \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle F(f)} \\ \widehat{\mathbb{V}_{[-]}} & \xleftarrow{\quad \cong \quad} & F(\widehat{\mathbb{V}_{[-]}}). \end{array}$$

Lemma 4. *The set-map f is a map of F-coalgebras.*

Proof: Let, for any innocent strategy $S \in \widehat{\mathbb{V}_{[n]}}$ and $i \in S(id_{[n]})$, $S_{|i}$ be the strategy mapping any view V to the fibre over i of $S(V) \to S(id_{[n]})$. Using the notations of Lemma 2, we must show that for any $i \in k$, we have

$(f(x))_{|i}(V \circ M) = f(z_i(M))(V)$. But Lemma 2 entails that $f(x)(V \circ M) \to f(x)(id_{[n]})$ is actually the coproduct over $i' \in k$ of all $f(z_{i'}(M))(V) \xrightarrow{!} 1 \xrightarrow{i'} \pi(a(x))$, so its fibre over i is indeed $f(z_i(M))(V)$. □

Lemma 5. *The map f is the unique map $X \to \widehat{\mathbb{V}_{[-]}}$ of F-coalgebras.*

Proof: Consider any such map g of coalgebras. It must be such that $g(x)(id_{[n]}) = \pi(a(x))$, and furthermore, using the same notation as before, for any $i \in k$ $(g(x))_{|i}(V \circ M) = g(z_i(M))(V)$, which imposes by induction that $f = g$. □

The last two lemmas directly entail Theorem 3.

4.4 Languages

In particular, the family $\widehat{\mathbb{V}_n}$ supports the operations of the grammar

$$\frac{\ldots\ n \vdash F_i\ \ldots\ (\forall i \in I)}{n \vdash \sum_{i \in I} F_i}\ (I \in \mathsf{FinOrd}_0)$$

$$\frac{\ldots\ n' \vdash F_M\ \ldots\ (\forall M \colon [n] \to [n'] \in \mathcal{M})}{n \vdash \langle M \mapsto F_M \rangle}\ .$$

Here, $n \vdash F$ denotes a presheaf of finite ordinals on $\mathbb{V}_n$. The interpretation is as follows: given presheaves $F_1, \ldots, F_I$, for $I \in \mathsf{FinOrd}_0$, the leftmost rule constructs the finite coproduct $\sum_{i \in I} F_i$ of presheaves (finite coproducts exist in $\widehat{\mathbb{V}_n}$ because they do in FinOrd). In particular, when I is the empty ordinal, we sum over an empty set, so the rule degenerates to

$$\frac{}{n \vdash \emptyset}\ .$$

In terms of presheaves, this is just the constantly empty presheaf.

For the second rule, if for all basic $M \colon [n] \to [n']$, we are given $F_M \in \widehat{\mathbb{V}_{[n']}}$, then $\langle M \mapsto F_M \rangle$ denotes the image under (7) of

$$(1, 1 \mapsto M \mapsto F_M).$$

Here, we provide an element of the right-hand side of (7), consisting of the finite ordinal $I = 1 = \{1\}$, and the function mapping M to $F_M \in \widehat{\mathbb{V}_{[n']}}$ (up to currying). That was for parsing; the intuition is that we construct a

$$\textsc{CCSApp}\quad \frac{}{\Xi;\Gamma \vdash x(a_1,\ldots,a_n)}\ ((x\colon n)\in\Xi \text{ and } a_1,\ldots,a_n\in\Gamma)$$

$$\frac{\Xi;\Gamma,a\vdash P}{\Xi;\Gamma\vdash \nu a.P}\ (a\notin\Gamma) \qquad \frac{\Xi;\Gamma\vdash P \qquad \Xi;\Gamma\vdash Q}{\Xi;\Gamma\vdash P|Q}$$

$$\frac{\ldots \qquad \Xi;\Gamma\vdash P_i \qquad \ldots \qquad (\forall i\in I)}{\Xi;\Gamma\vdash \sum_{i\in I}\alpha_i.P_i}\ (I\in\mathsf{FinOrd}_0 \text{ and } \forall i\in I, \lfloor\alpha_i\rfloor\in\Gamma)$$

$$\textsc{Global}\quad \frac{\Xi;\Delta_1\vdash P_1 \qquad \ldots \qquad \Xi;\Delta_n\vdash P_n \qquad \Xi;\Gamma\vdash P}{\Gamma\vdash \texttt{rec}\ x_1(\Delta_1)\ \texttt{:=}\ P_1,\ldots,x_n(\Delta_n)\ \texttt{:=}\ P_n\ \texttt{in}\ P}$$

Figure 3: CCS syntax

presheaf with one initial state, 1, which maps any view starting with M, say $V\circ M$, to $F_M(V)$. Thus the F_M's specify what remains of our presheaf after each possible basic move. In particular, when all the F_M's are empty, we obtain a presheaf which has an initial state, but which does nothing beyond it. We abbreviate it as $0=\langle\_\mapsto\emptyset\rangle$.

4.5 Translating CCS

It is rather easy to translate CCS into this language. First, define CCS syntax by the natural deduction rules in Figure 3, where Names and Vars are two fixed, disjoint, and infinite sets of *names* and *variables*; Ξ ranges over finite sequences of pairs $(x\colon n)$ of a variable x and its *arity* $n\in\mathsf{FinOrd}_0$, such that the variables are pairwise distinct; Γ ranges over finite sequences of pairwise distinct names; there are two judgements: $\Gamma\vdash P$ for *global* processes, $\Xi;\Gamma\vdash P$ for *open* processes. Rule Global is the only rule for forming global processes, and there $\Xi=(x_1\colon|\Delta_1|,\ldots,x_n\colon|\Delta_n|)$. Finally, α denotes a or $\overline{a}$, for $a\in\mathsf{Names}$, and $\lfloor a\rfloor=\lfloor\overline{a}\rfloor=a$.

First, we define the following (approximation of a) translation on open processes, mapping each open process $\Xi;\Gamma\vdash P$ to $[\![P]\!]\in\widehat{\mathbb{V}_n}$, for $n=|\Gamma|$. This translation ignores the recursive definitions, and we will refine it below to take them into account. We proceed by induction on P, leaving contexts

$\Xi;\Gamma$ implicit:

$$x(a_1,\dots,a_k) \mapsto \emptyset \qquad \nu a.P \mapsto \langle \nu_n \mapsto [\![P]\!], \_ \mapsto \emptyset \rangle$$

$$P|Q \mapsto \langle\ \pi^l_n \mapsto [\![P]\!],\ \pi^r_n \mapsto [\![Q]\!],\ \_ \mapsto \emptyset\ \rangle \qquad \textstyle\sum_{i\in I} \alpha_i.P_i \mapsto \langle\ (\iota^+_{n,j} \mapsto \sum_{k\in I_{\overline{j}}} [\![P_k]\!],\ \iota^-_{n,j} \mapsto \sum_{k\in I_j} [\![P_k]\!]\ )_{j\in n},\ \_ \mapsto \emptyset\ \rangle.$$

Let us explain intuitions and notation. In the first case, we assume implicitly that $(x\colon k) \in \Xi$; the intuition is just that we approximate variables with empty strategies. Next, $P|Q$ is translated to the strategy with one initial state, which only accepts left and right half-forking first, and then lets its avatars play $[\![P]\!]$ and $[\![Q]\!]$, respectively. Similarly, $\nu a.P$ is translated to the strategy with one initial state, accepting only the channel creation move, and then playing $[\![P]\!]$. In the last case, the guarded sum $\sum_{i\in I} \alpha_i.P_i$ is translated to the strategy with one initial state, which

- accepts input on any channel a when $\alpha_i = a$ for some $i \in I$, and output on any channel a when $\alpha_i = \overline{a}$ for some $i \in I$;
- after an input on a, plays the sum of all $[\![P_i]\!]$'s such that $\alpha_i = a$; and after an output on a, plays the sum of all $[\![P_i]\!]$'s such that $\alpha_i = \overline{a}$.

Formally, in the definition, we let, for all $j \in n$, $I_{\overline{j}} = \{i \in I \mid \alpha_i = \overline{a_j}\}$ and $I_j = \{i \in I \mid \alpha_i = a_j\}$. In particular, if $I = \emptyset$, we obtain 0.

Thus, almost all translations of open processes have exactly one initial state, i.e., map the identity view on $[n]$ to the singleton 1. The only exceptions are variable applications, which are mapped to the empty presheaf.

The translation extends to global processes as follows. Fixing a global process $Q = (\texttt{rec}\ x_1(\Delta_1)\ \texttt{:=}\ P_1, \dots, x_k(\Delta_k)\ \texttt{:=}\ P_k\ \texttt{in}\ P)$ typed in Γ with n names, define the sequence $(P^i)_{i\in \mathsf{FinOrd}_0}$ of open processes (all typed in $\Xi;\Gamma$) as follows. First, $P^0 = P$. Then, let $P^{i+1} = \partial P^i$, where ∂ is the *derivation* endomap on open processes typed in any extension $\Xi;(\Gamma,\Delta)$ of $\Xi;\Gamma$, which unfolds one layer of recursive definitions. This map is defined by induction on its argument as follows:

$$\partial(x_l(a_1,\dots,a_{k_l})) = P_l[b_j \mapsto a_j]_{1\le j\le k_l} \qquad \partial(\nu a.P) = \nu a.\partial P$$

$$\partial(P|Q) = \partial P|\partial Q \qquad \partial(\textstyle\sum_{i\in I} \alpha_i.P_i) = \sum_{i\in I} \alpha_i.(\partial P_i),$$

where for all $l \in \{1,\dots,k\}$, $\Delta_l = (b_1,\dots,b_{k_l})$, and $P[\sigma]$ denotes simultaneous, capture-avoiding substitution of names in P by σ.

By construction, the translations of these open processes form a sequence $[\![P^0]\!] \hookrightarrow [\![P^1]\!] \ldots$ of inclusions in $\widehat{\mathbb{V}_n}$, such that for any natural number i and view $V \in \mathbb{V}_n$ of length i (i.e., with i basic moves), $[\![P^j]\!](V)$ is fixed after $j = (k+1)i$, at worst, i.e., for all $j \geq (k+1)i$, $[\![P^j]\!](V) = [\![P^{(k+1)i}]\!](V)$. Thus, this sequence has a colimit in $\widehat{\mathbb{V}_n}$, the presheaf sending any view V of length i to $[\![P^{(k+1)i}]\!](V)$. We put:

Definition 16. *Let the translation of Q be $[\![Q]\!] = \mathrm{colim}_{i \in \mathsf{FinOrd}} [\![P^i]\!]$.*

Which equivalence is induced by this mapping on CCS, especially when taking into account the interactive equivalences developed in the next section? This is the main question we will try to address in future work.

5 Interactive equivalences

5.1 Fair testing vs. must testing: the standard case

An important part of concurrency theory consists in studying *behavioural equivalences*. Since each such equivalence is supposed to define when two processes behave the same, it might seem paradoxical to consider several of them. Van Glabbeek [41] argues that each behavioural equivalence corresponds to a physical scenario for observing processes.

A distinction we wish to make here is between *fair* scenarios, and *potentially unfair* ones. An example of a fair scenario is when parallel composition of processes is thought of as modelling different physical agents, e.g., in a game with several players. Otherwise said, players are really independent. On the other hand, an example of a potentially unfair scenario is when parallelism is implemented via a scheduler.

This has consequences on so-called *testing* equivalences [7]. Let $\heartsuit$ be a fixed action.

Definition 17. *A process P is* must orthogonal *to a context C, notation $P \perp^m C$, when all maximal traces of $C[P]$ play $\heartsuit$ at some point.*

Here, maximal means either infinite or finite without extensions. Let $P^{\perp^m}$ be the set of all contexts must orthogonal to P.

Definition 18. *P and Q are* must equivalent*, notation $P \sim_m Q$, when $P^{\perp^m} = Q^{\perp^m}$.*

In transition systems, or automata, we have $\Omega \sim_m \Omega|\overline{a}$ (where Ω is the looping process, producing infinitely many silent transitions). This might be surprising, because the context $C = a.\heartsuit \mid \square$ intuitively should distinguish these processes, by being orthogonal to $\Omega|\overline{a}$ but not to Ω alone. However, it is not orthogonal to $\Omega|\overline{a}$, because $C[\Omega|\overline{a}]$ has an infinite looping trace giving priority to Ω. This looping trace is unfair, because the synchronisation on a is never performed. Thus, one may view the equivalence $\Omega \sim_m \Omega|\overline{a}$ as exploiting potential unfairness of a hypothetical scheduler.

Usually, concurrency theorists consider this too coarse, and resort to *fair* testing equivalence.

Definition 19. *A process P is* fair orthogonal *to a context C, notation $P \perp^f C$, when all finite traces of $C[P]$ extend to traces that play $\heartsuit$ at some point.*

Again, $P^{\perp^f}$ denotes the set of all contexts fair orthogonal to P.

Definition 20. *P and Q are* fair equivalent*, notation $P \sim_f Q$, when $P^{\perp^f} = Q^{\perp^f}$.*

This solves the issue, i.e., $\Omega \nsim_f \Omega|\overline{a}$.

In summary, the mainstream setting for testing equivalences relies on traces; and the notion of maximality for traces is intrinsically unfair. This is usually rectified by resorting to fair testing equivalence over must testing equivalence. Our setting is more flexible, in the sense that maximal plays are better behaved than maximal traces. In terms of the previous section, this allows viewing the looping trace $\Omega|\overline{a}|a.\heartsuit \xrightarrow{\tau} \Omega|\overline{a}|a.\heartsuit \xrightarrow{\tau} \dots$ as non-maximal. In the next sections, we define an abstract notion of interactive equivalence (still in the particular case of CCS but in our setting) and we instantiate it to define and study the counterparts of must and fair testing equivalences.

5.2 Interactive equivalences

Definition 21. *A play is* closed-world *when it is a composite of closed-world extended moves.*

Equivalently, a play is closed-world when all of its basic moves are part of a closed-world move.

Let $\mathbb{W} \hookrightarrow \mathbb{E}$ be the full subcategory of closed-world plays, $\mathbb{W}(X)$ being the *fibre* over X for the projection functor $\mathbb{W} \to \mathbb{B}$, i.e., the subcategory of

$\mathbb{W}$ consisting of closed-world plays with base X, and morphisms (id_X, k) between them[4].

Let the category of *closed-world behaviours* on X be the category $\mathsf{G}_X = \widehat{\mathbb{W}(X)}$ of presheaves on $\mathbb{W}(X)$. We may now put:

Definition 22. *An* observable criterion *consists for all positions X, of a replete subcategory $\perp\!\!\!\perp_X \hookrightarrow \mathsf{G}_X$.*

Recall that $\perp\!\!\!\perp_X$ being replete means that for all $F \in \perp\!\!\!\perp_X$ and isomorphism $f\colon F \to F'$ in G_X, F' and f are in $\perp\!\!\!\perp_X$.

An observable criterion specifies the class of 'successful', closed-world behaviours. The two criteria considered below are two ways of formalising the idea that a successful behaviour is one in which all accepted closed-world plays are 'successful', in the sense that some player plays the tick move at some point.

We now define interactive equivalences. Recall that $[F, G]$ denotes the amalgamation of F and G, and that right Kan extension along i_Z^{op} induces a functor $\mathrm{Ran}_{i_Z^{op}}\colon \widehat{\mathbb{V}_Z} \to \widehat{\mathbb{E}_Z}$. Furthermore, precomposition with the canonical inclusion $j_Z\colon \mathbb{W}(Z) \hookrightarrow \mathbb{E}_Z$ induces a functor $j_Z^*\colon \widehat{\mathbb{E}_Z} \to \widehat{\mathbb{W}(Z)}$. Composing the two, we obtain a functor $\mathsf{Gl}\colon \mathsf{S}_Z \to \mathsf{G}_Z$:

$$\mathsf{S}_Z = \widehat{\mathbb{V}_Z} \xrightarrow{\mathrm{Ran}_{i_Z^{op}}} \widehat{\mathbb{E}_Z} \xrightarrow{j_Z^*} \widehat{\mathbb{W}(Z)} = \mathsf{G}_Z.$$

Definition 23. *For any innocent strategy F on X and any pushout square P of positions as on the right, with I consisting only of channels, let $F^{\perp\!\!\!\perp_P}$ be the class of all innocent strategies G on Y such that $\mathsf{Gl}([F, G]) \in \perp\!\!\!\perp_Z$.*

$$\begin{array}{ccc} I & \longrightarrow & Y \\ \downarrow & & \ulcorner\downarrow \\ X & \longrightarrow & Z \end{array} \qquad (8)$$

Here, G is thought of as a *test* for F. Also, P denotes the whole pushout square and $F^{\perp\!\!\!\perp_P}$ denotes all the valid tests for the considered pushout square P. From the CCS point of view, I corresponds to the set of names shared by the process under observation (F) and the test (G).

Definition 24. *Any two innocent strategies $F, F' \in \mathsf{S}_X$ are* $\perp\!\!\!\perp$-equivalent*, notation $F \sim_{\perp\!\!\!\perp} F'$, iff for all pushouts P as in 8, $F^{\perp\!\!\!\perp_P} = F'^{\perp\!\!\!\perp_P}$.*

[4]This is not exactly equivalent to what could be noted $\mathbb{W}_X$, since in the latter there are objects $U \hookleftarrow Y \hookrightarrow X$ with a strict inclusion $Y \hookrightarrow X$. However, both should be equivalent for what we do in this paper, i.e., fair and must equivalences.

5.3 Fair vs. must

Let us now define fair and must testing equivalences. Let a closed-world play be *successful* when it contains a $\heartsuit_n$. Furthermore, for any closed-world behaviour $G \in \mathsf{G}_X$ and closed-world play $U \in \mathbb{W}(X)$, an *extension* of a state $\sigma \in G(U)$ to U' is a $\sigma' \in G(U')$ with $i \colon U \to U'$ and $G(i)(\sigma') = \sigma$. The extension σ' is *successful* when U' is. The intuition is that the behaviour G, before reaching U' with state σ', passed through U with state σ.

Definition 25. *The* fair *criterion $\perp\!\!\!\perp^f$ contains all closed-world behaviours G such that any state $\sigma \in G(U)$ for finite U admits a successful extension.*

Now call an extension of $\sigma \in G(U)$ *strict* when $U \to U'$ is not surjective, or, equivalently, when U' contains more moves than U. For any closed-world behaviour $G \in \mathsf{G}_X$, a state $\sigma \in G(U)$ is *G-maximal* when it has no strict extension.

Definition 26. *Let the* must *criterion $\perp\!\!\!\perp^m$ consist of all closed-world behaviours G such that for all closed-world U and G-maximal $\sigma \in G(U)$, U is successful.*

As explained in the introduction and Section 5.1, unlike in the standard setting, this definition of must testing equivalence distinguishes between the processes Ω and $\Omega|\overline{a}$. Indeed, take the CCS context $C = a.\heartsuit \mid \Box$, which we can implement by choosing as a test the strategy $T = [\![a.\heartsuit]\!]$ on a single player knowing one channel a. Taking I to consist of the sole channel a, the pushout Z as in Definition 23 consists of two players, say x for the observed strategy and y for the test strategy, sharing the channel a. Now, assuming that Ω loops deterministically, the global behaviour $G = \mathsf{GI}([[\![P]\!], T])$ has exactly one state on the identity play, and again exactly one state on the play π_1 consisting of only one fork move by x. Thus, G reaches a position with three players, say x_1 playing Ω, x_2 playing $\overline{a}$, and y playing $a.\heartsuit$. The play with infinitely many silent moves by x_1 is not maximal: we could insert (anywhere in the sequence of moves by x_1) a synchronisation move by x_2 and y, and then a tick move by the avatar of y. Essentially: our notion of play is more fair than just traces.

To get more intuition about must testing equivalence in our setting, we prove that it actually coincides with the testing equivalence generated by the following criterion:

Definition 27. *The* spatially fair *criterion $\perp\!\!\!\perp^{sf}$ contains all closed-world behaviours G such that any state $\sigma \in G(U)$ admits a successful extension.*

This criterion is almost like the fair criterion, except that we do not restrict to finite plays. The key result to show the equivalence is:

Theorem 4. *For any innocent strategy F on X, any state $\sigma \in \mathsf{GI}(F)(U)$ admits a $\mathsf{GI}(F)$-maximal extension.*

The proof is in Appendix B. Thanks to the theorem, we have:

Lemma 6. *For all $F \in \mathsf{S}_X$, $\mathsf{GI}(F) \in \perp\!\!\!\perp^m_X$ iff $\mathsf{GI}(F) \in \perp\!\!\!\perp^{sf}_X$.*

Proof: Let $G = \mathsf{GI}(F)$.

($\Rightarrow$) By Theorem 4, any state $\sigma \in G(U)$ has a G-maximal extension $\sigma' \in G(U')$, which is successful by hypothesis, hence σ has a successful extension.

($\Leftarrow$) Any G-maximal $\sigma \in G(U)$ admits by hypothesis a successful extension which may only be on U by G-maximality, and hence U is successful. □ (Note that U is not necessarily finite in the proof of the right-to-left implication, so that the argument does not apply to the fair criterion.)

Now comes the expected result:

Theorem 5. *For all $F, F' \in \mathsf{S}_X$, $F \sim_{\perp\!\!\!\perp^m} F'$ iff $F \sim_{\perp\!\!\!\perp^{sf}} F'$.*

Proof: *($\Rightarrow$)* Consider two innocent strategies F and F' on X, and an innocent strategy G on Y (as in the pushout (8)). We have, using Lemma 6:

$$\begin{aligned} \mathsf{GI}(F \parallel G) \in \perp\!\!\!\perp^{sf} \quad &\text{iff} \quad \mathsf{GI}(F \parallel G) \in \perp\!\!\!\perp^{m} \\ &\text{iff} \quad \mathsf{GI}(F' \parallel G) \in \perp\!\!\!\perp^{m} \\ &\text{iff} \quad \mathsf{GI}(F' \parallel G) \in \perp\!\!\!\perp^{sf} \end{aligned}$$

($\Leftarrow$) Symmetric. □

Intuitively, must testing only consider spatially fair schedulings, in the sense that all players appearing in a play should be given the opportunity to play: no one should starve.

However, this is not the only source of unfairness, so that must testing and fair testing differ. To see this, consider the CCS process $P = \nu b.\texttt{rec}\ x(a,b) := \overline{b}|(b.(x(a,b)) + \overline{a})\ \texttt{in}\ x(a,b)$, that can repeatedly perform synchronisations on the private channel b, until it chooses to perform an output on a. We have $[\![\Omega]\!] \sim^{sf} [\![P]\!]$ while $[\![\Omega]\!] \not\sim^{f} [\![P]\!]$. Indeed, since the choice between doing a synchronisation on b or an output on a is done by a single player, the infinite play where the output on a is never performed is

maximal: no player starve, we just have a player that repeatedly chooses the same branch, in an unfair way.

We leave for future work the investigation of such unfair scenarios and their correlation to the corresponding behaviours in classical presentations of CCS.

A Temporal decomposition

This section is a proof of Theorem 2. Let us first review the general equivalences mentioned in the proof sketch. The product of a family of presheaf categories is isomorphic to the category of presheaves over the corresponding coproduct of categories:

Lemma 7. *We have* $\prod_{M \in \mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)} \cong [\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}]$.

Furthermore, let the functor $\Delta \colon \mathsf{Set} \to \widehat{\mathbb{C}}$ map any set X to the constant presheaf mapping any $C \in \mathbb{C}$ to X. We have:

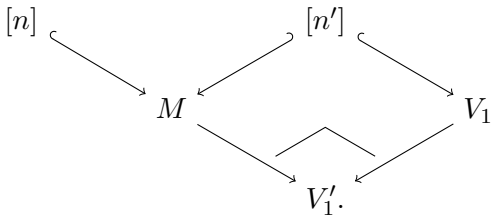
Lemma 8. *For any small category* $\mathbb{C}$, $\mathrm{Fam}(\widehat{\mathbb{C}}) \simeq (\widehat{\mathbb{C}} \downarrow \Delta)$.

Proof: A generalisation of the more well-known $\mathsf{Set}^X \simeq \mathsf{Set}/X$. □

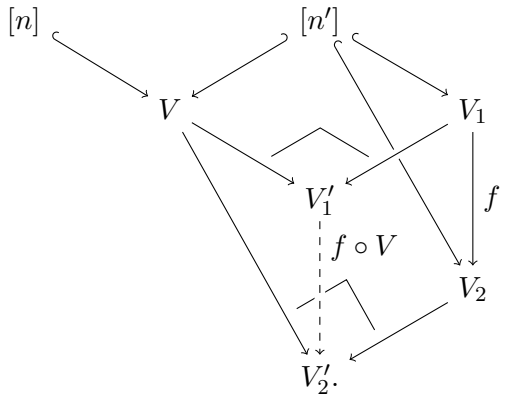
Corollary 1. *We have:*

$$\mathrm{Fam}\left(\prod_{M \in \mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)}\right) \simeq ([\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}] \downarrow \Delta).$$

We now construct the lax pushout (6). A first step is the construction, for each move $[n] \hookrightarrow M \hookleftarrow [n']$, of a functor $(- \circ M) \colon \mathbb{V}_{[n']} \to \mathbb{V}_{[n]}$ given by precomposition with M in $\mathsf{Cospan}(\widehat{\mathbb{C}})$. This functor maps any $V_1 \colon [n'] \hookrightarrow V_1$ to the view $V_1 \circ M$, i.e., the view $[n] \hookrightarrow V_1'$ defined by the colimit



This of course relies on the choice of such a colimit for every V and V_1. Any morphism $f\colon V_1 \to V_2$ in $\mathbb{V}_{[n']}$, letting $V_2' = V_2 \circ V$, is mapped to the dashed arrow induced by universal property of pushout in



Once the choice has been made on objects, the arrow map is determined uniquely.

This family of functors allows us to decompose $\mathbb{V}_{[n]}$ as follows:

Lemma 9. *The diagram*

$$\begin{array}{ccc} \sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)} & = & \sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)} \\ {\scriptstyle !}\downarrow & \overset{\lambda}{\swarrow} & \downarrow{\scriptstyle [-\circ M]_{M\in\mathcal{M}_n}} \\ 1 & \xrightarrow[\ulcorner id_{[n]} \urcorner]{} & \mathbb{V}^{op}_{[n]} \end{array} \tag{9}$$

is a lax pushout, where $\lambda_{M,V}\colon id_{[n]} \to M \circ V$, seen in $\mathbb{V}_{[n]}$, is the obvious inclusion.

Proof: For any category $\mathbb{C}$, taking such a lax pushout of $id_{\mathbb{C}}$ with 1 just adds a terminal object to $\mathbb{C}$. The rest is an easy verification. A dual result of course holds with $\mathbb{V}_{[n]}$, reversing the direction of λ. □

Now, it is well-known that, in any small 2-category $\mathbb{K}$, any contravariant hom-2-functor, i.e., 2-functor of the shape $\mathbb{K}(-, X)$ for $X \in K$, maps weighted colimits in $\mathbb{K}$ to weighted limits in Cat. For an introduction to weighted limits and colimits in the case of enrichment over Cat, see Kelly [26]. Here, for any 2-category P, and 2-functors $G\colon P \to \mathbb{K}$ and $J\colon P^{op} \to \mathsf{Cat}$, any colimit $L = J \star G$ of G weighted by J with unit $\xi\colon J \to \mathbb{K}(G(-), L)$ in

$[P^{op}, \mathsf{Cat}]$ is mapped, for any object $X \in \mathbb{K}$, by the hom-2-functor $\mathbb{K}(-, X)$ to a limit of $\mathbb{K}(G(-), X)\colon P^{op} \to \mathsf{Cat}$ weighted by J in Cat, with unit $\mathbb{K}(\xi, X)\colon J \to \mathsf{Cat}(\mathbb{K}(L, X), \mathbb{K}(G(-), X))$, in Cat. In particular, lax pushouts are mapped to lax pullbacks. As usual, considering a larger universe, we may replace Cat with CAT and obtain the same results with $\mathbb{K} = \mathsf{Cat}$.

Recalling our lax pushout (9) and taking the hom-categories to Set, we obtain a lax pullback

$$\begin{array}{ccc} [\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}] & = & [\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}] \\ {\scriptstyle !^*}\uparrow & \swarrow^{\lambda^*} & \uparrow \\ \mathsf{Set} & \longleftarrow & \mathsf{S}_{[n]} \end{array}$$

in CAT, i.e., a comma category. But observe that restriction along ! is precisely $\Delta\colon \mathsf{Set} \to [\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}]$, so we have indeed shown that $\mathsf{S}_{[n]}$ is a comma category $[\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}] \downarrow \Delta$.

B Maximal extensions

This section is a proof of Theorem 4.

Lemma 10. *For any position X, the category $\mathbb{W}(X)$ of closed-world plays is a preorder.*

Proof: Easy. □

In the following, we consider the quotient poset.

Lemma 11. *In $\mathbb{W}(X)$, any non-decreasing chain admits an upper bound.*

Recall $\mathcal{M}$, the graph of all basic moves, and the set $\mathcal{M}_n$ of edges from n, for each n. Let now, for each n, $\mathcal{M}_n^f$ be the analogous set with full moves, i.e., the set of isomorphism classes of full moves from $[n]$.

Lemma 12. *For each play $U \in \mathbb{E}_X$, the coproduct of all s maps from full moves*

$$\left(\sum_{n \in \mathsf{FinOrd}} \sum_{M \in \mathcal{M}_n^f} U(M) \right) \to \sum_{n \in \mathsf{FinOrd}} U[n], \tag{10}$$

is injective.

Recall here that for forking, we have also called s the common composite $l \circ s = r \circ s$ (see the discussion following Definition 3).

Proof: By induction on U. □

Lemma 13. *Any non-decreasing sequence in the poset $\mathbb{W}(X)$ admits its colimit in $\widehat{\mathbb{C}}$ as an upper bound.*

Proof: Consider any increasing sequence $U^1 \hookrightarrow U^2 \hookrightarrow \ldots$ of plays in $\mathbb{W}(X)$. Let U be its colimit in $\widehat{\mathbb{C}}$. We want to prove that U is a play.

First, observe that U satisfies joint injectivity of s-maps as in Lemma 12: indeed, if we had a player p and two full moves M and M' such that $s(M) = s(M') = p$, then all of M, M', and p would appear in some U^i, which, being a play, has to satisfy joint injectivity.

For each n, U^n comes with a sequence of compatible (closed-world) extended moves

$$X = X_0^n \hookrightarrow M_1^n \hookleftarrow X_1^n \hookrightarrow \ldots \hookleftarrow X_{i-1}^n \hookrightarrow M_i^n \hookleftarrow X_i^n \hookrightarrow \ldots$$

which are also (by the colimit cocone) morphisms above U in $\widehat{\mathbb{C}}$. For each $i \geq 1$, taking the colimit of the i first moves yields a finite play $X \hookrightarrow U_i^n \hookleftarrow X_i^n$. By convention, letting $U_0^n = X$ extends this to $i \geq 0$. Similarly, we may consider all the given plays infinite, by accepting not only extended moves, but also identity cospans.

We consider the poset of pairs $(N, n) \in \{(0,0)\} \uplus \sum_{N \in \mathsf{FinOrd}^*} N$, with lexicographic order, i.e., $(N, n) \leq (N', n')$ when $N < N'$ or when $N = N'$ and $n \leq n'$.

We will construct by induction on (N, n) a sequence of composable closed-world moves, with colimit U', such that for all (N, n), $U^n_{N-n+1} \subseteq U'$ in $\mathbb{W}(X)/U$. More precisely, we construct for each (N, n) an integer $K_{N,n}$ and a sequence

$$X = X_0^{N,n} \hookrightarrow M_1^{N,n} \hookleftarrow X_1^{N,n} \hookrightarrow \ldots \hookleftarrow X_{K_{N,n}-1}^{N,n} \hookrightarrow M_{K_{N,n}}^{N,n} \hookleftarrow X_{K_{N,n}}^{N,n},$$

(again, if $K_{N,n} = 0$, we mean the empty sequence) such that

- for all $(N', n') < (N, n)$, we have $K_{N',n'} \leq K_{N,n}$ and the sequence $(M_i^{N',n'})_{i \in K_{N',n'}}$ is a prefix of $(M_{i \in K_{N,n}}^{N,n})$;
- and the colimit, say $U_{N,n}$, of $(M_i^{N,n})_{i \in K_{N,n}}$ is such that for all $(N', n') \leq (N, n)$, $U^{n'}_{N-n'+1} \subseteq U_{N,n}$ in $\mathbb{W}(X)/U$.

For the base case, we let $K_{0,0} = 0$, which forces $M^{0,0}$ to be the empty sequence on X.

For the induction step, consider any $(N, n) \neq (0, 0)$, and let (N_0, n_0) be the predecessor of (N, n). The induction hypothesis gives a K_{N_0,n_0} and a sequence $(M_i^{N_0,n_0})_{i \in K_{N_0,n_0}}$ satisfying some hypotheses, among which the existence of a diagram

$$\begin{array}{ccccccc} X & \longrightarrow & U^n_{N-n} & \longleftarrow & X^n_{N-n} & \longrightarrow & M^n_{N-n+1} \longleftarrow X^n_{N-n+1} \\ \| & & \downarrow & & & & \\ X & \longrightarrow & U_{N_0,n_0} & \longleftarrow & X^{N_0,n_0}_{K_{N_0,n_0}} & & \end{array}$$

above U.

Now, if $M^n_{N-n+1} \to U$ factors through U_{N_0,n_0}, then we put $K_{N,n} = K_{N_0,n_0}$ and $(M_i^{N,n})_{i \in K_{N,n}} = (M_i^{N_0,n_0})_{i \in K_{N_0,n_0}}$, and all induction hypotheses go through.

Otherwise, M^n_{N-n+1} is played by players in X^n_{N-n} which are not in the joint image of all s maps (10) in U_{N_0,n_0}, otherwise s maps in U could not be jointly injective, contradicting Lemma 12. Technically, the diagram

$$X^n_{N-n} \to M^n_{N-n+1} \leftarrow X^n_{N-n+1}$$

is obtained by pushing some (non-extended) closed-world move $Y \to M \leftarrow Y'$ along some morphism $I \to Z$ from an interface I, and the induced morphism $Y \to X^n_{N-n} \to U^n_{N-n} \to U_{N_0,n_0}$ factors through $X^{N_0,n_0}_{K_{N_0,n_0}}$. We consider the subposition $Z' \subseteq X^{N_0,n_0}_{K_{N_0,n_0}}$ making

$$\begin{array}{ccc} I & \hookrightarrow & Y \\ \downarrow & & \downarrow \\ Z' & \hookrightarrow & X^{N_0,n_0}_{K_{N_0,n_0}} \end{array}$$

a pushout; Z' consists of the players in $X^{N_0,n_0}_{K_{N_0,n_0}}$ that are not in the image of Y, plus their names, plus possibly missing names from I.

Then, pushing $Y \to M \leftarrow Y'$ along $I \to Z'$, we obtain an extended move $X^{N_0,n_0}_{K_{N_0,n_0}} \hookrightarrow M' \hookleftarrow X'$. We let $K_{N,n} = K_{N_0,n_0} + 1$ and define $(M_i^{N,n})_{i \in K_{N,n}}$ to be the extension of $(M_i^{N_0,n_0})_{i \in K_{N_0,n_0}}$ by M'. This induces a unique map $U_{N,n} \to U$ by universal property of $U_{N,n}$ as a colimit. All induction hypotheses go through; in particular, U^n_{N-n+1} is a union $U^n_{N-n} \cup M^n_{N-n+1}$ in

$\mathbb{W}(X)/U$, and actually a union $U^n_{N-n} \cup M$; similarly, $U_{N,n} = U_{N_0,n_0} \cup M$; so, since we have $U^n_{N-n} \subseteq U_{N_0,n_0}$ by induction hypothesis, we obtain $U^n_{N-n+1} \subseteq U_{N,n}$.

The sequences $M^{N,n}$ induce by union a possibly infinite sequence of closed-world extended moves, i.e., a closed-world play U', such that for all (N,n), $U^n_{N-n+1} \subseteq U'$, hence, for all n, $U^n \subseteq U' \subseteq U$, i.e., $U' \cong U$. Thus, U is indeed a play. □

We are almost ready for proving Theorem 4. We just need one more lemma. Consider any innocent strategy F on X, play $U \in \mathbb{W}(X)$, and any state $\sigma \in \mathsf{GI}(F)(U)$. Consider now the poset F_σ of $\mathsf{GI}(F)$-extensions of σ (made into a poset by choosing a skeleton of $\mathbb{W}(X)$), where $\sigma' \in F(U') \leq \sigma'' \in F(U'')$ iff $U' \leq U''$. This poset is not empty, since it contains σ. Furthermore, we have:

Lemma 14. *Any non-decreasing sequence in F_σ admits an upper bound.*

Proof: Any such sequence, say $(\sigma_i)_{i \in \mathsf{FinOrd}}$, induces a non-decreasing sequence of plays in $\mathbb{W}(X)$, say $(U_i)_i$, which by Lemma 13 admits its colimit, say U', as an upper bound. Now, any view inclusion $j \colon V \hookrightarrow U'$, factors through some U_i, and we let $\sigma_j = (\sigma_i)_{|V}$ (this does not depend on the choice of i). This assignment determines (by innocence of F and by construction of the right Kan extension as an end) an element $\sigma' \in F(U')$, which is an upper bound for $(\sigma_i)_{i \in \mathsf{FinOrd}}$. □

Proof of Theorem 4: Consider any innocent strategy F on X, play $U \in \mathbb{W}(X)$, and any state $\sigma \in \mathsf{GI}(F)(U)$. Consider as above the poset F_σ of $\mathsf{GI}(F)$-extensions of σ. By the last lemma, we may apply Zorn's lemma to choose a maximal element of F_σ, which is a $\mathsf{GI}(F)$-maximal extension of σ. □

References

[1] Emmanuel Beffara. *Logique, réalisabilité et concurrence.* PhD thesis, Université Paris 7, December 2005.

[2] Marcello M. Bonsangue, Jan J. M. M. Rutten, and Alexandra Silva. A kleene theorem for polynomial coalgebras. In Luca de Alfaro, editor, *FOSSACS*, volume 5504 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2009.

[3] Ed Brinksma, Arend Rensink, and Walter Vogler. Fair testing. In Insup Lee and Scott A. Smolka, editors, *CONCUR*, volume 962 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 1995.

[4] Albert Burroni. Higher-dimensional word problems with applications to equational logic. *Theoretical Computer Science*, 115(1):43–62, 1993.

[5] Aurelio Carboni and Peter Johnstone. Connected limits, familial representability and artin glueing. *Mathematical Structures in Computer Science*, 5(4):441–459, 1995.

[6] Aurelio Carboni and Peter Johnstone. Corrigenda for 'connected limits, familial representability and artin glueing'. *Mathematical Structures in Computer Science*, 14(1):185–187, 2004.

[7] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.

[8] Olivier Delande and Dale Miller. A neutral approach to proof and refutation in mall. In *LICS '08* [29], pages 498–508.

[9] H. Ehrig, H.-J. Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism and Distribution*. World Scientific, 1999.

[10] Marcelo P. Fiore. Second-order and dependently-sorted abstract syntax. In *LICS '08* [29], pages 57–68.

[11] Fabio Gadducci, Reiko Heckel, and Mercè Llabrés. A bi-categorical axiomatisation of concurrent graph rewriting. *Electronic Notes in Theoretical Computer Science*, 29, 1999.

[12] Fabio Gadducci and Ugo Montanari. The tile model. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 133–166. The MIT Press, 2000.

[13] Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[14] Yves Guiraud and Philippe Malbos. Higher-dimensional categories with finite derivation type. *Theory and Applications of Categories*, 22(18):420–278, 2009.

[15] André Hirschowitz, Michel Hirschowitz, and Tom Hirschowitz. Contraction-free proofs and finitary games for linear logic. *Electronic Notes in Theoretical Computer Science*, 249:287–305, 2009.

[16] André Hirschowitz and Marco Maggesi. Modules over monads and linearity. In Daniel Leivant and Ruy J. G. B. de Queiroz, editors, *WoLLIC*, volume 4576 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2007.

[17] André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Information and Computation*, 208(5):545–564, 2010.

[18] Tom Hirschowitz. Cartesian closed 2-categories and permutation equivalence in higher-order rewriting. Preprint. `http://hal.archives-ouvertes.fr/hal-00540205/en/`.

[19] Tom Hirschowitz and Damien Pous. Innocent strategies as presheaves and interactive equivalences for CCS. In Alexandra Silva, Simon Bliudze, Roberto Bruni, and Marco Carbone, editors, *ICE*, volume 59 of *EPTCS*, pages 2–24, 2011.

[20] Martin Hyland. *Semantics and Logics of Computation*, chapter Game Semantics. Cambridge University Press, 1997.

[21] Bart Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

[22] Ole H. Jensen and Robin Milner. Bigraphs and mobile processes (revised). Technical Report TR580, University of Cambridge, 2004.

[23] P. T. Johnstone, S. Lack, and P. Sobociński. Quasitoposes, quasiadhesive categories and Artin glueing. In *CALCO*, volume 4624 of *LNCS*, pages 312–326. Springer Verlag, 2007.

[24] André Joyal, Mogens Nielsen, and Glynn Winskel. Bisimulation and open maps. In *LICS '93*, pages 418–427. IEEE Computer Society, 1993.

[25] Stefano Kasangian and Anna Labella. Observational trees as models for concurrency. *Mathematical Structures in Computer Science*, 9(6):687–718, 1999.

[26] G. M. Kelly. Elementary observations on 2-categorical limits. *Bulletin of the Australian Mathematical Society*, 39:301–317, 1989.

[27] Joachim Kock. Polynomial functors and trees. *International Mathematics Research Notices*, 2011(3):609–673, 2011.

[28] Jean-Louis Krivine. Dependent choice, 'quote' and the clock. *Theor. Comput. Sci.*, 308(1-3):259–276, 2003.

[29] *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 2008.

[30] Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer, 2nd edition, 1998.

[31] Saunders MacLane and Ieke Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer, 1992.

[32] Paul-André Melliès. Asynchronous games 2: the true concurrency of innocence. In *Proc. CONCUR '04*, volume 3170 of *LNCS*, pages 448–465. Springer Verlag, 2004.

[33] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[34] V. Natarajan and Rance Cleaveland. Divergence and fair testing. In Zoltán Fülöp and Ferenc Gécseg, editors, *ICALP*, volume 944 of *Lecture Notes in Computer Science*, pages 648–659. Springer, 1995.

[35] Tobias Nipkow. Higher-order critical pairs. In *LICS '91*, pages 342–349. IEEE Computer Society, 1991.

[36] Gordon D. Plotkin. A structural approach to operational semantics. DAIMI Report FN-19, Computer Science Department, Aarhus University, 1981.

[37] Julian Rathke and Pawel Sobocinski. Deconstructing behavioural theories of mobility. In *IFIP TCS*, volume 273 of *IFIP*, pages 507–520. Springer, 2008.

[38] Vladimiro Sassone and Pawel Sobociński. Deriving bisimulation congruences using 2-categories. *Nordic Journal of Computing*, 10(2), 2003.

[39] Peter Sewell. From rewrite to bisimulation congruences. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 1998.

[40] Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *LICS '97*, pages 280–291, 1997.

[41] Rob J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990.

[42] Angelo Vistoli. Notes on Grothendieck topologies, fibered categories and descent theory. Preprint. `http://arxiv.org/abs/math/0412512`, 2007.

UNTYPING TYPED ALGEBRAS AND COLOURING CYCLIC LINEAR LOGIC

DAMIEN POUS

CNRS (LIG, UMR 5217, Grenoble)

Abstract. We prove "untyping" theorems: in some typed theories (semirings, Kleene algebras, residuated lattices, involutive residuated lattices), typed equations can be derived from the underlying untyped equations. As a consequence, the corresponding untyped decision procedures can be extended for free to the typed settings. Some of these theorems are obtained via a detour through fragments of cyclic linear logic, and give rise to a substantial optimisation of standard proof search algorithms.

Introduction

Motivations. The literature contains many decidability or complexity results for various algebraic structures. Some of these structures (rings, Kleene algebras [22], residuated lattices [31]) can be generalised to *typed* structures, where the elements come with a domain and a codomain, and where operations are defined only when these domains and codomains agree according to some simple rules. Although such typed structures are frequently encountered in practice (e.g., rectangular matrices, heterogeneous binary relations, or more generally, categories), there are apparently no proper tools to easily reason about these.

This is notably problematic in proof assistants, where powerful decision procedures are required to let the user focus on difficult reasoning steps by leaving administrative details to the computer. Indeed, although some important theories can be decided automatically in Coq or HOL (e.g., Presburger arithmetic [29], elementary real algebra [15], rings [14]), there are no high-level tools to reason about heterogeneous relations or rectangular matrices.

In this paper, we show how to extend the standard decision procedures from the untyped structures to the corresponding typed structures. In particular, we make it possible to use standard tools to reason about rectangular matrices or heterogeneous relations, without bothering about types (i.e., matrix dimensions or domain/codomain information). The approach we propose is depicted below: we study "untyping" theorems that allow one to

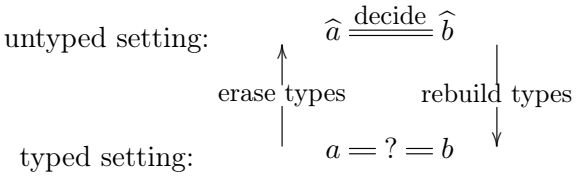
1998 ACM Subject Classification: F.4.1 [Proof theory], F.4.3 [Decision problems].

Key words and phrases: involutive residuated lattices, cyclic linear logic, Kleene algebra, typed algebra, decision procedures, sequent calculus, proof search.

Extended version of the abstract that appeared in Proc. CSL'10 [34].

Partially funded by the French projects "Choco", ANR-07-BLAN-0324 and "PiCoq", ANR-10-BLAN-0305.

prove typed equations as follows: 1) erase type informations, 2) prove the equation using standard, untyped, decision procedures, and 3) derive a typed proof from the untyped one.



Besides the theoretical aspects, an important motivation behind this work comes from a Coq library [5] in which we developed efficient tactics for partial axiomatisations of relations: the ideas presented here were used and integrated in this library to extend our tactics to typed structures, for free.

Overview. We shall mainly focus on the two algebraic structures we mentioned above, since they raise different problems and illustrate several aspects of these untyping theorems: Kleene algebras [21] and residuated lattices [19].

- The case of Kleene algebras is the simplest one. The main difficulty comes from the annihilating element (0): its polymorphic typing rule requires us to show that equational proofs can be factorised so as to use the annihilation laws at first, and then reason using the other axioms.
- The case of residuated structures is more involved: due to the particular form of axioms about residuals, we cannot rely on standard equational axiomatisations of these structures. Instead, we need to exploit an equivalent cut-free sequent proof system (first proposed by Ono and Komori [31]), and to notice that this proof system corresponds to the intuitionistic fragment of cyclic linear logic [40]. The latter logic is much more concise and the corresponding proof nets are easier to reason about, so that we obtain the untyping theorem in this setting. We finally port the result back to residuated lattices by standard means.

The above sequent proof systems have the sub-formula property, so that they yield decision procedures, using proof search algorithms. As an unexpected application, we show that the untyping theorem makes it possible to improve these algorithms by reducing the set of proofs that have to be explored.

Outline. We introduce our notations and make the notion of typed structure precise in §1. We study Kleene algebras and residuated lattices in §2 and §3, respectively. The optimisation of proof search is analysed in §4; we conclude with related work, and directions for future work in §5.

1. Notation, typed structures

Let $\mathcal{X}$ be an arbitrary set of *variables*, ranged over using letters x, y. Given a signature Σ, we let a, b, c range over the set $T(\Sigma + \mathcal{X})$ of *terms with variables*. Given a set $\mathcal{T}$ of *objects* (ranged over using letters n, m, p, q), a *type* is a pair (n, m) of objects (which we denote by $n \to m$, following categorical notation), a *type environment* $\Gamma : \mathcal{X} \to \mathcal{T}^2$ is a function from variables to types, and we will define *type judgements* of the form $\Gamma \vdash a : n \to m$, to be read "in environment Γ, term a has type $n \to m$, or, equivalently, a is a morphism from n to m".

By $\Gamma \vdash a, b : n \to m$, we mean that both a and b have type $n \to m$; type judgements will include the following rule for variables:

$$\frac{\Gamma(x) = (n, m)}{\Gamma \vdash x : n \to m}\ \textsc{Tv}$$

Similarly, we will define *typed equality* judgements of the form $\Gamma \vdash a = b : n \to m$: "in environment Γ, terms a and b are equal, at type $n \to m$". Equality judgements will generally include the following rules, so as to obtain an equivalence relation at each type:

$$\frac{\Gamma(x) = (n, m)}{\Gamma \vdash x = x : n \to m}\ \textsc{v} \qquad \frac{\Gamma \vdash a = b : n \to m \quad \Gamma \vdash b = c : n \to m}{\Gamma \vdash a = c : n \to m}\ \textsc{t} \qquad \frac{\Gamma \vdash a = b : n \to m}{\Gamma \vdash b = a : n \to m}\ \textsc{s}$$

By taking the singleton set as set of objects ($\mathcal{T} = \{\emptyset\}$), we recover standard, untyped structures: the only typing environment is $\widehat{\emptyset} : x \mapsto (\emptyset, \emptyset)$, and types become uninformative (this corresponds to working in a one-object category; all operations are total functions). To alleviate notations, since the typing environment will always be either $\widehat{\emptyset}$ or an abstract constant value Γ, we shall leave it implicit in type and equality judgements, by relying on the absence or the presence of types to indicate which one to use. For example, we shall write $\vdash a = b : n \to m$ for $\Gamma \vdash a = b : n \to m$, while $\vdash a = b$ will denote the judgement $\widehat{\emptyset} \vdash a = b : \emptyset \to \emptyset$.

The question we study in this paper is the following one: given a signature and a set of inference rules defining a type judgement and an equality judgement, does the implication below hold, for all a, b, n, m?

$$\begin{cases} \vdash a, b : n \to m \\ \vdash a = b \end{cases} \qquad \text{entails} \qquad \vdash a = b : n \to m\ .$$

In other words, in order to prove an equality in a typed structure, is it safe to remove all type annotations, so as to work in the untyped underlying structure?

2. Kleene algebras

We study the case of residuated lattices in §3; here we focus on Kleene algebras. In order to illustrate our methodology, we actually give the proof in three steps, by considering two intermediate algebraic structures: monoids and semirings. The former admit a rather simple and direct proof, while the latter are sufficient to expose concisely the main difficulty in handling Kleene algebras.

2.1. **Monoids.**

Definition 2.1. *Typed monoids* are defined by the signature $\{\cdot_2, 1_0\}$, together with the following inference rules, in addition to the rules from §1.

$$\frac{}{\vdash 1 : n \to n}\ \textsc{To} \qquad \frac{\vdash a : n \to m \qquad \vdash b : m \to p}{\vdash a \cdot b : n \to p}\ \textsc{Td} \qquad \frac{}{\vdash 1 = 1 : n \to n}\ \textsc{o}$$

$$\frac{\vdash a = a' : n \to m \qquad \vdash b = b' : m \to p}{\vdash a \cdot b = a' \cdot b' : n \to p}\ \textsc{d} \qquad \frac{\vdash a : n \to m}{\vdash 1 \cdot a = a : n \to m}\ \textsc{od}$$

$$\frac{\vdash a : n \to m \qquad \vdash b : m \to p \qquad \vdash c : p \to q}{\vdash (a \cdot b) \cdot c = a \cdot (b \cdot c) : n \to q}\ \textsc{da} \qquad \frac{\vdash a : n \to m}{\vdash a \cdot 1 = a : n \to m}\ \textsc{do}$$

In other words, typed monoids are just categories: 1 and $\cdot$ correspond to identities and composition. Rules (O) and (D) ensure that equality is reflexive at each type (point (i) below) and preserved by composition. As expected, equalities relate correctly typed terms only (ii):

Lemma 2.2.

(i) If $\vdash a : n \to m$, *then* $\vdash a = a : n \to m$.
(ii) If $\vdash a = b : n \to m$, *then* $\vdash a, b : n \to m$.

Moreover, in this setting, type judgements enjoy some form of injectivity (types are not uniquely determined due to the unit (1), which is typed in a polymorphic way):

Lemma 2.3. *If* $\vdash a : n \to m$ *and* $\vdash a : n' \to m'$, *then we have* $n = n'$ *iff* $m = m'$.

We need another lemma to obtain the untyping theorem: all terms related by the untyped equality admit the same type derivations.

Lemma 2.4. *If* $\vdash a = b$; *then for all* n, m, *we have* $\vdash a : n \to m$ *iff* $\vdash b : n \to m$.

Theorem 2.5. *If* $\vdash a = b$ *and* $\vdash a, b : n \to m$, *then* $\vdash a = b : n \to m$.

Proof. We reason by induction on the derivation $\vdash a = b$; the interesting cases are the following ones:

- the last rule used is the transitivity rule (T): we have $\vdash a = b$, $\vdash b = c$, $\vdash a, c : n \to m$, and we need to show that $\vdash a = c : n \to m$. By Lemma 2.4, we have $\vdash b : n \to m$, so that by the induction hypotheses, we get $\vdash a = b : n \to m$ and $\vdash b = c : n \to m$, and we can apply rule (T).
- the last rule used is the compatibility of $\cdot$ (D): we have $\vdash a = a'$, $\vdash b = b'$, $\vdash a \cdot b, a' \cdot b' : n \to m$, and we need to show that $\vdash a \cdot b = a' \cdot b' : n \to m$. By case analysis on the typing judgements, we deduce that $\vdash a : n \to p$, $\vdash b : p \to m$, $\vdash a' : n \to q$, $\vdash b' : q \to m$, for some p, q. Thanks to Lemmas 2.3 and 2.4, we have $p = q$, so that we can conclude using the induction hypotheses ($\vdash a = a' : n \to p$ and $\vdash b = b' : p \to m$), and rule (D). □

Note that the converse of Theorem 2.5 ($\vdash a = b : n \to m$ entails $\vdash a = b$) is straightforward, so that we actually have an equivalence.

2.2. **Non-commutative semirings.**

Definition 2.6. *Typed semirings* are defined by the signature $\{\cdot_2, +_2, 1_0, 0_0\}$, together with the following rules, in addition to the rules from Def. 2.1 and §1.

$$\frac{}{\vdash 0 : n \to m}\ \textsc{Tz} \qquad \frac{\vdash a, b : n \to m}{\vdash a + b : n \to m}\ \textsc{Tp} \qquad \frac{\vdash a = a' : n \to m \qquad \vdash b = b' : n \to m}{\vdash a + b = a' + b' : n \to m}\ \textsc{p}$$

$$\frac{}{\vdash 0 = 0 : n \to m}\ \textsc{z} \qquad \frac{\vdash a : n \to m}{\vdash a + 0 = a : n \to m}\ \textsc{pz} \qquad \frac{\vdash a, b : n \to m}{\vdash a + b = b + a : n \to m}\ \textsc{pc}$$

$$\frac{\vdash a, b, c : n \to m}{\vdash (a + b) + c = a + (b + c) : n \to m}\ \textsc{pa} \qquad \frac{\vdash a : n \to m \qquad \vdash b, c : m \to p}{\vdash a \cdot (b + c) = a \cdot b + a \cdot c : n \to p}\ \textsc{dp}$$

$$\frac{\vdash a : n \to m}{\vdash a \cdot 0 = 0 : n \to p}\ \textsc{dz} \qquad \frac{\vdash a : n \to m}{\vdash 0 \cdot a = 0 : p \to m}\ \textsc{zd} \qquad \frac{\vdash a : n \to m \qquad \vdash b, c : p \to n}{\vdash (b + c) \cdot a = b \cdot a + c \cdot a : p \to m}\ \textsc{pd}$$

In other words, typed semiring are categories enriched over a commutative monoid: each homset is equipped with a commutative monoid structure (typing rules (Tz,Tp) and rules (p,pz,pc,pa)), composition distributes over these monoid structures (rules (dp,dz,pd,zd)).

Lemma 2.2 is also valid in this setting: equality is reflexive and relates correctly typed terms only. However, due to the presence of the annihilator element (0), Lemmas 2.3 and 2.4 no longer hold: 0 has any type, and we have $\vdash x \cdot 0 \cdot x = 0$ while $x \cdot 0 \cdot x$ only admits $\Gamma(x)$ as a valid type. Moreover, some valid proofs cannot be typed just by adding decorations: for example, $0 = 0 \cdot a \cdot a = 0$ is a valid untyped proof of $0 = 0$; however, this proof cannot be typed if a has a non-square type. Therefore, we have to adopt another strategy: we reduce the problem to the annihilator-free case, by showing that equality proofs can be factorised so as to use rules (pz), (dz), and (zd) at first, as oriented rewriting rules.

Definition 2.7. Let a be a term; we denote by $a_\downarrow$ the *normal form of* a, obtained with the following convergent rewriting system:

$$a + 0 \to a \qquad 0 + a \to a \qquad 0 \cdot a \to 0 \qquad a \cdot 0 \to 0$$

We say that a is *strict* if $a_\downarrow \neq 0$.

This normalisation procedure preserves types and equality; moreover, on strict terms, we recover the injectivity property of types we had for monoids:

Lemma 2.8. *If* $\vdash a : n \to m$*, then* $\vdash a_\downarrow : n \to m$ *and* $\vdash a = a_\downarrow : n \to m$*.*

Lemma 2.9. *For all strict terms* a *such that* $\vdash a : n \to m$ *and* $\vdash a : n' \to m'$*, we have* $n = n'$ *iff* $m = m'$*.*

We can then define a notion of strict equality judgement, where the annihilation laws are not allowed:

Definition 2.10. We let $\_ \vdash^+ \_ = \_ : \_ \to \_$ denote the *strict equality* judgement obtained by removing rules (dz) and (zd), and replacing rules (dp) and (pd) with the

following variants, where the factor has to be strict.

$$\frac{\vdash a : n \to m \qquad \vdash b, c : m \to p \qquad a_\downarrow \neq 0}{\vdash^+ a \cdot (b + c) = a \cdot b + a \cdot c : n \to p}\ \text{DP}^+$$

$$\frac{\vdash a : n \to m \qquad \vdash b, c : p \to n \qquad a_\downarrow \neq 0}{\vdash^+ (b + c) \cdot a = b \cdot a + c \cdot a : p \to m}\ \text{PD}^+$$

Using the same methodology as previously, one easily obtain the untyping theorem for strict equality judgements.

Lemma 2.11. *If $\vdash^+ a = b$; then for all n, m, we have $\vdash a : n \to m$ iff $\vdash b : n \to m$.*

Proposition 2.12. *If $\vdash^+ a = b$ and $\vdash a, b : n \to m$, then $\vdash^+ a = b : n \to m$.*

Note that the patched rules for distributivity, (DP$^+$) and (PD$^+$) are required in order to obtain Lemma 2.11: if a was not required to be strict, we would have $\vdash^+ 0 \cdot (x + y) = 0 \cdot x + 0 \cdot y$, and the right-hand side can be typed in environment $\Gamma = \{x \mapsto (3, 2),\ y \mapsto (4, 2)\}$ while the left-hand side cannot.

We now have to show that any equality proof can be factorised, so as to obtain a strict equality proof relating the corresponding normal forms:

Proposition 2.13. *If $\vdash a = b$, then we have $\vdash^+ a_\downarrow = b_\downarrow$.*

Proof. We first show by induction that whenever $\vdash a = b$, a is strict iff b is strict (†). Then we proceed by induction on the derivation $\vdash a = b$, we detail only some cases:

- (D) we have $\vdash^+ a_\downarrow = a'_\downarrow$ and $\vdash^+ b_\downarrow = b'_\downarrow$ by induction; we need to show that $\vdash^+ (a \cdot b)_\downarrow = (a' \cdot b')_\downarrow$. If one of a, a', b, b' is not strict, then $(a \cdot b)_\downarrow = (a' \cdot b')_\downarrow = 0$, thanks to (†), so that we are done; otherwise, $(a \cdot b)_\downarrow = a_\downarrow \cdot b_\downarrow$, and $(a' \cdot b')_\downarrow = a'_\downarrow \cdot b'_\downarrow$, so that we can apply rule (D).
- (DZ) trivial, since $(a \cdot 0)_\downarrow = 0$.
- (DP) we need to show that $\vdash^+ (a \cdot (b + c))_\downarrow = (a \cdot b + a \cdot c)_\downarrow$; if one of a, b, c is not strict, both sides reduce to the same term, so that we can apply Lemma 2.2(i) (which holds in this setting); otherwise we have $(a \cdot (b + c))_\downarrow = a_\downarrow \cdot (b_\downarrow + c_\downarrow)$ and $(a \cdot b + a \cdot c)_\downarrow = a_\downarrow \cdot b_\downarrow + a_\downarrow \cdot c_\downarrow$, so that we can apply rule (DP$^+$). □

We finally obtain the untyping theorem by putting all together:

Theorem 2.14. *In semirings, for all a, b, n, m such that $\vdash a, b : n \to m$, we have $\vdash a = b$ iff $\vdash a = b : n \to m$.*

Proof. The reverse implication is straightforward; we prove the direct one. By Lemma 2.8, using the transitivity and symmetry rules, it suffices to show $\vdash a_\downarrow = b_\downarrow : n \to m$. This is clearly the case whenever $\vdash^+ a_\downarrow = b_\downarrow : n \to m$, which follows from Props. 2.13 and 2.12. □

2.3. **Kleene algebras.**

Kleene algebras are idempotent semirings equipped with a star operation [21]; they admit several important models, among which binary relations and *regular languages* (the latter is complete [25, 22]; since equality of regular languages is decidable, so is the equational theory of Kleene algebras). Like previously, we type Kleene algebras in a natural way, where star operates on "square" types: types of the form $n \to n$, i.e., square matrices or homogeneous binary relations.

Definition 2.15. We define *typed Kleene algebras* by the signature $\{\cdot_2, +_2, \star_1, 1_0, 0_0\}$, together with the following rules, in addition that from Defs. 2.1 and 2.6, and §1, and where $\vdash a \leq b : n \to m$ is an abbreviation for $\vdash a + b = b : n \to m$.

$$\frac{\vdash a : n \to n}{\vdash a^\star : n \to n}\ \textsc{Ts} \qquad \frac{\vdash a = b : n \to n}{\vdash a^\star = b^\star : n \to n}\ \textsc{S} \qquad \frac{\vdash a : n \to m}{\vdash a + a = a : n \to m}\ \textsc{PI}$$

$$\frac{\vdash a : n \to n}{\vdash 1 + a \cdot a^\star = a^\star : n \to n}\ \textsc{SP} \qquad \frac{\vdash a \cdot b \leq b : n \to m}{\vdash a^\star \cdot b \leq b : n \to m}\ \textsc{SL} \qquad \frac{\vdash b \cdot a \leq b : n \to m}{\vdash b \cdot a^\star \leq b : n \to m}\ \textsc{SR}$$

The untyped version of this axiomatisation is that from Kozen [22]: axiom (PI) corresponds to idempotence of $+$, the three other rules define the star operation (we omitted the mirror image of axiom (SP), which is derivable from the other ones [5]). Note that due to rules (SL) and (SR), we are no longer in a purely equational setting; indeed, the algebra of regular events is not finitely based [36].

The proof of the untyping theorem for Kleene algebras is obtained along the lines of the proof for non-commutative semirings. We just highlight the main differences here, complete proofs are available as Coq scripts [33]. First, it is a simple exercise to check that the following lemma holds:

Lemma 2.16. *For all n, we have $\vdash 0^\star = 1 : n \to n$.*

This allows us to extend the rewriting system from Def. 2.7 with the rule $0^\star \to 1$, so that the annihilator can also be removed in this setting. In particular, we obtain:

Lemma 2.17. *If $\vdash a : n \to m$, then $\vdash a_\downarrow : n \to m$ and $\vdash a = a_\downarrow : n \to m$.*

Lemma 2.18. *For all strict terms a such that $\vdash a : n \to m$ and $\vdash a : n' \to m'$, we have $n = n'$ iff $m = m'$.*

Let $\_ \vdash^+ \_ = \_ : \_ \to \_$ denote the *strict equality* judgement obtained like previously (Def. 2.10), and where we moreover adapt rules (SL) and (SR) so that b is required to be strict:

$$\frac{\vdash^+ a \cdot b \leq b : n \to m \qquad b_\downarrow \neq 0}{\vdash^+ a^\star \cdot b \leq b : n \to m}\ \textsc{SL}^+ \qquad \frac{\vdash^+ b \cdot a \leq b : n \to m \qquad b_\downarrow \neq 0}{\vdash^+ b \cdot a^\star \leq b : n \to m}\ \textsc{SR}^+$$

These patched rules ($\textsc{SL}^+$) and ($\textsc{SR}^+$) are required to obtain the following counterpart to Lemma 2.11: otherwise, we would have $\vdash^+ a^\star \cdot 0 \leq 0$, where the right-hand side has any type while the type of the left-hand side is constrained by a.

Lemma 2.19. *If $\vdash^+ a = b$; then for all n, m, we have $\vdash a : n \to m$ iff $\vdash b : n \to m$.*

Proof. Similar to the proof of Lemma 2.11. Recall that $\vdash^+ a \leq b$ is an abbreviation for $\vdash^+ a + b = b$; the rule (SL$^+$) is handled as follows. Suppose that $\vdash^+ a^\star \cdot b + b = b$ was obtained using this rule:

- if $\vdash a^\star \cdot b + b : n \to m$, then we necessarily have $\vdash b : n \to m$;
- conversely, if $\vdash b : n \to m$ then we have $\vdash a \cdot b + b : n \to m$ by induction. Therefore, there exists p such that $\vdash a : n \to p$ and $\vdash b : p \to m$. Since b was required to be strict, we can use Lemma 2.18 to deduce $n = p$, $\vdash a : n \to n$, and finally, $\vdash a^\star \cdot b + b : n \to m$.

Rule (SR$^+$) is handled symmetrically, and rule (SP) is straightforward. □

The untyping theorem for strict equality follows easily:

Proposition 2.20. *If $\vdash^+ a = b$ and $\vdash a, b : n \to m$, then $\vdash^+ a = b : n \to m$.*

Proof. Like for Theorem 2.5 and Prop. 2.12, we proceed by induction on the untyped derivation to add type annotations. We detail the case of rule (SL$^+$): suppose that $\vdash^+ a^\star \cdot b \leq b$ was obtained using the untyped version of rule (SL$^+$), and $\vdash a^\star \cdot b + b, b : n \to m$. Necessarily, $\vdash a : n \to n$ and $\vdash a \cdot b + b : n \to m$, so that we have $\vdash a \cdot b \leq b : n \to m$ by induction. We conclude using the typed version of rule (SL$^+$): $\vdash a^\star \cdot b \leq b : n \to m$. □

We finally have to prove that Kleene algebra equality proofs can be factorised using the strict equality judgement:

Proposition 2.21. *If $\vdash a = b$, then we have $\vdash^+ a_\downarrow = b_\downarrow$.*

Proof. By induction on the derivation, like for Prop. 2.13. We detail only the rules involving Kleene star:

- (SP): if $a_\downarrow = 0$ then $(1 + a \cdot a^\star)_\downarrow = (a^\star)_\downarrow = 1$ so that we can apply (O); otherwise, $(1 + a \cdot a^\star)_\downarrow = 1 + a_\downarrow \cdot a_\downarrow{}^\star$ and $(a^\star)_\downarrow = a_\downarrow{}^\star$: we can apply (SP).
- (SL): suppose that $\vdash a^\star \cdot b \leq b$ was obtained using this rule, we have to show that $\vdash^+ (a^\star \cdot b)_\downarrow \leq b_\downarrow$. If $b_\downarrow = 0$ then $(a^\star \cdot b)_\downarrow = 0$ and we use rule (Z). Otherwise b is strict, and either $a_\downarrow = 0$, in which case $(a^\star \cdot b)_\downarrow = 1 \cdot b_\downarrow$, and we can use rules (OD) and (PI) to get $\vdash^+ 1 \cdot b_\downarrow \leq b_\downarrow$; or a is also strict. In the latter case, we use the induction hypothesis: $\vdash^+ (a \cdot b)_\downarrow \leq b_\downarrow$, i.e., $\vdash^+ a_\downarrow \cdot b_\downarrow \leq b_\downarrow$, and we conclude using rule (SL$^+$).
- (SR): symmetric to the previous case.

(Note that we implicitly use the fact that normalisation commutes with sum, so that we have $\vdash^+ a_\downarrow \leq b_\downarrow$ iff $\vdash^+ (a + b)_\downarrow = b_\downarrow$.) □

Theorem 2.22. *In Kleene algebras, for all a, b, n, m such that $\vdash a, b : n \to m$, we have $\vdash a = b$ iff $\vdash a = b : n \to m$.*

2.4. **Non-commutative rings.**

Before moving to residuated structures, we briefly discuss the case of non-commutative rings. Indeed, although rings are quite similar to semirings, they cannot be handled in the same way.

Definition 2.23. We define *typed rings* by the signature $\{\cdot_2, +_2, -_1, 1_0, 0_0\}$, together with the following rules, in addition that from Defs. 2.1 and 2.6, and §1.

$$\frac{\vdash a : n \to m}{\vdash -a : n \to m}\ \textsc{Ti} \qquad \frac{\vdash a = b : n \to m}{\vdash -a = -b : n \to m}\ \textsc{i} \qquad \frac{\vdash a : n \to m}{\vdash a + (-a) = 0 : n \to m}\ \textsc{pi}$$

Due to the axiom (PI), we cannot define a simple function to remove annihilators and obtain a factorisation system. Indeed, we have $\vdash a = b$ iff $\vdash a + (-b) = 0$, so that strictness amounts to provability; we no longer have a simple syntactical criterion. However, unlike terms of Kleene algebras, terms of non-commutative rings can easily be put in normal form (by expanding the underlying polynomials and ordering monomials lexicographically—assuming that the set of variables is ordered). This allows us to obtain the untyping theorem by reasoning about the normalisation function.

Let $\langle a \rangle$ denote the normal form of the term a (we do not define formally this standard function here since we are mainly interested in the methodology).

Proposition 2.24. *For all a, b, n, m, we have*

(i) $\vdash a = b$ *iff* $\langle a \rangle = \langle b \rangle$*;*
(ii) if $\vdash a : n \to m$*, then* $\vdash \langle a \rangle : n \to m$*;*
(iii) if $\vdash a : n \to m$*, then* $\vdash \langle a \rangle = a : n \to m$*.*

Proof. *(i)* Standard: this is the correctness and completeness of the untyped decision procedure: two expressions are equal if and only if they share the same normal form.
(ii) By a straightforward induction on the typing derivation.
(iii) Also by induction on the typing derivation, it amounts to replaying the standard correctness proof and checking that it is actually well-typed. □

The untyping theorem follows immediately:

Corollary 2.25. *In non-commutative rings, for all a, b, n, m such that $\vdash a, b : n \to m$, we have $\vdash a = b$ iff $\vdash a = b : n \to m$.*

Proof. If $\vdash a = b$ then $\langle a \rangle = \langle b \rangle$ by the point (i) above, which entails $\vdash \langle a \rangle = \langle b \rangle : n \to m$ by reflexivity since $\vdash \langle a \rangle : n \to m$ by (ii), from which we deduce $\vdash a = b : n \to m$ by (iii). The converse implication is straightforward, as in the previous sections. □

3. Residuated lattices

We now move to our second example, *residuated lattices*. These structures also admit binary relations as models; they are of special interest to reason algebraically about well-founded relations. For example, residuation is used to prove Newman's Lemma in relation algebras [9]. We start with a simpler structure.

A *residuated monoid* is a tuple $(X, \leq, \cdot, 1, \backslash, /)$, such that $(X, \leq)$ is a partial order, $(X, \cdot, 1)$ is a monoid whose product is monotonic ($a \leq a'$ and $b \leq b'$ entail $a \cdot b \leq a' \cdot b'$), and $\backslash, /$ are binary operations, respectively called *left* and *right divisions*, characterised by the following equivalences:

$$a \cdot b \leq c \quad \Leftrightarrow \quad b \leq a \backslash c \quad \Leftrightarrow \quad a \leq c / b$$

$$\frac{}{x \vdash x}\ \textsc{v} \qquad \frac{}{\epsilon \vdash 1}\ \textsc{Io} \qquad \frac{l \vdash a \qquad l' \vdash a'}{l; l' \vdash a \cdot a'}\ \textsc{Id} \qquad \frac{l; b \vdash a}{l \vdash a/b}\ \textsc{Ir} \qquad \frac{b; l \vdash a}{l \vdash b\backslash a}\ \textsc{Il}$$

$$\frac{l; l' \vdash a}{l; 1; l' \vdash a}\ \textsc{Eo} \qquad \frac{l; b; c; l' \vdash a}{l; b \cdot c; l' \vdash a}\ \textsc{Ed} \qquad \frac{k \vdash b \qquad l; c; l' \vdash a}{l; c/b; k; l' \vdash a}\ \textsc{Er} \qquad \frac{k \vdash b \qquad l; c; l' \vdash a}{l; k; b\backslash c; l' \vdash a}\ \textsc{El}$$

Figure 1. Gentzen proof system for residuated monoids.

Such a structure can be typed in a natural way, by using the following rules for left and right divisions:

$$\frac{\vdash c : n \to m \qquad \vdash a : n \to p}{\vdash a\backslash c : p \to m}\ \textsc{Tl} \qquad \frac{\vdash c : n \to m \qquad \vdash b : p \to m}{\vdash c/b : n \to p}\ \textsc{Tr}$$

Although we can easily define a set of axioms to capture equalities provable in residuated monoids [19], the transitivity rule (T) becomes problematic in this setting (there is no counterpart to Lemma 2.4). Instead, we exploit a characterisation due to Ono and Komori [31], based on a Gentzen proof system for the full Lambek calculus [26]. Indeed, the "cut" rule corresponding to this system, which plays the role of the transitivity rule, can be eliminated. Therefore, this characterisation allows us to avoid the problems we encountered with standard equational proof systems. In some sense, moving to cut-free proofs corresponds to using a factorisation system, like we did in the previous section (Prop. 2.13).

3.1. **Gentzen proof system for residuated monoids.**

Let l, k, h range over lists of terms, let $l; k$ denote the concatenation of l and k, and let ϵ be the empty list. The Gentzen proof system is presented on Fig. 1; it relates lists of terms to terms. It is quite standard [19]: there is an axiom rule (V), and, for each operator, an introduction and an elimination rule. The axiom rule can be generalised to terms (*i*), the cut rule is admissible (*ii*), and the proof system is correct and complete w.r.t. residuated monoids (*iii*).

Proposition 3.1.

(i) For all a, we have $a \vdash a$.
(ii) For all l, k, k', a, b such that $l \vdash a$ and $k; a; k' \vdash b$, we have $k; l; k' \vdash b$.
(iii) For all a, b, we have $a \vdash b$ iff $a \leq b$ holds in all residuated monoids.

Proof. Point (*i*) is easy; see [31, 30, 19] for cut admissibility and completeness. □

Type decorations can be added to the proof system in a straightforward way (see Fig. 2). However, using this proof system, we were able to prove the untyping theorem only for the unit-free fragment: we needed to assume that terms have at most one type, which is not true in the presence of 1. This proof was rather involved, so that we did not manage to circumvent this difficulty in a nice and direct way. Instead, as hinted in the introduction, we move to the following more symmetrical setting.

$$\frac{\Gamma(x) = (n, m)}{x \vdash x : n \to m}\ \textsc{v} \qquad \frac{}{\epsilon \vdash 1 : n \to n}\ \textsc{Io} \qquad \frac{l; l' \vdash a : n \to m}{l; 1; l' \vdash a : n \to m}\ \textsc{Eo}$$

$$\frac{l \vdash a : n \to m \qquad l' \vdash a' : m \to p}{l; l' \vdash a \cdot a' : n \to p}\ \textsc{Id} \qquad \frac{l; b; c; l' \vdash a : n \to m}{l; b \cdot c; l' \vdash a : n \to m}\ \textsc{Ed}$$

$$\frac{\vdash b : p \to m \qquad l; b \vdash a : n \to m}{l \vdash a/b : n \to p}\ \textsc{Ir} \qquad \frac{\vdash l' : m \to q \qquad k \vdash b : n \to m \qquad l; c; l' \vdash a : p \to q}{l; c/b; k; l' \vdash a : p \to q}\ \textsc{Er}$$

$$\frac{\vdash b : n \to p \qquad b; l \vdash a : n \to m}{l \vdash b\backslash a : p \to m}\ \textsc{Il} \qquad \frac{\vdash l : p \to m \qquad k \vdash b : m \to n \qquad l; c; l' \vdash a : p \to q}{l; k; b\backslash c; l' \vdash a : p \to q}\ \textsc{El}$$

Figure 2. Typed Gentzen proof system for residuated monoids.

3.2. **Cyclic MLL.**

The sequent proof system for residuated monoids (Fig. 1) actually corresponds to a non-commutative version of intuitionistic multiplicative linear logic (IMLL) [13]: the product $(\cdot)$ is a non-commutative tensor $(\otimes)$, and left and right divisions $(\backslash, /)$ are the corresponding left and right linear implications $(\multimap, \circ\!\!-)$. Moreover, it happens that this system is just the intuitionistic fragment of cyclic multiplicative linear logic (MLL) [40]. The untyping theorem turned out to be easier to prove in this setting, which we describe below.

We assume a copy $\mathcal{X}^\perp$ of the set of variables $(\mathcal{X})$, and we denote by $x^\perp$ the corresponding elements which we call *dual variables*. From now on, we shall consider terms with both kinds of variables: $T(\Sigma + X + X^\perp)$. We keep an algebraic terminology to remain consistent with the previous sections; notice that using terminology from logic, a term is a formula and a variable is an atomic formula.

Definition 3.2. *Typed MLL terms* are defined by the signature $\{\otimes_2, \parr_2, 1_0, \perp_0\}$, together with the following typing rules:

$$\frac{\Gamma(x) = (n, m)}{\vdash x : n \to m}\ \textsc{Tv} \qquad \frac{}{\vdash 1 : n \to n}\ \textsc{T}_1 \qquad \frac{\vdash a : n \to m \qquad \vdash b : m \to p}{\vdash a \otimes b : n \to p}\ \textsc{T}_\otimes$$

$$\frac{\Gamma(x) = (n, m)}{\vdash x^\perp : m \to n}\ \textsc{Tv}^\perp \qquad \frac{}{\vdash \perp : n \to n}\ \textsc{T}_\perp \qquad \frac{\vdash a : n \to m \qquad \vdash b : m \to p}{\vdash a \parr b : n \to p}\ \textsc{T}_\parr$$

Tensor $(\otimes)$ and par $(\parr)$ are typed like the previous dot operation; bottom $(\perp)$ is typed like the unit (1); dual variables are typed by mirroring the types of the corresponding variables. We extend type judgements to lists of terms as follows:

$$\frac{}{\vdash \epsilon : n \to n}\ \textsc{Te} \qquad \frac{\vdash a : n \to m \qquad \vdash l : m \to p}{\vdash a; l : n \to p}\ \textsc{Tc}$$

(be careful not to confuse $\vdash a, b : n \to m$, which indicates that both a and b have type $n \to m$, with $\vdash a; b : n \to m$, which indicates that the list $a; b$ has type $n \to m$). *Linear*

$$\frac{}{\vdash 1 : n}\ 1 \qquad \frac{\vdash l : n}{\vdash \bot; l : n}\ \bot \qquad \frac{\vdash l; a : n \qquad \vdash b; k : n}{\vdash l; a \otimes b; k : n}\ \otimes \qquad \frac{\vdash a; b; l : n}{\vdash a \parr b; l : n}\ \parr$$

$$\frac{\Gamma(x) = (n, m)}{\vdash x^\bot; x : m}\ \mathrm{A} \qquad \frac{\vdash a : n \to m \qquad \vdash l; a : m}{\vdash a; l : n}\ \mathrm{E}$$

Figure 3. Typed proof system for Cyclic MLL.

negation is defined over terms and lists of terms as follows:

$$(x)^\bot \triangleq x^\bot \qquad 1^\bot \triangleq \bot \qquad (a \otimes b)^\bot \triangleq b^\bot \parr a^\bot \qquad (a; l)^\bot \triangleq l^\bot; a^\bot$$
$$(x^\bot)^\bot \triangleq x \qquad \bot^\bot \triangleq 1 \qquad (a \parr b)^\bot \triangleq b^\bot \otimes a^\bot \qquad \epsilon^\bot \triangleq \epsilon$$

Note that since we are in a non-commutative setting, negation has to reverse the arguments of tensors and pars, as well as lists. Negation is involutive and mirrors type judgements:

Lemma 3.3. *For all l, $l^{\bot\bot} = l$; for all l, n, m, $\vdash l : n \to m$ iff $\vdash l^\bot : m \to n$.*

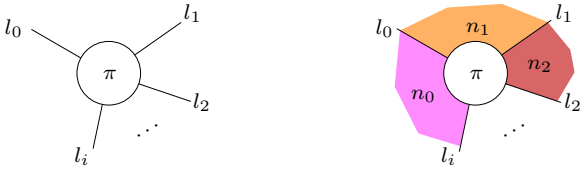
If we were using a two-sided presentation of MLL, judgements would be of the form $l \vdash k : m \to n$, intuitively meaning "$l \vdash k$ is derivable in cyclic MLL, and lists l and k have type $m \to n$". Instead, we work with one-sided sequents to benefit from the symmetrical nature of MLL. At the untyped level, this means that we replace $l \vdash k$ with $\vdash l^\bot; k$. According to the previous intuitions, the list $l^\bot; k$ has a square type $n \to n$: the object m is hidden in the concatenation, so that it suffices to record the outer object (n). Judgements finally take the form $\vdash l : n$, meaning "the one-sided MLL sequent $\vdash l$ is derivable at type $n \to n$".

Definition 3.4. *Typed cyclic MLL* is defined by the sequent calculus from Fig. 3.

Except for type decorations, the system is standard: the five first rules are the logical rules of MLL [13]. Rule (E) is the only structural rule, this is a restricted form of the exchange rule, yielding cyclic permutations: sequents have to be thought of as rings [40]. As before, we added type decorations in a minimal way, so as to ensure that derivable sequents have square types, as explained above:

Lemma 3.5. *For all l, n, if $\vdash l : n$ then $\vdash l : n \to n$.*

We now give a graphical interpretation of the untyping theorem, using proof nets. Since provability is preserved by cyclic permutations, one can draw proof structures by putting the terms of a sequent on a circle [40]. For example, a proof π of a sequent $\vdash l_0, \ldots, l_i$ will be represented by a proof net whose interface is given by the left drawing below.



Suppose now that the corresponding list admits a square type: $\vdash l : n \to n$, i.e., $\forall j \leq i$, $\vdash l_j : n_j \to n_{j+1}$, for some $n_0, \ldots, n_{i+1}$ with $n = n_0 = n_{i+1}$. One can add these type decorations as background colours, in the areas delimited by terms, as we did on the right-hand side.

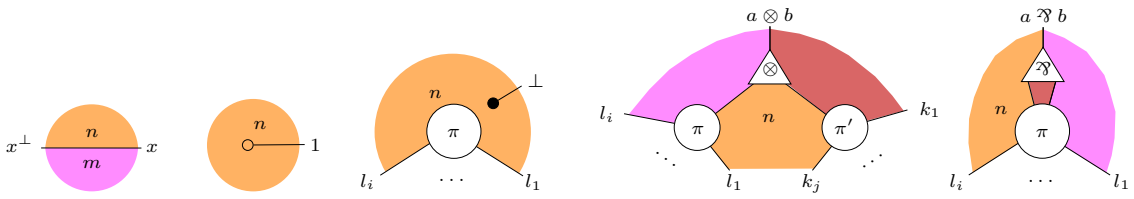
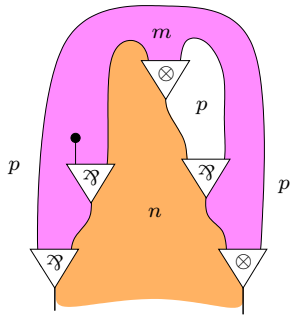


Figure 4. Proof nets for Cyclic MLL.

The logical rules of the proof system (Fig. 3) can then be represented by the proof net constructions from Fig. 4 (thanks to this sequent representation, the exchange rule (E) is implicit). Since these constructions preserve planarity, all proof nets are planar [3], and the idea of background colours makes sense. Moreover, they can be coloured in a consistent way, so that typed derivations correspond to proof nets that can be entirely and consistently coloured. Therefore, one way to prove the untyping theorem consists in showing that any proof net whose outer interface can be coloured can be coloured entirely. As an example, we give an untyped derivation below, together with the corresponding proof net. Assuming that $\Gamma(x) = n \to m$ and $\Gamma(y) = m \to p$, the conclusion has type $p \to p$, and the outer interface of the proof net can be coloured (here, with colours p and n). The untyping theorem will ensure that there exists a typed proof; indeed, the whole proof net can be coloured in a consistent way.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash x^\perp; x}\,\mathrm{A} \quad \dfrac{}{\vdash y; y^\perp}\,\mathrm{E,A}}{\vdash x^\perp; (x \otimes y); y^\perp}\,\otimes}{\vdash x^\perp; (x \otimes y) \parr y^\perp}\,\parr \quad \dfrac{}{\vdash y; y^\perp}\,\mathrm{E,A}}{\vdash x^\perp; ((x \otimes y) \parr y^\perp) \otimes y; y^\perp}\,\otimes}{\vdash \perp; x^\perp; ((x \otimes y) \parr y^\perp) \otimes y; y^\perp}\,\perp}{\vdash y^\perp; \perp; x^\perp; ((x \otimes y) \parr y^\perp) \otimes y}\,\mathrm{E}}{\vdash y^\perp \parr \perp \parr x^\perp; ((x \otimes y) \parr y^\perp) \otimes y}\,\parr$$



We now embark in the proof of the untyping theorem for cyclic MLL; the key property is that the types of derivable sequents are all squares:

Proposition 3.6. *If* $\vdash l$ *and* $\vdash l : n \to m$*, then* $n = m$*.*

Proof. We proceed by induction on the untyped derivation $\vdash l$, but we prove a stronger property: "the potential types of all cyclic permutations of l are squares", i.e., for all h,k such that $l = h; k$, for all n, m such that $\vdash k; h : n \to m$, $n = m$. The most involved case is that of the tensor rule. Using symmetry arguments, we can assume that the cutting point belongs to the left premise: the conclusion of the tensor rule is $\vdash l; l'; a \otimes b; k$, we suppose that the induction hypothesis holds for $l; l'; a$ and $b; k$, and knowing that $\vdash l'; a \otimes b; k; l : n \to m$, we have to show $n = m$. Clearly, we have $\vdash l'; a : n \to p$, $\vdash b; k : p \to q$, and $\vdash l : q \to m$ for some p, q. By induction on the second premise, we have $p = q$, so that $\vdash l'; a; l : n \to m$. Since the latter list is a cyclic permutation of $l; l'; a$, we can conclude with the induction hypothesis on the first premise. □

Theorem 3.7. *In cyclic MLL, if* $\vdash l : n \to n$*, then we have* $\vdash l$ *iff* $\vdash l : n$*.*

Proof. The right-to-left implication is straightforward; for the direct implication, we proceed by induction on the untyped derivation. The previous proposition is required in the case of the tensor rule: we know that $\vdash l; a$, $\vdash b; k$, and $\vdash l; a \otimes b; k : n \to n$, and we have to show that $\vdash l; a \otimes b; k : n$. Necessarily, there is some m such that $\vdash l; a : n \to m$ and $\vdash b; k : m \to n$; moreover, by Prop. 3.6, $n = m$. Therefore, we can apply the induction hypotheses (so that $\vdash l; a : n$ and $\vdash b; k : n$) and we conclude with the typed tensor rule. □

3.3. **Intuitionistic fragment.**

To deduce that the untyping theorem holds in residuated monoids, it suffices to show that the typed version of the proof system from §3.1 corresponds to the intuitionistic fragment of the proof system from Fig. 3. This is well-known for the untyped case, and type decorations do not add particular difficulties. Therefore, we just give a brief overview of the extended proof.

The idea is to define the following families of *input* and *output* terms (Danos-Regnier polarities [37, 4]), and to work with sequents composed of exactly one output term and an arbitrary number of input terms.

$$\begin{aligned} i &::= x^\perp \mid \bot \mid i \parr i \mid i \otimes o \mid o \otimes i \\ o &::= x \mid 1 \mid o \otimes o \mid i \parr o \mid o \parr i \end{aligned}$$

Negation $(-^\perp)$ establishes a bijection between input and output terms. Terms of residuated monoids (IMLL formulae) are encoded into output terms as follows.

$$\lfloor a \cdot b \rfloor \triangleq \lfloor a \rfloor \otimes \lfloor b \rfloor \qquad \lfloor a/b \rfloor \triangleq \lfloor a \rfloor \parr \lfloor b \rfloor^\perp \qquad \lfloor x \rfloor \triangleq x$$
$$\lfloor 1 \rfloor \triangleq 1 \qquad \lfloor a \backslash b \rfloor \triangleq \lfloor a \rfloor^\perp \parr \lfloor b \rfloor$$

This encoding is a bijection between IMLL terms and MLL output terms; it preserves typing judgements:

Lemma 3.8. *For all* a, n, m*, we have* $\vdash a : n \to m$ *iff* $\vdash \lfloor a \rfloor : n \to m$*.*

(Note that we heavily rely on overloading to keep notation simple.) The next proposition shows that we actually obtained a fragment of typed cyclic MLL; it requires the lemma below: input-only lists are not derivable. The untyping theorem for residuated monoids follows using Thm. 3.7.

Lemma 3.9. *If* $\vdash l$*, then* l *contains at least one output term.*

Proposition 3.10. *If* $\vdash l, a : n \to m$*, then* $l \vdash a : n \to m$ *iff* $\vdash \lfloor l \rfloor^\perp; \lfloor a \rfloor : m$*.*

Proof. The forward implication is proved by an induction on the sequent derivation. For the reverse direction, we actually prove the following stronger property, by induction on the untyped MLL derivation:

> "for all h, a, k, n, m such that we have $\vdash \lfloor h \rfloor^\perp; \lfloor a \rfloor; \lfloor k \rfloor^\perp$, $\vdash h; k : n \to m$, and $\vdash a : n \to m$, we have $h; k \vdash a : n \to m$".

This generalisation is required to handle the exchange rule. We detail only the key cases:

- If the tensor rule was used last, on the output term (which was thus of the form $\lfloor a \cdot b \rfloor = \lfloor a \rfloor \otimes \lfloor b \rfloor$):
$$\frac{\vdash \lfloor h \rfloor^\perp; \lfloor a \rfloor \qquad \vdash \lfloor b \rfloor; \lfloor k \rfloor^\perp}{\vdash \lfloor h \rfloor^\perp; \lfloor a \rfloor \otimes \lfloor b \rfloor; \lfloor k \rfloor^\perp} \otimes$$
Since $\vdash a \cdot b : n \to m$, and $\vdash h; k : n \to m$, we have p, q such that $\vdash a : n \to p$, $\vdash b : p \to m$, $\vdash h : n \to q$ and $\vdash k : q \to m$. Therefore, by Lemmas 3.3 and 3.8, we have $\vdash \lfloor h \rfloor^\perp; \lfloor a \rfloor : q \to p$, whence $p = q$ by Prop. 3.6. We can thus apply the induction hypothesis to the two premises to obtain $h \vdash a : n \to p$ and $k \vdash b : p \to m$ (using an empty sequence in both cases). We conclude using rule (Id) from Fig. 1.
- If the tensor rule was used last, on one of the input terms, say on b in $h = h_1; b; h_2$, with $\lfloor b \rfloor^\perp = c \otimes d$:
$$\frac{\vdash \lfloor h_2 \rfloor^\perp; c \qquad \vdash d; \lfloor h_1 \rfloor^\perp, \lfloor a \rfloor, k^\perp}{\vdash \lfloor h_2 \rfloor^\perp; c \otimes d; \lfloor h_1 \rfloor^\perp; \lfloor a \rfloor; \lfloor k \rfloor^\perp} \otimes$$
Since $\lfloor h \rfloor_2^\perp; c$ is provable and $\lfloor h \rfloor_2^\perp$ contains only input terms, c is necessarily an output term by Lemma 3.9. Therefore there is only one possibility ensuring $\lfloor b \rfloor = d^\perp \parr c^\perp$: the term b must be of the form d'/c', with $\lfloor d' \rfloor = d^\perp$ and $\lfloor c' \rfloor = c$.
We have $\vdash h_1; d'/c'; h_2; k : n \to m$ and $\vdash a : n \to m$, i.e., $\vdash h_1 : n \to p$, $\vdash d' : p \to q$, $\vdash c' : r \to q$, $\vdash h_2 : r \to s$, and $\vdash k : s \to m$ for some p, q, r, s. We first notice that the provable sequent $\lfloor h_2 \rfloor^\perp; c$ has type $s \to q$, so that $s = q$ by Prop. 3.6. By induction, we then deduce $h_2 \vdash c' : r \to q$ and $h_1; d'; k \vdash a : n \to m$, and we conclude using rule (Er) from Fig. 1. □

Corollary 3.11. *In residuated monoids, if* $\vdash l, a : n \to m$*, then we have* $l \vdash a$ *iff* $l \vdash a : n \to m$.

3.4. **Residuated lattices: additives.**

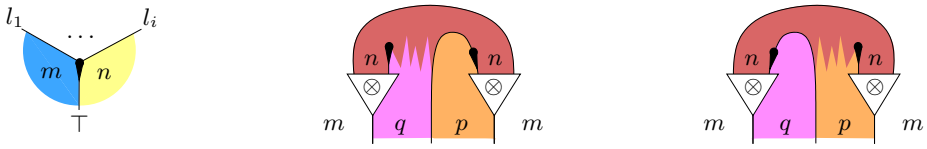
The Gentzen proof system we presented for residuated monoids (Fig. 1) was actually designed for residuated *lattices* [31], obtained by further requiring the partial order $(X, \leq)$ to be a lattice $(X, \vee, \wedge)$. Binary relations fall into this family, by considering set-theoretic unions and intersections. The previous proofs scale without major difficulty: on the logical side, this amounts to considering the additive binary connectives $(\oplus, \&)$. By working in multiplicative additive linear logic (MALL) without additive constants, we get an untyping theorem for *involutive residuated lattices* [39]; we deduce the untyping theorem for residuated lattices by considering the corresponding intuitionistic fragment (see [33] for proofs).

On the contrary, and rather surprisingly, the theorem breaks if we include additive constants $(0, \top)$, or equivalently, if we consider *bounded* residuated lattices. The corresponding typing rules are given below, together with the logical rule for top (there is no rule for zero).
$$\frac{}{\vdash 0 : n \to m}\ \mathrm{T}_0 \qquad \frac{}{\vdash \top : n \to m}\ \mathrm{T}_\top \qquad \frac{\vdash l : m \to n}{\vdash \top; l : n}\ \top$$

The sequent $x^\perp \otimes \top; y^\perp; \top \otimes x$ gives a counter-example. This sequent basically admits the two following untyped proofs:

$$\dfrac{\dfrac{\dfrac{}{\vdash y^\perp; \top}\,\mathrm{E},\top \quad \dfrac{\dfrac{}{\vdash x; x^\perp}\,\mathrm{E},\mathrm{A} \quad \dfrac{}{\vdash \top}\,\top}{\vdash x; x^\perp \otimes \top}\,\otimes}{\vdash y^\perp; \top \otimes x; x^\perp \otimes \top}\,\otimes}{\vdash x^\perp \otimes \top; y^\perp; \top \otimes x}\,\mathrm{E} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \top}\,\top \quad \dfrac{\dfrac{}{\vdash x; x^\perp}\,\mathrm{E},\mathrm{A} \quad \dfrac{}{\vdash \top; y^\perp}\,\top}{\vdash x; x^\perp \otimes \top; y^\perp}\,\otimes}{\vdash \top \otimes x; x^\perp \otimes \top; y^\perp}\,\otimes}{\vdash x^\perp \otimes \top; y^\perp; \top \otimes x}\,\mathrm{E},\mathrm{E}$$

However, this sequent admits the square type $m \to m$ whenever $\Gamma(x) = (n, m)$ and $\Gamma(y) = (p, q)$, while the above proofs cannot be typed unless $n = q$ or $n = p$, respectively. Graphically, these proofs correspond to the proof nets below (where the proof net construction for rule $(\top)$ is depicted on the left-hand side); these proof nets cannot be coloured unless $n = q$ or $n = p$.



This counter-example for MALL also gives a counter-example for IMALL: the above proofs translate to intuitionistic proofs of $y \cdot (\top \backslash x) \vdash \top \cdot x$, which is also not derivable in the typed setting, unless $n = q$ or $n = p$.

The problem is actually even stronger: while $S \cdot (\top \backslash R) \subseteq \top \cdot R$ holds for all homogeneous binary relations R, S (by the above untyped proofs, for example), this law does not hold for arbitrary heterogeneous relations (see Remark 3.12 below). This shows that we cannot always reduce the analysis of typed structures to that of the underlying untyped structures. Here, the equational theory of heterogeneous binary relations does not reduce to the equational theory of homogeneous binary relations.

Remark 3.12. The containment $S \cdot (\top \backslash R) \subseteq \top \cdot R$ does not necessarily hold for all heterogeneous binary relations R, S, although it holds for all heterogeneous binary relations on non-empty sets.

Proof. Let A, B, C, D be four sets, let $R \subseteq B \times C$ be a binary relation from B to C, and let $S \subseteq D \times A$ be a binary relation from D to A. To be precise, we denote by $\top_{X,Y}$ the full relation between sets X and Y ($X \times Y$), and the containment from the statement can be rewritten as

$$S \cdot (\top_{B,A} \backslash R) \subseteq \top_{D,B} \cdot R \ .$$

For all relations $T \subseteq B \times A$, the relation $T \backslash R$ is characterised as follows:

$$T \backslash R = \{(i, j) \in A \times C \mid \forall k \in B, (k, i) \in T \to (k, j) \in R\} \ .$$

- if B is the empty set, then $R = \top_{D,B} \cdot R = \emptyset$, and by the above characterisation, we have $\top_{B,A} \backslash R = A \times C$. Therefore, we can contradict the containment by taking any non-empty relation for S. (Note that this cannot happen in an homogeneous setting: we have $A = B = C = D$ so that taking the empty set for B forces both R and S to be empty.)

- if B is not empty, then we have

$$\begin{aligned}\top_{B,A}\backslash R &= \{(i,j) \in A \times C \mid \forall k \in B, (k,i) \in \top_{B,A} \to (k,j) \in R\}\\ &= \{(i,j) \in A \times C \mid \forall k \in B, (k,j) \in R\}\\ &\subseteq \{(i,j) \in A \times C \mid \exists k \in B, (k,j) \in R\}\\ &= \{(i,j) \in A \times C \mid \exists k \in B, (i,k) \in \top_{A,B} \wedge (k,j) \in R\}\\ &= \top_{A,B} \cdot R \ ;\end{aligned}$$

  Therefore, since $S \subseteq \top_{D,A}$, we can conclude:

$$S \cdot (\top_{B,A}\backslash R) \subseteq \top_{D,A} \cdot \top_{A,B} \cdot R \subseteq \top_{D,B} \cdot R \ .$$

□

We do not know whether relations on empty sets are required to get such a counter-example in the model of binary relations. In other words, for the signature of bounded residuated lattices, does the equational theory of heterogeneous binary relations on non-empty sets reduce to the equational theory of homogeneous binary relations?

4. Improving proof search for residuated structures.

The sequent proof systems we mentioned in the previous section have the sub-formula property, so that provability is decidable in each case, using a simple proof search algorithm [30]. Surprisingly, the concept of type can be used to cut off useless branches. Indeed, recall Prop. 3.6: "the types of any derivable sequent are squares". By contrapositive, given an untyped sequent l, one can easily compute an abstract 'most general type and environment' $(n \to m, \Gamma)$, such that $\Gamma \vdash l : n \to m$ holds (taking $\mathbb{N}$ as the set of objects, for example); if $n \neq m$, then the sequent is not derivable, and proof search can fail immediately on this sequent.

We did some experiments with a simple prototype [33]: we implemented focused [2] proof search for cyclic MALL, i.e., a recursive algorithm composed of an *asynchronous* phase which is deterministic and a *synchronous* phase, where branching occurs (e.g., when applying the tensor rule ($\otimes$)). The optimisation consists in checking that the most general type of the sequent is square before entering the synchronous phases. The overall complexity remains exponential (provability is NP-complete [32]—PSPACE-complete with additives [20]) but we get an exponential speed-up: we can abort proof search immediately on approximately two sequents out of three.

The experimental results are given on Fig. 5 and 6—raw data is available from [33]. We generate (pseudo) random sequents in normal form with respect to the laws of neutral elements for multiplicative constants (1 and $\bot$), with a given number of leaves (variables, dual variables or constants), and where variables are picked in a set of the specified size. E.g., $a \otimes \bot; b^\bot$ is a sequent with three leaves and two variables, which can also be considered as a sequent with three leaves and four variables, where two variables are not used.

Each point of Fig. 5 was obtained by timing focused proof search with and without optimisation, on a set of 100 000 sequents with the given characteristics: fixed number of variables and varying size on the left-hand side, fixed size and varying number of variables on the right-hand side. While the optimisation introduces a small amount of overhead for very small sequents or sequents with few variables, we gain more than one order of magnitude for larger sequents. One can also notice that the more variables are available, the more

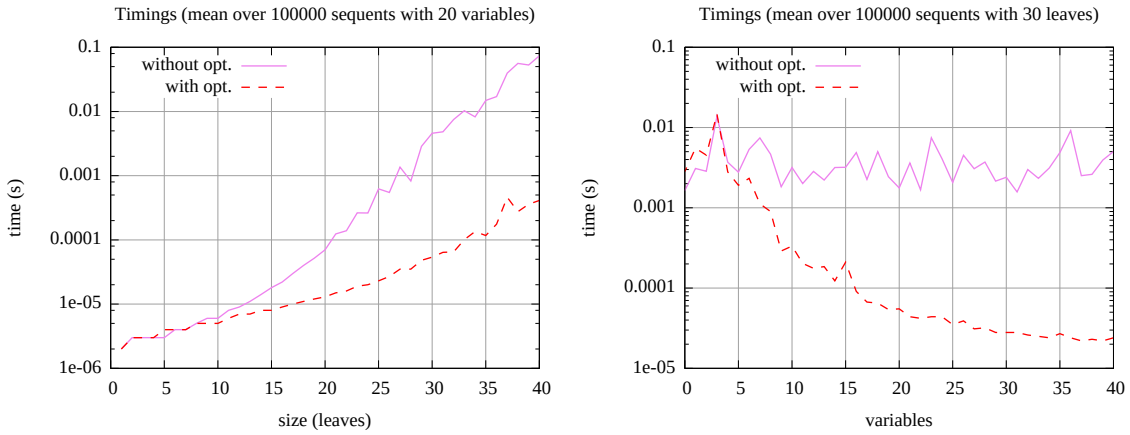


Figure 5. Searching times for focused proof search with and without optimisation.

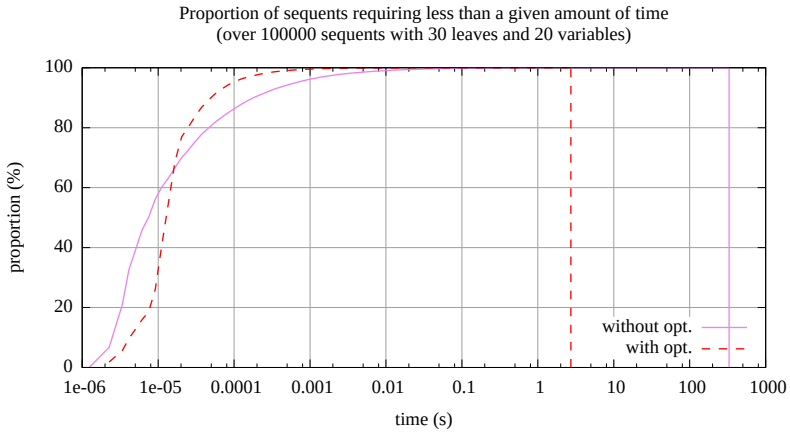


Figure 6. Distribution of searching times.

efficient the optimisation is: indeed, sequents with a lot of different variables tend to have non-square types more easily, so that they can be ruled out more frequently.

We did not report standard deviation in Fig. 5 since it does not make sense in this setting: we have an unbounded set of potential values, and the actual complexity of proof search is highly stochastic. Instead, we computed the distribution of searching times: Fig. 6 shows the proportion of sequents that are solved in a given amount of time, among sequents with a fixed size and number of variables—here, 30 leaves and 20 variables. While 60% of the sequents are solved in less that 10^{-5}s (with or without optimisation), some of them require much more time: up to five minutes without optimisation, and up to three seconds with the optimisation. All in all, the overhead which is paid on 'easily solved' sequents gets compensated by the drastic improvement on 'harder' sequents.

5. Conclusions and directions for future work

We proved untyping theorems for several standard structures, allowing us to extend decidability results to the typed settings, and to discover an optimisation of proof search for cyclic linear logic. All results have been formally checked [33] with the Coq proof assistant.

The untyping theorem for typed Kleene algebras is quite important in the ATBR Coq library [5]: it allows one to use our tactic for Kleene algebras in typed settings, and, in particular, with heterogeneous binary relations. The underlying decision procedure being quite involved, we can hardly imagine proving its soundness with respect to typed settings in a direct way. Even writing a type-preserving version of the algorithm seems challenging.

At another level, we used the untyping theorem for semirings in order to formalise Kozen's completeness proof [22] for Kleene algebras. Indeed, this proof heavily relies on matrix constructions, so that having adequate lemmas and tactics for working with possibly rectangular matrices was a big plus: this allowed us to avoid the ad-hoc constructions Kozen used to inject rectangular matrices into square ones.

5.1. **References and related work.**

The relationship between residuated lattices and substructural logics is due to Ono and Komori [31]; see [11] for a thorough introduction. Cyclic linear logic was suggested by Girard and studied by Yetter [40]. To the best of our knowledge, the idea of adding types to the above structures is new. The axiomatisation of Kleene algebras is due to Kozen [22].

Our typed structures can be seen as very special cases of *partial* algebras [6], where the domain of partial operations is defined by typing judgements. Similarly, one could use *many-sorted* algebras [16] to mimic types using sorts. Several encodings from partial algebras to total ones were proposed in the literature [28, 7]. Although they are quite general, these results do not apply here: these encodings do not preserve the considered theory since they need to introduce new symbols and equations; as a consequence, ordinary untyped decision procedures can no longer be used after the translation. Dojer has shown that under some conditions, convergent term rewriting systems for total algebras can be used to prove existence equations in partial algebras [8]. While it seems applicable to semirings, this approach does not scale to Kleene algebras or residuated lattices, for which decidability does not arise from a term rewriting system.

The idea of proving typed equations from untyped ones also appears in the context of "Pure Type Systems" (PTSs), where one can use either an untyped conversion rule, or a typed equality judgement. Whether these two possible presentations were equivalent was open for some time [12]; Adams has shown that this is the case for "functional" PTSs [1], Herbelin and Siles recently generalised the result to all PTSs [38]. Although the types we use here are quite basic (i.e., a type is just a pair of abstract objects), our use of cut-free proof systems and factorisation systems is reminiscent to their use of the Church-Rosser property. Note however that unlike in functional programming languages, where one usually relies on a Hindley-Milner type inference algorithm [17, 27] to rule out ill-typed programs, no inference algorithm is required with the algebraic theories presented here: such an algorithm would always succeed since an untyped proof systematically yields a typed proof.

Closer to our work is that of Kozen, who first proposed the idea of untyping typed Kleene algebras, in order to avoid the aforementioned matrix constructions [24]. He provided a different answer, however: using model-theoretic arguments, he proved an untyping theorem for the universal theory of "1-free Kleene algebras". The restriction to 1-free expressions is

required, as shown by the following counter-example: $\vdash 0 = 1 \Rightarrow a = b$ is a theorem of semirings, although there are non trivial typed semirings where $0 = 1$ holds at some types (e.g., empty matrices), while $a = b$ is not universally true at other types.

5.2. **Handling other structures.**

Action algebras [35, 18] are a natural extension of the structures we studied in this paper: they combine the ingredients from residuated lattices and Kleene algebras. In this setting, left and right divisions make it possible to obtain a variety rather than a quasi-variety: inference rules (SL) and (SR), about the star operation, can be replaced by the following equational axioms:

$$\frac{}{\vdash (a\backslash a)^\star = a\backslash a}\ \text{SL'} \qquad\qquad \frac{}{\vdash (a/a)^\star = a/a}\ \text{SR'}$$

Although we do not know whether the untyping theorem holds in this case, we can think of two strategies to tackle this problem: 1) find a cut-free extension of the Gentzen proof system for residuated lattices and adapt our current proof—such an extension is left as an open question in [18], it would possibly entail decidability of the equational theory of action algebras; 2) find a "direct" proof of the untyping theorem for residuated monoids, without using a Gentzen proof system, so that the methodology we used for Kleene algebras can be extended. Also note that we necessarily have to exclude the annihilator element (0): with divisions, top ($\top$) can be defined as $0/0$, so that the counter-example for bounded residuated lattices (§3.4) applies. Consistently, there is no way to remove this element using a factorisation system: expressions like $a \cdot \top$ cannot be simplified.

Kleene algebras with tests [23] are another extension of Kleene algebras, which is useful in program verification. Their equational theory is decidable, but one cannot rely on a factorisation system to remove annihilators in this setting: like for rings (§2.4), the complement operation of the Boolean algebra is problematic. Moreover, like for Kleene algebras, there are no known notions of normal form in Kleene algebras with tests, so that the approach we described in §2.4 is not possible. Nonetheless, the untyping theorem is likely to hold for these structures since the Boolean algebras of tests are inherently homogeneous.

Finally, although our methodology for semirings can be adapted to handle the cases of *allegories* and *distributive allegories* [10] (see [33] for proofs), the case of *division allegories*—where left and right divisions are added—remains open.

5.3. **Towards a generic theory.**

The typed structures we focused on can be described in terms of enriched categories, and the untyping theorems can be rephrased as asserting the existence of faithful functors to one-object categories. It would therefore be interesting to find out whether category theory may help to define a reasonable class of structures for which the untyping theorem holds. In particular, how could we exclude the counter-example with additive constants in MALL?

For structures that are varieties, another approach would consist of using term rewriting theory to obtain generic factorisation theorems (Lemma 2.13, which we used to handle the annihilating element in semirings, would become a particular case). This seems rather difficult, however, since these kind of properties are quite sensitive to the whole set of operations and axioms that are considered.

Acknowledgements.

We are grateful to Olivier Laurent and Tom Hirschowitz for the highly stimulating discussions we had on linear logic and about this work.

References

[1] R. Adams. Pure type systems with judgemental equality. *J. Funct. Program.*, 16(2):219–246, 2006.
[2] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.
[3] G. Bellin and A. Fleury. Planar and braided proof-nets for MLL with mix. *Archive for Mathematical Logic*, 37:309–325, 1998.
[4] G. Bellin and P. Scott. On the π-calculus and linear logic. *TCS*, 135:11–65, 1994.
[5] T. Braibant and D. Pous. An efficient Coq tactic for deciding Kleene algebras. In *Proc. ITP*, volume 6172 of *LNCS*, pages 163–178. Springer, 2010.
[6] P. Burmeister. Partial algebra — an introductory survey. In *Algebras and Orders*, volume 389 of *NATO ASI*, pages 1–70. Kluwer Pub., 1993.
[7] R. Diaconescu. An encoding of partial algebras as total algebras. *Inf. Process. Lett.*, 109(23-24):1245–1251, 2009.
[8] N. Dojer. Applying term rewriting to partial algebra theory. *Fund. Inf.*, 63(4):375–384, 2004.
[9] H. Doornbos, R. Backhouse, and J. van der Woude. A calculational approach to mathematical induction. *TCS*, 179(1-2):103–135, 1997.
[10] P. Freyd and A. Scedrov. *Categories, Allegories*. North Holland, 1990.
[11] N. Galatos, P. Jipsen, T. Kowalski, and H. Ono. Residuated lattices: an algebraic glimpse at substructural logics. *Stud. in Log. and Found. of Math.*, 151:532, 2007.
[12] H. Geuvers and B. Werner. On the Church-Rosser property for expressive type systems and its consequences for their metatheoretic study. In *Proc. LICS*, pages 320–329. IEEE, 1994.
[13] J.-Y. Girard. Linear logic. *TCS*, 50:1–102, 1987.
[14] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Proc. TPHOLs*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.
[15] J. Harrison. A HOL decision procedure for elementary real algebra. In *HUG*, volume 780 of *LNCS*, pages 426–435. Springer, 1993.
[16] P. Higgins. Algebras with a scheme of operators. *Math. Nach.*, 27:115–132, 1963.
[17] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
[18] P. Jipsen. From semirings to residuated Kleene lattices. *Stud. Log.*, 76(2):291–303, 2004.
[19] P. Jipsen and C. Tsinakis. A survey of residuated lattices. *Ord. Alg. Struct.*, 2002.
[20] M. Kanovich. The complexity of neutrals in linear logic. In *Proc. LICS*, pages 486–495. IEEE, 1995.
[21] S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–41. Princeton University Press, 1956.
[22] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. and Comput.*, 110(2):366–390, 1994.
[23] D. Kozen. Kleene algebra with tests. *Trans. PLS*, 19(3):427–443, 1997.
[24] D. Kozen. Typed Kleene algebra. Technical Report 98-1669, Cornell Univ., 1998.
[25] D. Krob. Complete systems of B-rational identities. *TCS*, 89(2):207–343, 1991.
[26] J. Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958.
[27] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
[28] T. Mossakowski. Relating CASL with other specification languages: the institution level. *TCS*, 286(2):367–475, 2002.
[29] M. Norrish. Complete integer decision procedures as derived rules in HOL. In *Proc. TPHOLs*, volume 2758 of *LNCS*, pages 71–86. Springer, 2003.
[30] M. Okada and K. Terui. The finite model property for various fragments of intuitionistic linear logic. *J. Sym. Log.*, 64(2):790–802, 1999.
[31] H. Ono and Y. Komori. Logics without the contraction rule. *J. Sym. Log.*, 50(1):169–201, 1985.
[32] M. Pentus. Lambek calculus is NP-complete. *TCS*, 357(1-3):186–201, 2006.

[33] D. Pous. Web appendix for [34] and this paper, http://perso.ens-lyon.fr/damien.pous/utas.
[34] D. Pous. Untyping typed algebraic structures and colouring proof nets of cyclic linear logic. In *Proc. CSL*, volume 6247 of *LNCS*, pages 484–498. Springer, August 2010.
[35] V. R. Pratt. Action logic and pure induction. In *Proc. JELIA*, volume 478 of *LNCS*, pages 97–120. Springer, 1990.
[36] V. Redko. On defining relations for the algebra of regular events (Russian). *Ukrain. Mat. Z.*, 16:120–126, 1964.
[37] L. Regnier. *Lambda-calcul et réseaux*. Thèse de doctorat, Univ. Paris VII, 1992.
[38] V. Siles and H. Herbelin. Equality is typable in semi-full pure type systems. In *Proc. LICS*, pages 21–30. IEEE, 2010.
[39] A. Wille. A Gentzen system for involutive residuated lattices. *Alg. Univ.*, 54:449–463, 2005.
[40] D. Yetter. Quantales and (noncommutative) linear logic. *J. Sym. Log.*, 55(1):41–64, 1990.

Revisiting glue expressiveness in component-based systems*

Cinzia Di Giusto and Jean-Bernard Stefani

INRIA Rhône Alpes, Grenoble, France

Abstract. We take a fresh look at the expressivity of BIP, a recent influential formal component model developed by J. Sifakis et al. We introduce a process calculus, called CAB, that models composite components as the combination of a *glue* (using BIP terminology) and subcomponents, and that constitutes a conservative extension of BIP with more dynamic forms of glues. We study the Turing completeness of CAB variants that differ only in their language for glues. We show that limiting the glue language to BIP glues suffices to obtain Turing-completeness, whereas removing priorities from the control language loses Turing-completeness. We also show that adding a simple form of dynamic component creation in the control language without priorities is enough to regain Turing completeness. These results complement those obtained on BIP, highlighting in particular the key role of priorities for expressivity.

1 Introduction

Component-based software engineering is by now well entrenched in various areas, from embedded systems to Web applications, and is supported by numerous standards, including UML. Its central tenet is that complex systems can be built by composing, or *gluing* together possibly independently developed components.

In their paper on glue expressiveness [3] Bliudze and Sifakis have proposed to look at the expressive power of *glues* or composition operators in an effort to assess the relative merits of different component frameworks with respect to their composition capabilities. In essence, the criterion they use to compare two sets $\mathcal{G}_1$ and $\mathcal{G}_2$ of composition operators is whether it is possible, given a family of primitive components $\mathcal{B}$ and an equivalence relation $\sim$ between these components, to find, for a given operator $g_1 \in \mathcal{G}_1$, a corresponding operator $g_2 \in \mathcal{G}_2$ such that all their compositions are equivalent, i.e. $\forall B_1, \ldots, B_n \in \mathcal{B}$: $g_1(B_1, \ldots, B_n) \sim g_2(B_1, \ldots, B_n)$. As a notable result, they showed that their BIP component framework, whose glues feature multiparty synchronization and priorities, is universal with respect to a family of operators defined by inference rules in a subset of the GSOS format.

This work, however, leaves open a number of questions, in particular regarding the form glues can take, and their intrinsic expressivity. Indeed, the notion of

* Research partially funded by ANR Project PiCoq, Fondation de Coopération Scientifique Digiteo Triangle de la Physique, and Minalogic Project Mind.

glue in [3] is essentially a static one. One may legitimately argue in favor of more dynamic forms of composition, e.g. to allow the creation of new components or the replacement of existing ones to accommodate different forms of software update. Even without considering full dynamic reconfiguration, one may take into account changes in configuration or interconnection between components, e.g. to accommodate different *modes* of operation, where the notion of mode is loosely understood as a collection of execution states [9]. It thus appears beneficial to consider not just static glues but glue processes in their own right.

In the paper, we adopt this view: we model component assemblages as terms in a process calculus, called CAB (for Components And Behaviors). A component assemblage (or composite component) in CAB takes the form $l[C_1; \ldots; C_n \rhd B]$, where l is the name of the composite, $C_1, \ldots, C_n$ are the subcomponents of the composite, i.e. the components that are glued together (using BIP terminology) in the assemblage, and B is the *glue* – a term in a simple process calculus which we call the *glue language*. By construction, we recover BIP glues as essentially single state processes of our glue language.

With this view of glues as terms of a glue language, new expressivity questions arise, such as:

1. What is the expressivity of the resulting process calculus (in particular, if we restrict the glue language to terms corresponding to BIP glues only)?
2. What is the expressivity of the calculus if we remove the possibility of specifying priority constraints in the glue language ?
3. What is the expressivity of the calculus if we add more dynamic forms of control, such as component creation, in the glue language ?

In this paper we (begin to) answer these questions using classical Turing-completeness as our benchmark for expressivity. Following BIP, the CAB calculus is parametric over a family $\mathcal{P}$ of primitive components. So if we considered a large enough family, these questions would be trivial. Instead, we restrict our primitive components to be given by terms from the glue language itself – which form a strict non-Turing-complete subset of CCS – so as to characterize the intrinsic expressivity of the glue language. The questions then become non-trivial, and we obtain answers that may even appear surprising. Indeed, we first show that even with the restricted glue language consisting of static BIP glues only, the resulting variant of CAB is Turing-complete. Second, we show that this expressivity is lost if one restricts oneself to a subset of the glue language without priority constraints. These results confirms the expressive power of priorities, which was pointed out but not necessarily as clearly apparent in earlier works on BIP and process calculi with priorities. Finally, as a first answer to the last question, we show that we recover Turing-completeness if we add a very simple form of component creation in our glue language without priorities.

To summarize, our contributions are the following:

– We introduce a new process calculus, CAB, that extends the BIP framework with dynamic composition (or glue) capabilities.

– We demonstrate the expressiveness of priorities in the BIP framework by showing that BIP glues, composing simple CCS processes, is enough to obtain a Turing-complete language, and that Turing completeness is lost if we remove priorities.
– We show that Turing-completeness can be retained if we introduce more dynamic aspects in the language, namely a simple form of component creation.

The paper is organized as follows. Section 2 introduces the CAB process calculus and defines its operational semantics in SOS style. Section 3 proves our first result: CAB, restricted to a control language consisting of BIP glues, is Turing-complete. Section 4 proves our two other results: dropping priorities from CAB results in a non Turing-complete language; adding component creation to the control language without priorities is enough to regain Turing-completeness. Section 5 concludes the paper and discusses some related works.

2 CAB: syntax and semantics

We introduce in this section the CAB process calculus. In order to explain its constructs, as well as to make its relationship with the BIP framework clear, we begin by recalling the definition of the latter.

The BIP framework. We rely on the description of the BIP framework provided by [2,3]. A BIP component is simply a labeled transition system (LTS), whose labels are ports[1].

Definition 1. *A* component *is an LTS* $B = (Q, P, \rightarrow)$ *where*

1. Q *is a set of states*
2. P *is a set of ports*
3. $\rightarrow \subseteq Q \times P \times Q$ *is a set of transitions. We use* $q \xrightarrow{a} q'$ *to denote* $(q, a, q') \in \rightarrow$.

Components can be composed (glued) to form systems. A composition is given by a set of rules (the glue) that enforce synchronization and priority constraints among them.

Definition 2. *A* BIP system S *that glues together* n *components* $B_i = (Q_i, P_i, \rightarrow_i)$ *where ports and states are pairwise disjoint, is an LTS* $S = (\mathbb{Q}, \mathbb{P}, \rightarrow_S)$ *where* $\mathbb{Q} = \prod_{i=1}^{n} Q_i$, $\mathbb{P} = \bigcup_{i=1}^{n} P_i$ *and where* $\rightarrow_S$ *is a relation derivable as the least relation satisfying a finite set of rules*[2] *obeying the following format:*

$$r : \frac{\{B_i \xrightarrow{a_i} B_i'\}_{i \in I} \quad \{B_j \xrightarrow{b_j^k} \not\rightarrow \mid k \in [1..m_j]\}_{j \in J}}{(B_1, \ldots, B_n) \xrightarrow{a}_S (B_1', \ldots, B_n')} \tag{1}$$

[1] This is a difference with the definition in [3], where labels are defined to be *sets* of ports. We have adopted labels as simple ports in this paper to simplify the presentation. Our results are not impacted by this decision, however, for our processes only have a fixed finite number of distinct ports, so that we can always bijectively map a set of ports onto a single port.

[2] The finiteness of the set of rules defining a glue seems implicit in [3].

where I and J are sets of indexes in $[1,n]$, $B'_i = B_i$ if $I \notin I$, and $I \neq \emptyset$ (i.e. there is at least one positive premise).

Note that by definition there is at most one positive premise for each B_i in a rule in BIP format. The key features of the BIP framework are: (i) the ability to build hierarchical components; (ii) the concept of an explicit entity (the glue) responsible for the composition of components; (iii) the support of multipoint synchronizations, manifested by the positive premises in glue rules; (iv) The presence of priority constraints, given by the negative premises in glue rules.

The CAB calculus. As indicated in the introduction, we retain for CAB the general structure of composite components suggested by the BIP framework: a component in CAB takes the form $l[C_1; \ldots; C_n \rhd B]$, where l is the name of the component, $C_1, \ldots, C_n$ are its subcomponents and B is the glue. In contrast to glues in BIP, a glue in CAB can evolve over time, corresponding to changes in the synchronization and priority constraints among components, and is given by a term of a process calculus we call the *glue language*. We adopt in this paper a very simple glue language featuring:

- *Action prefix* $\alpha.B$, where α is an action, and B a continuation glue. The presence of action prefix in our glue language allows the definition of dynamic glues.
- *Parallel composition* $B_1 \parallel B_2$, where B_1 and B_2 are two glues. The parallel composition of glues can be interpreted as an *and* operator combining the synchronization and priority constraints embodied by B_1 and B_2. It is important to note that the two branches B_1 and B_2 in a parallel composition $B_1 \parallel B_2$ do *not* interact.
- *Recursion* $recX.B$, where X is a process variable, and B a glue. This allows the definition of glues with cyclic behaviors.

Formally, let $\mathcal{N}_P = \{a, b, c \ldots\}$ and $\mathcal{N}_C = \{h, k, l \ldots\}$ be denumerable sets of ports names and components names respectively. The CAB calculus is parametric over a set $\mathcal{P}$ of *primitive components* defined as labeled transition systems with labels in $\mathcal{N}_P$. We define CAB($\mathcal{P}$) processes as follows:

Definition 3 (CAB). *The set of CAB($\mathcal{P}$) processes is described by the following grammar, where P denotes an element of $\mathcal{P}$:*

$$\begin{array}{ll} S ::= l[C \rhd B] \mid l[P] & Act ::= \emptyset \mid \{evt\} \\ C ::= \mathbf{0} \mid S \mid C; C & evt ::= l : a \mid evt, evt \\ B ::= \mathbf{0} \mid \langle Act, Tag, Act \rangle.B \mid B \parallel B \mid rec\, X.B \mid X & Tag ::= \tau \mid a \end{array}$$

In order to simplify notation we write $l : a$ instead of $l : \{a\}$, and a instead of $l : a$ when it is clear from the context which component is providing event a. We abbreviate $\alpha.\mathbf{0}$ to α. We define $S.\mathtt{nm} = l$ if $S = l[P]$ or $S = l[C \rhd B]$ for some P, C, B (i.e. the function $\mathtt{nm}$ returns the name of an individual component S).

Actions in our glue language differ from those in classical process calculi, such as CCS, for they play different roles: they embody synchronization and priority

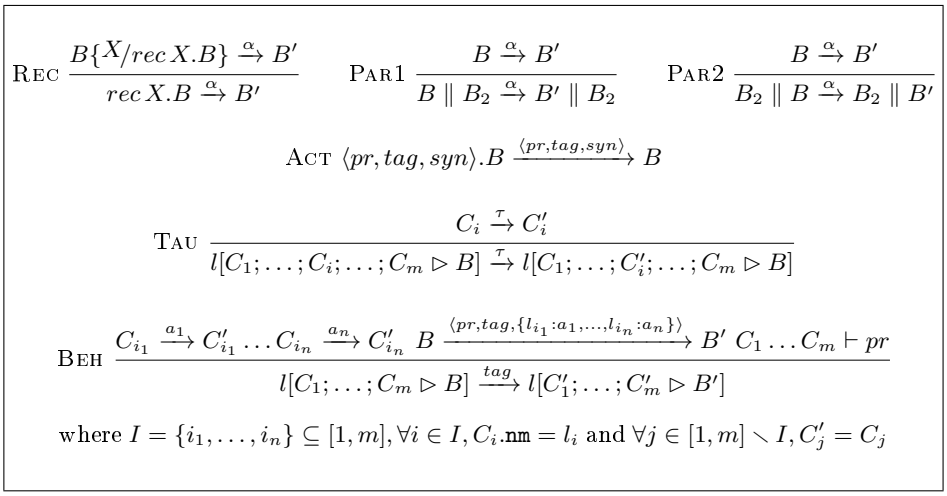


Fig. 1: A labeled transition system semantics for CAB($\mathcal{P}$).

constraints that apply to subcomponents in a composition, and they provide a form of label renaming. An action is a triplet of the form $\langle pr, tag, syn\rangle$, where pr is a priority constraint (i.e. events in subcomponents which would preempt the synchronization syn), syn is a synchronization constraint (i.e. events to be synchronized between subcomponents), and tag is an event made visible by the composite as a result of a successful syn synchronization.

Hence a glue B of the form $\langle\{l : a\}, t, \{l_1 : c_1, l_2 : c_2\}\rangle.B'$ specifies a synchronization constraint between two subcomponents l_1 and l_2: if the first one is ready to perform event c_1, and the other is ready to perform event c_2, then the composition is ready to perform event t, provided that subcomponent l is not ready to perform event a. When the event t of the composite is performed (implying the two subcomponents l_1 and l_2 have performed events c_1 and c_2, respectively), a new glue B' is then put in place to control the behavior of the composite. Note that tag t can be either τ (which denotes an internal event) or a port (an event). Hence a tag $t = \tau$ results in a synchronization between subcomponents that takes place silently, with no implication from the environment of the composite. A tag $t \neq \tau$ subjects the evolution of the composite to the availability of an appropriate synchronization on t in the environment of the composite.

The operational semantics of CAB($\mathcal{P}$) is defined as the least labeled transition relation derivable by the inference rules in Figure 1. Rules for parallel composition and recursion are defined as usual. Rules BEH and TAU define the evolution of an aggregation of components inside a composite named l. Rule BEH stipulates that if a glue B is ready to perform an action $\langle pr, tag, \{l_1 : a_1, \ldots, l_n : a_n\}\rangle$ and components named $l_1, \ldots, l_n$ are ready to perform $a_1, \ldots, a_n$ respectively, then their composition is ready to perform action tag, provided priority constraint pr is satisfied. Having a priority constraint satisfied is defined as follows.

Let $pr = \{l_{j_1} : c_{j_1}, \ldots, l_{j_k} : c_{j_k}\}$ with $J = \{j_1 \ldots j_m\} \subseteq [1, m]$, we say that $C_1 \ldots C_m \vdash pr$ iff for every $i \in J$, $S_i \stackrel{c_i}{\not\longrightarrow}$ with $S_i.\mathtt{nm} = l_i$ and $S_i \in \{C_1, \ldots, C_m\}$. If $pr = \emptyset$ we are not imposing any priority policy on the synchronization. Similarly, with an action of the form $\langle pr, tag, \emptyset \rangle$ there is no synchronization requirement, but the environment of the composite must be ready to perform tag in order for the system to evolve.

Notation 1 *We denote with* $!\alpha.P$ *the process* $rec\, X.\ \alpha.(P \parallel X)$. *We use* $\longrightarrow$ *to denote the relation* $\stackrel{\tau}{\rightarrow}$. *We use* $\prod_{i=1}^{n} B_i$ *to denote* $B_1 \parallel \ldots \parallel B_n$[3].

Encoding BIP. The operational semantics, and in particular rule BEH, above was defined so as to mimic very closely the capabilities of glues in BIP. We now clarify the relationship between CAB($\mathcal{P}$) and BIP systems defined over a set $\mathcal{P}$ of components. We can encode a BIP glue G in CAB($\mathcal{P}$) as follows. By definition, G is given by a finite set of rules r that obey the format given in Definition 2. Let r be such a rule:

$$r : \frac{\{C_i \stackrel{a_i}{\longrightarrow} C_i'\}_{i \in I} \quad \{C_j \stackrel{c_j^k}{\not\longrightarrow} |\ k \in [1..m_j]\}_{j \in J}}{(C_1, \ldots, C_n) \stackrel{tag}{\longrightarrow} (C_1', \ldots, C_n')}$$

where I and J are set of indexes in $[1, n]$. The encoding $[\![r]\!]$ of rule r in CAB($\mathcal{P}$) is defined as:

$$[\![r]\!] = !\langle \{h_j : c_j^k \mid k \in [1, m_j]\}_{j \in J}, tag, \{h_i : a_i\}_{i \in I} \rangle$$

A BIP composition S with glue rules $r_1, \ldots, r_p$ and components $C_1, \ldots, C_n \in \mathcal{P}$ can thus be encoded as follows:

$$[\![S]\!] = l[h_1[C_1]; \ldots; h_n[C_n] \rhd \prod_{i=1}^{p} [\![r_i]\!]]$$

By construction, we obtain:

Theorem 2. *BIP systems defined over a set* $\mathcal{P}$ *of components can be encoded in CAB(*$\mathcal{P}$*): any BIP system* S *is strongly bisimilar to its encoding* $[\![S]\!]$.

3 Turing-completeness of CAB

In this section as in the rest of the paper, we work within CAB($\emptyset$), which, for simplicity, we denote CAB. We show the Turing-completeness of CAB by proving we can encode Minsky machines into it. This gives us a result on the *intrinsic* expressive power of the CAB glue language, in the sense that it does not depend on the presence of primitive components: we only construct component systems using glue language terms. Note that this is equivalent to considering

[3] The parallel operator $\parallel$ is commutative and associative modulo strong bisimilarity.

$$\text{M-Inc}\ \frac{i : \texttt{INC}(r_j) \quad m'_j = m_j + 1 \quad m'_{1-j} = m_{1-j}}{(i, m_0, m_1) \longrightarrow_{\text{M}} (i+1, m'_0, m'_1)}$$

$$\text{M-Dec}\ \frac{i : \texttt{DECJ}(r_j, s) \qquad m_j \neq 0 \qquad m'_j = m_j - 1 \qquad m'_{1-j} = m_{1-j}}{(i, m_0, m_1) \longrightarrow_{\text{M}} (i+1, m'_0, m'_1)}$$

$$\text{M-Jmp}\ \frac{i : \texttt{DECJ}(r_j, s) \quad m_j = 0}{(i, m_0, m_1) \longrightarrow_{\text{M}} (s, m_0, m_1)} \qquad \text{M-Halt}\ \frac{i : \texttt{HALT}}{(i, m_0, m_1) \not\rightarrow_{\text{M}}}$$

Fig. 2: Semantics of Minsky machines

only primitive components which are labeled transition systems defined by CAB terms of the form $l[\mathbf{0} \triangleright B]$, where B is a term with actions of the form $\langle \emptyset, a, \emptyset \rangle$. For reference, these primitive processes are given by terms of the following grammar, whose operational semantics is given by rules Rec, Par1, Par2, and Act in Figure 1:

$$B ::= \mathbf{0} \mid \langle \emptyset, a, \emptyset \rangle.B \mid B \parallel B \mid rec\, X.B \mid X$$

Minsky Machines. Minsky machines [10] provide a Turing-complete model of computation. A Minsky machine is composed of a set of sequential, labeled instructions, and at least two registers. Registers r_j ($j \in \{0, 1\}$) can hold arbitrarily large natural numbers. Instructions $(1 : I_1), \ldots, (n : I_n)$ can be of two kinds: $\texttt{INC}(r_j)$ adds 1 to register r_j and proceeds to the next instruction; $\texttt{DECJ}(r_j, s)$ jumps to instruction s if r_j is zero, otherwise it decreases register r_j by 1 and proceeds to the next instruction. A Minsky machine includes a program counter p indicating the label of the instruction being executed. In its initial state, the machine has both registers initialized to m_0 and m_1 respectively, and the program counter p is set to the first instruction. The Minsky machine stops whenever the program counter is set to the HALT instruction. A *configuration* of a Minsky machine is a tuple (i, m_0, m_1); it consists of the current program counter and the values of the registers. Formally, the reduction relation over configurations of a Minsky machine , denoted $\longrightarrow_{\text{M}}$, is defined in Figure 2.

The encoding. The encoding of Minsky machines in CAB, denoted $[\![\cdot]\!]_1$, is given in Figure 3. We now give some intuitions on it. Given a Minsky machine M, we encode it as a system m. m contains three components: the two registers r_0 and r_1, and the program counter. The instructions of the machine are encoded in the glue of m. Numbers inside registers are encoded in the glue as the parallel composition of as many occurrences of the unit process $\langle \emptyset, u_j, \emptyset \rangle$ as the number to be encoded. An increment simply adds an occurrence of the unit process $\langle \emptyset, u_j, \emptyset \rangle$ to the register. The decrement and jump is encoded as the parallel composition

$$[\![R_j = m]\!]_1 = r_j[\mathbf{0} \rhd \prod_1^m \langle \emptyset, u_j, \emptyset\rangle \parallel !\langle \emptyset, z_j, \emptyset\rangle \parallel !\langle \emptyset, inc_j, \emptyset\rangle.\langle \emptyset, u_j, \emptyset\rangle]$$

INSTRUCTIONS $(i : I_i)$

$$\begin{aligned}
[\![(i : \texttt{INC}(r_j))]\!]_1 &= !\langle \emptyset, \tau, \{p_i, inc_j, next_{i+1}\}\rangle \\
[\![(i : \texttt{DECJ}(r_j, s))]\!]_1 &= !\langle \emptyset, \tau, \{p_i, u_j, next_{i+1}\}\rangle \parallel !\langle r_j : u_j, \tau, \{p_i, z_j, next_s\}\rangle) \\
[\![(i : \texttt{HALT})]\!]_1 &= \langle \emptyset, halt, p_i\rangle
\end{aligned}$$

Fig. 3: Encoding of Minsky machines into CAB.

of the two branches. The decrement branch simply removes one occurrence of the unit process $\langle \emptyset, u_j, \emptyset\rangle$, if such occurrence is available. The jump branch is guarded by the priority $r_j : u_j$. In other words, to be able to execute the jump, it is necessary to check that the register is indeed empty. If this is the case the program counter is updated accordingly. More formally, the encoding of a configuration in the Minsky machine is defined as follows:

Definition 4. *Let M be a Minsky machine and (k, m_0, m_1) one of its configurations. The encoding of $[\![k, m_0, m_1]\!]_1$ is defined as*

$$\begin{aligned}
m[[\![R_0 = m_0]\!]_1; [\![R_1 = m_1]\!]_1; pr[\mathbf{0} \rhd \prod_{i=1}^n !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle]; \\
pr[\mathbf{0} \rhd \langle \emptyset, p_k, \emptyset\rangle \parallel \prod_{i=1}^n !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle] \rhd \prod_{i=1}^n [\![i : I_i]\!]_1]
\end{aligned}$$

where the encoding of registers and instructions is defined in Figure 3.

Notice that in order to synchronize at the same time p_i and $next_i$ we have to duplicate the component representing the program counter. This does not introduce non determinism as only one instance of the action $\langle \emptyset, p_i, \emptyset\rangle$ is available at every step.

The correctness of the encoding follows by a case analysis on the type of instruction performed when the program counter reaches k. This is formalized by the following Lemma.

Lemma 1. *Let M be a Minsky machine and (k, m_0, m_1) one of its configuration then $(k, m_0, m_1) \longrightarrow_{\mathrm{M}} (k', m_0', m_1')$ iff $[\![k, m_0, m_1]\!]_1 \longrightarrow [\![k', m_0', m_1']\!]_1$.*

Proof (Sketch). Here we show only that if $(k, m_0, m_1) \longrightarrow_{\mathrm{M}} (k', m_0', m_1')$ then $[\![k, m_0, m_1]\!]_1 \longrightarrow [\![k', m_0', m_1']\!]_1$ when the k-th instruction is a decrement on register $m_0 > 0$. The other cases and the other direction are similar or simpler.

Then, from Definition 4, we have that

$$\begin{aligned}
m[[\![R_0 = m_0]\!]_1; [\![R_1 = m_1]\!]_1; pr[\mathbf{0} \rhd \prod_{i=1}^n !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle]; \\
pr[\mathbf{0} \rhd \langle \emptyset, p_k, \emptyset\rangle \parallel \prod_{i=1}^n !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle] \rhd \prod_{i=1}^n [\![i : I_i]\!]_1]
\end{aligned}$$

where the k-th instruction is encoded as

$$!\langle \emptyset, \tau, \{p_k, u_0, next_{k+1}\}\rangle \parallel !\langle r_0 : u_0, \tau, \{p_k, z_0, next_s\}\rangle)$$

and $m_0' = m_0 - 1$, $k' = k + 1$. In this case, the only possible evolution is the one that synchronizes the program counter p_k, the unit u_0 inside register r_0 and $next_{k+1}$, evolving into the system:

$$m[[\![R_0 = m_0 - 1]\!]_1; [\![R_1 = m_1]\!]_1; pr[\mathbf{0} \rhd \langle \emptyset, p_{k+1}, \emptyset\rangle \parallel \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle];$$
$$pr[\mathbf{0} \rhd \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle] \rhd \prod_{i=1}^{n} [\![i : I_i]\!]_1]$$

Now, it is easy to see that the system above corresponds to $[\![k', m_0', m_1']\!]_1$. □

By means of the previous lemma, we can state the operational correspondence between M and its encoding $[\![M]\!]_1$.

Theorem 3. *Let M be a Minsky machine and $[\![M]\!]_1$ as defined in Definition 4. Then M halts with registers $R_i = m_i'$ for $i \in [0, 1]$ iff $[\![M]\!]_1 \xrightarrow{halt}$ and locations r_i for $i \in [0, 1]$ is $[\![R_i = m_i']\!]_1$.*

It is important to notice that our encoding relies on elementary components of the form $l[\mathbf{0} \rhd B]$, which are glued together by glue terms which are essentially in BIP format, as discussed in Section 2. The above theorem gives us actually a stronger result which says that the subset of CAB where glues are restricted to be in BIP format, and where primitive components correspond to labeled transition systems given by elementary components of the form $l[\mathbf{0} \rhd B]$, is Turing-complete.

4 Expressivity of CAB variants

We have shown that CAB is Turing powerful. We now investigate the sources of expressiveness in the language. The first thing we show is that in the encoding given in Section 3 the presence of priorities is essential. Indeed we can prove that if we consider a fragment of CAB without priorities the resulting language is not Turing powerful anymore. This can be proven by providing an encoding into Petri nets, a well known non Turing-powerful model.

4.1 CAB without priorities

A *Petri net* (see e.g. [6]) is a tuple $N = (P, T, m_0)$, where P and T are finite sets of *places* and *transitions*, respectively. A finite multiset over the set S of places is called a *marking*, and m_0 is the initial marking. Given a marking m and a place p, we say that the place p contains $m(p)$ *tokens* in the marking m if there

are $m(p)$ occurrences of p in the multiset m. A transition is a pair of markings written in the form $m' \Rightarrow m''$. The marking m of a Petri net can be modified by means of transitions firing: a transition $m' \Rightarrow m''$ can fire if $m(p) \geq m'(p)$ for every place $p \in S$; upon transition firing the new marking of the net becomes $n = (m \setminus m') \uplus m''$ where $\setminus$ and $\uplus$ are the difference and union operators for multisets, respectively. This is written as $m \to n$.

We denote the fragment of CAB without priorities as CAB^{-p}. This fragment is obtained by replacing production $\langle Act, Tag, Act\rangle$ with $\langle \emptyset, Tag, Act\rangle$ in Definition 3. Before presenting the encoding into Petri Nets, we introduce some more terminology: we define a notion of top level actions in the glue of a component.

Definition 5 (top). *Let $l[C \rhd B]$ be a system in CAB. $top(B)$ is defined inductively on the structure of the glue B as follows:*

$$top(\mathbf{0}) = top(X) ::= \emptyset \qquad top(\langle pr, tag, syn\rangle.B) ::= \{\langle pr, tag, syn\rangle\}$$
$$top(recX.B) ::= top(B) \qquad top(B_1 \parallel B_2) ::= top(B_1) \cup top(B_2)$$

We also define how to build the graph of precedence of a glue B:

Definition 6. *Let $l[C \rhd B]$ be a system in CAB. The graph of B, denoted with $\mathcal{G}(B) = (Nodes(B), Edges(B))$, is a directed graph, inductively defined as:*

$$\begin{aligned}
\mathcal{G}(\mathbf{0}) \quad &::= Nodes(B) = \{\mathbf{0}\},\\
&\quad\ Edges(B) = \emptyset\\
\mathcal{G}(\langle act\rangle.B_1) \ &::= Nodes(B) = \{\langle act\rangle\} \cup Nodes(B_1),\\
&\quad\ Edges(B) = \{\langle act\rangle \to x \mid x \in top(B_1)\} \cup Edges(B_1)\\
\mathcal{G}(B_1 \parallel B_2) \ &::= Nodes(B) = Nodes(B_1) \cup Nodes(B_2),\\
&\quad\ Edges(B) = Edges(B_1) \cup Edges(B_2)\\
\mathcal{G}(recX.B_1) \ &::= Nodes(B) = Nodes(B_1)\\
&\quad\ Edges(B) = Edges(B_1) \text{ where every time we encounter}\\
&\qquad\qquad X \text{ we add an edge to the nodes in } top(B_1)
\end{aligned}$$

Let $n \in Nodes(B)$, we denote with $Adj(n)$ the list of nodes adjacent to n.

The idea is that every system is a Petri Net and the marking represents the components that are ready to interact at a given instant. Transitions mimic the semantics of CAB^{-p} systems. The construction of the Petri Net is inductive on the hierarchy of components: let $S = l_S[S_1; \ldots; S_m \rhd B_S]$ be a system in CAB^{-p}. We assume that k is the maximum number of levels of nesting in S. We decorate every location in S with the corresponding level of nesting in S, from 1 the innermost, to k the outermost level.

Let $\mathcal{PN}(S_i) = (P(S_i), T(S_i), m_0(S_i))$ be the Petri Net for the subsystem S_i for all $i \in [1, m]$. $\mathcal{PN}(S)$ is built by taking:

- as set of places, the set of all places of the subnets for $S_1 \ldots S_n$ plus all the nodes in the graph of the behavior B_S:

$$P(S) = \bigcup_{i=1}^{n} P(S_i) \cup \{[l_S^k : \langle \emptyset, tag, syn\rangle] \mid \langle \emptyset, tag, syn\rangle \in Nodes(B_S)\};$$

Notice that there is a bijection between nodes in the graphs of glues and the places in the Petri Net. Hence for every node n in the graph of glue located at l in level j there exists a distinctive place $[l^j : n]$ and vice-versa.

- as set of transitions all the transitions of subnets $\mathcal{PN}(S_1) \ldots \mathcal{PN}(S_n)$ plus for all nodes $\langle \emptyset, tag, syn \rangle$ in $Nodes(B_S)$ where $tag = \tau$ we add a set of transitions that:
  - Take as precondition, recursively on the part syn of the nodes considered, all the places $[l^j : \langle \emptyset, t, s \rangle]$ for $j \in [1, k-1]$ and such that $l : t$ appears in the synchronization part syn in one of the nodes. Notice that, this accounts in considering in a single transition all the components involved in a τ step: i.e. the places involved in the precondition correspond to all the leafs in the derivation tree of the τ step.
  - Take as postcondition all the places built from nodes in the adjacent list of all the nodes obtained by places in the preconditions.

  For instance, consider the system

$$l^3[l^2[l^1[\mathbf{0} \rhd \langle \emptyset, a, \emptyset \rangle.\mathbf{0}] \rhd \langle \emptyset, b, \{l^1 : a\} \rangle.\mathbf{0}] \rhd \langle \emptyset, \tau, \{l^2 : b\} \rangle.\mathbf{0}]$$

  here there is a single transition that takes as precondition the places: $\{[l^3 : \langle \emptyset, \tau, \{l^2 : b\} \rangle], [l^2 : \langle \emptyset, b, \{l^1 : a\} \rangle], [l^3 : \langle \emptyset, a, \emptyset \rangle]\}$ and as post condition the places $\{[l^1 : \mathbf{0}], [l^2 : \mathbf{0}], [l^3 : \mathbf{0}]\}$
- as initial marking, the initial marking of all subnets plus the nodes corresponding to the top level actions in B_S:

$$m_0(S) = \uplus_{i=1}^{n} m_0(S_i) \uplus \{[l_S^k : n] \mid n \in top(B_S)\}$$

The correctness of the above construction follows by induction on the nesting of components.

Theorem 4. *Let $S = l_S[S_1; \ldots; S_m \rhd B_S]$ be a system in CAB^{-p}, and $\mathcal{PN}(S) = (P(S), T(S), m_0(S))$ the corresponding Petri Net. Then $S \longrightarrow S'$ iff there exists a marking m' such that $m_0(S) \Rightarrow m'$ and m' is a marking that takes all the top level actions in S'.*

Proof. Here we show only the correctness direction, soundness is similar. Let $S = l_S[S_1; \ldots; S_m \rhd B_S]$ be a system in CAB^{-p}, and $m_0(S)$ the initial marking in the Petri Net constructed as described above. The proof proceeds by induction on the nesting of components in S. If $S \longrightarrow S'$ then we have that either rule Beh or Tau has been used. The case of Tau follows by inductive hypothesis. Instead if the τ step comes from Beh , we have that there exists an action $\langle \emptyset, \tau, \{a_1 \ldots a_n\} \rangle$ at top level in B_S. Moreover we have $C_{i_1} \ldots C_{i_n}$ components that are offering actions $a_1 \ldots a_n$ respectively. Hence at top level in these components we have an action $\langle \emptyset, a_{i_j}, syn \rangle$ for $j \in [1, n]$. Therefore, by construction we have a token in all these places and the transition can fire, moving all tokens in the successors of the action: i.e. in all the nodes of the adjacency list, that by construction corresponds to the new action at top level in S'. □

4.2 Recovering expressiveness

We, now, introduce a new construct to CAB^{-p} to recover the loss of expressiveness due to the absence of priorities. We consider an operator that adds new components inside a system. To this aim, we add to Definition 3 the following production:

$$B ::= new\ S$$

with this operational semantics:

$$\textsc{New}\ new\ S \xrightarrow{new\ S} \mathbf{0} \qquad \textsc{Cre}\ \frac{B \xrightarrow{new\ S} B'}{l[C \rhd B] \xrightarrow{\tau} l[C; S \rhd B']}$$

Thanks to the interplay between the creation of new components and recursion we can re-obtain Turing equivalence. The result, similarly to the one in Section 3, is obtained by resorting to an encoding of Minsky machines. We proceed by giving some intuitions on the encoding given in Figure 4. Registers are encoded as a hierarchy of components that handle both the representation of the number and a mechanism to increment or decrement. The nesting of these components represents the number contained. At every instant, the mechanism controlling the register is placed in the innermost position. Thus, whenever an increment takes place, a new component is created inside the deepest level and all the control is transfered to the newly created object: this is the role of $a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset \rangle]$ which activates the current instance. On the contrary, in case of a decrement, the current instance is deactivated: i.e. it remains as garbage but it cannot be used anymore and a signal is passed to the upper component so to activate decrements and increments at the proper level of nesting. Notice, that in order to communicate with the active instance, it is necessary to equip every level of the nesting with a process Fwd. This process is responsible for forwarding increment and decrement events to reach the component that controls the simulation of the computation. Without loss of generality, we assume that registers are initialized to zero. The following definition formalizes the encoding of a Minsky machine M:

Definition 7. *Let M be a Minsky machine with registers initialized to 0 and program counter set to 1: its encoding $[\![M]\!]_2$ is*

$$m[[\![R_0 = 0]\!]_2; [\![R_1 = 0]\!]_2; pr[\mathbf{0} \rhd \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset \rangle.\langle \emptyset, p_i, \emptyset \rangle];$$
$$pr[\mathbf{0} \rhd \langle \emptyset, p_1, \emptyset \rangle \parallel \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset \rangle.\langle \emptyset, p_i, \emptyset \rangle] \rhd \prod_{i=1}^{n} [\![i : I_i]\!]_2]$$

where the encoding of registers and instructions is defined in Figure 4. [4]

[4] Notice that the interplay of recursion and creation of new components is implicit in the definition of INC and $Level$. The same thing could have been written as: $!recX\langle \emptyset, inc_j, act_j \rangle.new\ a[a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset \rangle] \rhd Fwd \parallel DEC \parallel X]$.

$$
\begin{array}{ll}
[\![R_j = 0]\!]_2 & ::= r_j[a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset\rangle \parallel !\langle \emptyset, zero_j, \emptyset\rangle.\langle \emptyset, act_j, \emptyset\rangle] \rhd Fwd \parallel Z \parallel INC] \\
Fwd & ::= !\langle \emptyset, inc_j, inc_j\rangle \parallel !\langle \emptyset, dec_j, dec_j\rangle \\
Z & ::= !\langle \emptyset, z_j, act_j\rangle.\langle \emptyset, \tau, zero_j\rangle \\
INC & ::= !\langle \emptyset, inc_j, act_j\rangle.new\ Level \\
Level & ::= a[a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset\rangle] \rhd Fwd \parallel DEC \parallel INC] \\
DEC & ::= \langle \emptyset, dec_j, act_j\rangle.\langle \emptyset, act_j, \emptyset\rangle
\end{array}
$$

INSTRUCTIONS $(i : I_i)$

$$
\begin{array}{ll}
[\![(i : \texttt{INC}(r_j))]\!]_2 & = !\langle \emptyset, \tau, \{p_i, inc_j, next_{i+1}\}\rangle \\
[\![(i : \texttt{DECJ}(r_j, s))]\!]_2 & = !\langle \emptyset, \tau, \{p_i, dec_j, next_{i+1}\}\rangle \parallel !\langle \emptyset, \tau, \{p_i, z_j, next_s\}\rangle) \\
[\![(i : \texttt{HALT})]\!]_2 & = \langle \emptyset, halt, p_i\rangle
\end{array}
$$

Fig. 4: Encoding of Minsky machines into CAB without priorities.

Similarly as before, the correctness of the encoding follows by a case analysis on the type of instruction performed when the program counter reaches k. Notice that, depending on the specific computation there can be components as $a[a[\mathbf{0} \rhd \mathbf{0}] \rhd Fwd \parallel INC]$ "floating" in the system. Nevertheless this garbage can be ignored as it is never re-used: i.e. it cannot interact with the rest of the system.

Lemma 2. *Let M be a Minsky machine and (k, m_0, m_1) one of its configuration then $(k, m_0, m_1) \longrightarrow_{\mathrm{M}} (k', m_0', m_1')$ iff $[\![k, m_0, m_1]\!]_2 \longrightarrow [\![k', m_0', m_1']\!]_2$.*

Proof (Sketch). Here we show only that if $(k, m_0, m_1) \longrightarrow_{\mathrm{M}} (k', m_0', m_1')$ then $[\![k, m_0, m_1]\!]_2 \longrightarrow [\![k', m_0', m_1']\!]_2$ when the k-th instruction is a decrement on register $m_0 > 0$. The other cases and the other direction are similar or simpler.

We first define $[\![k, m_0, m_1]\!]_2$, for the sake of simplicity we will not consider the occurrences of garbage objects, taking for grant that those will not interfere with the computation.

$$
\begin{array}{rl}
[\![k, m_0, m_1]\!]_2 ::= m[[\![R_0 = m_0]\!]_2; [\![R_1 = m_1]\!]_2; pr[\mathbf{0} \rhd \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle]; & \\
pr[\mathbf{0} \rhd \langle \emptyset, p_k, \emptyset\rangle \parallel \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle] \rhd \prod_{i=1}^{n} [\![i : I_i]\!]_2] &
\end{array}
$$

where

$$
\begin{array}{l}
[\![R_j = m_j]\!]_2 ::= r_j[a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset\rangle \parallel !\langle \emptyset, zero_j, \emptyset\rangle.\langle \emptyset, act_j, \emptyset\rangle], \\
C[\ldots C[a[a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset\rangle] \rhd Fwd \parallel DEC \parallel INC]] \ldots] \rhd Fwd \parallel Z \parallel INC]
\end{array}
$$

and $C[\bullet] = a[a[\mathbf{0} \rhd \mathbf{0}], \bullet \rhd Fwd \parallel DEC \parallel INC]$ is repeated m_j times. The k-th instruction is encoded as

$$
!\langle \emptyset, \tau, \{p_k, dec_0, next_{k+1}\}\rangle \parallel !\langle r_0 : u_0, \tau, \{p_k, z_0, next_s\}\rangle)
$$

and $m'_0 = m_0 - 1$, $k' = k + 1$. In this case, the only possible evolution is the one that synchronizes the program counter p_k, the message dec_0 inside register r_0 and $next_{k+1}$, evolving into the system:

$$m[[\![R_0 = m_0 - 1]\!]_2; [\![R_1 = m_1]\!]_2; pr[\mathbf{0} \rhd \langle \emptyset, p_{k+1}, \emptyset \rangle \parallel \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset \rangle.\langle \emptyset, p_i, \emptyset \rangle];$$
$$pr[\mathbf{0} \rhd \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset \rangle.\langle \emptyset, p_i, \emptyset \rangle] \rhd \prod_{i=1}^{n} [\![i : I_i]\!]_2]$$

Notice that the message on dec_0 will start a chain of synchronizations between components $a[\dots]$ through the Fwd event to reach the deepest component and then activate the real decrement. It is easy to conclude that the system above corresponds to $[\![k', m'_0, m'_1]\!]_2$. □

The previous lemma allows us to conclude:

Theorem 5. *Let M be a Minsky machine and $[\![M]\!]_2$ as defined in Definition 7. Then M halts with registers $R_i = m'_i$ for $i \in [0, 1]$ iff $[\![M]\!]_2 \xrightarrow{halt}$ and locations r_i for $i \in [0, 1]$ is $[\![R_i = m'_i]\!]_2$.*

5 Final Remarks

We have taken in this paper a decidedly process algebraic view of glues in component-based systems, introducing an alternate view, and an extension, of the BIP framework in the form of the CAB process calculus. We have studied the expressiveness of CAB, which gave us a way to characterize the intrinsic (i.e. not relatively to a predefined family of components) expressive power of its glue language. We have shown that, while being very simple, the calculus is Turing-complete thanks mainly to the presence of priorities. As a matter of fact, we have shown that the fragment of CAB where priorities have been removed is only as expressive as Petri nets, which is a testament to the gain in expressive power obtained through the use of priorities. However expressiveness can be recovered in a calculus without priorities if dynamic operators are added to the language.

We have already discussed in the introduction the relations with the BIP framework and seen how the present paper brings new light on BIP expressiveness. Here we relate our paper to other works studying the expressiveness of multiparty synchronization or priority. Multiparty synchronization has been proposed in several process calculi. One of the first proposals is CSP [7] where synchronization can take place among all processes that share a channel with the same name. A recent work by Laneve and Vitale [8] has shown that a calculus able to synchronize on n channels is strictly more expressive than one that can only synchronize up to $n - 1$ channels. [5] shows a similar result in the context of a concurrent logic calculus. In the current paper we have mostly shown the benefit of priorities for expressiveness. However we suspect that multiparty

synchronization is also important for expressiveness. In our two encodings of Minsky machines in Section 3 and in Section 4, we rely decisively on 3-way synchronization; whether it is absolutely required is a question for further study.

Several works tackle the problem of adding priority mechanisms in a process calculus [4]. In [11] it has been shown that CCS enriched with a form of priority guards is strictly more expressive than CCS: essentially, it is possible to model the leader election problem in CCS with priorities, which is not the case with plain CCS. Analogously, [12] shows that a core calculus similar to CCS, if extended with several kinds of priorities, can model the leader election problem while the core calculus can not. Both these studies state the impossibility to encode the calculus with priorities in the plain calculus. In contrast, we show in this paper an absolute increase in expressiveness from Petri Nets to Minsky machines. Closer to the present work is the paper in [1], where the authors show that CCS without restriction, and with replication instead of recursion, can be encoded into Petri Nets while the same calculus enriched with priorities and a weak form of restriction is Turing-powerful. Compared to [1] we are considering recursive processes instead of replicated ones thus the drop of expressiveness when not using priorities is stronger in our case.

As for future work, we plan to investigate other, more involved, forms of dynamic configuration of components. Moreover we are interested in understanding if our result of Turing completeness can be related to the ability of simulating all recursively enumerable LTSs thus making unnecessary the presence of the parameter $\mathcal{P}$ in the full calculus CAB($\mathcal{P}$).

References

1. J. Aranda, F. Valencia, and C. Versari. On the expressive power of restriction and priorities in ccs with replication. In *FOSSACS*, pages 242–256. Springer, 2009.
2. S. Bliudze and J. Sifakis. The algebra of connectors - structuring interaction in bip. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.
3. S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008.
4. R. Cleaveland, G. Lüttgen, and V. Natarajan. Priority in process algebra. Technical report, Nasa, 1999.
5. C. Di Giusto, M. Gabbrielli, and M. C. Meo. On the expressive power of multiple heads in chr. *To appear in ACM Transactions on Computational Logic*, 2010.
6. J. Esparza and M. Nielsen. Decidability issues for petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.
7. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.
8. C. Laneve and A. Vitale. The expressive power of synchronizations. In *LICS '10*, pages 382–391, Washington, DC, USA, 2010. IEEE Computer Society.
9. F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3), 2003.
10. M. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.
11. I. Phillips. CCS with priority guards. *J. Log. Algebr. Progr.*, 75(1):139–165, 2008.
12. C. Versari, N. Busi, and R. Gorrieri. An expressiveness study of priority in process calculi. *Mathematical. Structures in Comp. Sci.*, 19:1161–1189, 2009.

Controlling Reversibility in Higher-Order Pi

Ivan Lanese[1], Claudio Antares Mezzina[2],
Alan Schmitt[2], and Jean-Bernard Stefani[2]

[1] University of Bologna & INRIA, Italy
[2] INRIA Grenoble-Rhône-Alpes, France

Abstract. We present in this paper a fine-grained rollback primitive for the higher-order π-calculus (HOπ), that builds on the reversibility apparatus of reversible HOπ [9]. The definition of a proper semantics for such a primitive is a surprisingly delicate matter because of the potential interferences between concurrent rollbacks. We define in this paper a high-level operational semantics which we prove sound and complete with respect to reversible HOπ backward reduction. We also define a lower-level distributed semantics, which is closer to an actual implementation of the rollback primitive, and we prove it to be fully abstract with respect to the high-level semantics.

1 Introduction

Motivation and contributions. Reversible computing, or related notions, can be found in many areas, including hardware design, program debugging, discrete-event simulation, biological modeling, and quantum computing (see [2] and the introduction of [10] for early surveys on reversible computing). Of particular interest is the application of reversibility to the study of programming abstractions for fault-tolerant systems. In particular, most fault tolerance schemes based on *system recovery* techniques [1], including rollback/recovery schemes and transaction abstractions, imply some form of undo. The ability to undo any single action in a reversible computation model provides an ideal setting to study, revisit, or imagine alternatives to these different schemes. This is in part the motivation behind the recent development of the reversible process calculi RCCS [4] and $\rho\pi$ [9], with [5] showing how a general notion of interactive transaction emerges from the introduction of irreversible (commit) actions in RCCS. However, these calculi provide very little in the way of controlling reversibility. The notion of irreversible action in RCCS only prevents a computation from rolling back past a certain point. Exploiting the low-level reversibility machinery available in these models of computation for fault-recovery purposes would require more extensive control on the reversal of actions, including *when* they can take place and *how far back* (along a past computation) they apply.

We present in this paper the study of a fine-grained rollback control primitive, where potentially every single step in a concurrent execution can be undone. Specifically, we introduce a rollback construct for an asynchronous higher-order

π-calculus (HOπ [11]), building on the machinery of $\rho\pi$, the reversible higher-order π-calculus presented in [9]. We chose HOπ as our substrate because we find it a convenient starting point for studying distributed programming models with inherently higher-order features such as dynamic code update, which we aim to combine with abstractions for system recovery and fault tolerance. Surprisingly, finding a suitable definition for a fine-grained rollback construct in HOπ is more difficult than one may think, even with the help of the reversible machinery from [9]. There are two main difficulties. The first one is in actually pinning down the intended effect of a rollback operation, especially in presence of concurrent rollbacks. The second one is in finding a suitably distributed semantics for rollback, dealing only with local information and not relying on complex atomic transitions involving a potentially unbounded number of distinct processes.

We show in this paper how to deal with these difficulties by making the following contributions: (i) we define a high-level operational semantics for a rollback construct in an asynchronous higher-order π-calculus, which we prove *maximally permissive*, in the sense that it makes reachable all past states in a given computation; (ii) we present a low-level semantics for the proposed rollback construct which can be understood as a fully distributed variant of our high-level semantics, and we prove it to be *fully abstract* with respect to the high-level one.

Paper Outline. In Section 2, we informally present our rollback calculus, which we call roll-π, and illustrate the difficulties that may arise in defining a fine-grained rollback primitive. In Section 3, we formalize roll-π and its high-level operational semantics. In Section 4, we present a distributed operational semantics for roll-π, and we prove that it is fully abstract with respect to the high-level one. Section 5 discusses related work and concludes the paper. The interested reader can find proofs of the main results in [8].

2 Informal presentation

To define roll-π and its rollback construct, we rely on the same support for reversibility as in $\rho\pi$ [9]. Let us review briefly its basic mechanisms.

Reversibility in $\rho\pi$. We attach to each process P a unique tag κ (either simple, written as k, or composite, denoted as $\langle h_i, \tilde{h}\rangle \cdot k$). The uniqueness of tags for processes is achieved thanks to the following structural congruence rule that defines how tags and parallel composition commute.

$$k : \prod_{i=1}^{n} \tau_i \equiv \nu \tilde{h}.\ \prod_{i=1}^{n} (\langle h_i, \tilde{h}\rangle \cdot k : \tau_i) \quad \text{with } \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2 \qquad (1)$$

In equation (1), $\prod_{i=1}^{n}$ is n-ary parallel composition and ν is the restriction operator, both standard from the π-calculus. Each *thread* τ_i is either a message, of the form $a\langle P\rangle$ (where a is a channel name), or a receiver process (also called a *trigger*),

of the form $a(X) \triangleright P$. A *forward* computation step (or forward reduction step, noted with arrow $\twoheadrightarrow$) consists of the reception of a message by a receiver process, and takes the following form (note that $\rho\pi$ is an asynchronous calculus).

$$(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q) \twoheadrightarrow \nu k.\, k : Q\{^P/_X\} \mid [M; k] \tag{2}$$

In this forward step, κ_1 identifies a thread consisting of message $a\langle P\rangle$ on channel a, and κ_2 identifies a thread consisting of a trigger process $a(X) \triangleright Q$ that expects a message on channel a. The result of the message input yields, as usual, an instance $Q\{^P/_X\}$ of the body of the trigger Q with the formal parameter X instantiated by the received value, i.e., the process P ($\rho\pi$ is higher-order). Message input also has two side effects: (i) the tagging of the newly created process $Q\{^P/_X\}$ by a fresh tag k, and (ii) the creation of a memory $[M; k]$, which records the original two threads, $M = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q)$, together with tag k.

In $\rho\pi$, a forward reduction step such as (2) above is systematically associated with a backward reduction step (noted with arrow $\rightsquigarrow$) of the form:

$$(k : Q) \mid [M; k] \rightsquigarrow M \tag{3}$$

which undoes the communication between threads κ_1 and κ_2. When necessary to avoid confusion, we will add a $\rho\pi$ subscript to arrows representing $\rho\pi$ reductions.

Given a configuration M, the set of memories present in M provides us with an ordering $:>$ between tags in M that reflects their causal dependency: if memory $[\kappa_1 : P_1 \mid \kappa_2 : P_2; k]$ occurs in M, then $\kappa_i > k$. Also, $k > \langle h_i, \tilde{h}\rangle \cdot k$, and we define the relation $:>$ as the reflexive and transitive closure of the $>$ relation. We say that tag κ has κ' as a causal antecedent if $\kappa' :> \kappa$.

Reversibility in roll-π. The notion of memory introduced in $\rho\pi$ is in some way a checkpoint, uniquely identified by its tag. In roll-π, we exploit this intuition to introduce an explicit form of backward reduction. Specifically, backward reduction is not allowed by default as in $\rho\pi$, but has to be triggered by an instruction of the form roll k, whose intent is that the current computation be rolled back to a state just prior to the creation of the memory bearing the tag k. To be able to form an instruction of the form roll k, one needs a way to pass the knowledge of a memory tag to a process. This is achieved in roll-π by adding a bound variable to each trigger process, which now takes the form $a(X) \triangleright_\gamma P$, where γ is the tag variable bound by the trigger construct and whose scope is P. A forward reduction step in roll-π therefore is:

$$(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow \nu k.\, k : Q\{^{P,k}/_{X,\gamma}\} \mid [M; k] \tag{4}$$

where the only difference with (2) lies in the fact that the newly created tag k is passed as an argument to the trigger body Q. We write $a(X) \triangleright P$ in place of $a(X) \triangleright_\gamma P$ if the tag variable γ does not appear free in P.

Now, given the above intent for the rollback primitive roll, how does one define its operational semantics? As hinted at in the introduction, this is actually a subtler affair than one may expect. A big difference with $\rho\pi$, where communication steps are undone one by one, is that the k in roll k may refer to a

communication step far in the past. So the idea behind a $\mathsf{roll}\ k$ is to restore the content of a memory $[M;k]$ and to delete all its forward history. Consider the following attempt at a rule for roll:

$$(\textsc{Naive}) \quad \frac{N \blacktriangleright k \qquad \mathtt{complete}(N \mid [M;k] \mid (\kappa : \mathsf{roll}\ k))}{N \mid [M;k] \mid (\kappa : \mathsf{roll}\ k) \rightsquigarrow M \mid N \lightning_k}$$

The different predicates and the $\lightning$ operator used in the rule are defined formally in the next section, but an informal explanation should be enough to understand how the rule works. Briefly, the assertion $N \blacktriangleright k$ states that all the active threads and memories in N bear tags κ that have k as causal antecedent, i.e., $k :> \kappa$ (N does not contain unrelated processes). The assertion $\mathtt{complete}(M_c)$ states that configuration M_c gathers all the threads (inside or outside memories) whose tags have as a causal antecedent the tag of a memory in M_c itself, i.e., if a memory in M_c is of the form $[M';k']$ (the communication M' created a process tagged with k'), then a process or a memory containing a process tagged with k' has to be in M_c (M_c contains every related process). The premises of rule Naive thus asserts that the configuration $M_c = N \mid [M;k] \mid \kappa : \mathsf{roll}\ k$, on the left hand side of the reduction in the conclusion of the rule, gathers all (and only) the threads and memories which have originated from the process tagged by k, itself created by the interaction of the message and trigger recorded in M. Being complete, M_c is thus ready to be rolled back and replaced by the configuration M which is at its origin. Rolling back M_c has another effect, noted as $N \lightning_k$ in the right hand side of the conclusion, which is to release from memories those messages or triggers which do not have k as a causal antecedent, but which participated in communications with causal descendants of k.

For instance, the configuration $M_0 = M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$, where $M_1 = (\kappa_0 : a\langle P\rangle) \mid (\kappa_1 : a(X) \triangleright_\gamma c\langle \mathsf{roll}\ \gamma\rangle)$, has the following forward reductions (where $M_2 = (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$):

$$\begin{aligned} M_0 &\twoheadrightarrow \nu k.\, [M_1;k] \mid (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y) \\ &\twoheadrightarrow \nu k, l.\, [M_1;k] \mid [M_2;l] \mid (l : \mathsf{roll}\ k) = M_3 \end{aligned}$$

Applying rule Naive (and structural congruence, defined later) on M_3 we get:

$$M_3 \rightsquigarrow M_1 \mid [M_2;l]\lightning_k = M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y) = M_0$$

where $(\kappa_2 : c(Y) \triangleright_\delta Y)$ is released from memory $[M_2;l]$ because it does not have k as a causal antecedent.

Rule Naive looks reasonable enough, but difficulties arise when concurrent rollbacks are taken into account. Consider the following configuration:

$$M = (k_1 : \tau_1) \mid (k_2 : a\langle \mathbf{0}\rangle) \mid (k_3 : \tau_3) \mid (k_4 : b\langle \mathbf{0}\rangle)$$

where[1] $\tau_1 = a(X) \triangleright_\gamma d\langle \mathbf{0}\rangle \mid (c(Y) \triangleright \mathsf{roll}\ \gamma)$ and $\tau_3 = b(Z) \triangleright_\delta c\langle \mathbf{0}\rangle \mid (d(U) \triangleright \mathsf{roll}\ \delta)$.

[1] We assume parallel composition has precedence over trigger.

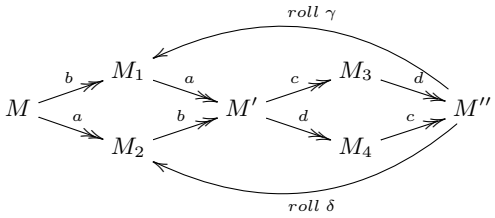


Fig. 1. Concurrent rollback anomaly

The most interesting reductions of M are depicted in Figure 1. Forward reductions are labelled by the name of the channel used for communication, while backward reductions are labelled by the executed roll instruction. The main processes and short-cuts are detailed below:

$$M_1 = \nu l_2, h_3, h_4.\, \sigma_1 \mid [\sigma_2; l_2] \mid (\kappa_3 : c\langle\mathbf{0}\rangle) \mid (\kappa_4 : \tau_4)$$
$$M_2 = \nu l_1, h_1, h_2.\, [\sigma_1; l_1] \mid (\kappa_1 : d\langle\mathbf{0}\rangle) \mid (\kappa_2 : \tau_2) \mid \sigma_2$$
$$M'' = \nu l_1 \ldots l_4, h_1 \ldots h_4.\, [\sigma_1; l_1] \mid [\sigma_2; l_2] \mid [\sigma_3; l_3] \mid [\sigma_4; l_4] \mid (l_3 : \mathsf{roll}\ l_1) \mid (l_4 : \mathsf{roll}\ l_2)$$

$$\sigma_1 = (k_1 : \tau_1) \mid (k_2 : a\langle\mathbf{0}\rangle) \quad \sigma_2 = (k_3 : \tau_3) \mid (k_4 : b\langle\mathbf{0}\rangle) \quad \tau_2 = c(Y) \triangleright \mathsf{roll}\ l_1$$
$$\sigma_3 = (\kappa_2 : \tau_2) \mid (\kappa_3 : c\langle\mathbf{0}\rangle) \quad \sigma_4 = (\kappa_1 : d\langle\mathbf{0}\rangle) \mid (\kappa_4 : \tau_4) \quad \tau_4 = d(U) \triangleright \mathsf{roll}\ l_2$$

The anomaly here is that there is no way from M_1 or M_2 to get back to the original configuration M, despite the fact that M'' has two roll instructions which would seem sufficient to undo all the reductions which lead from M to M''. Note that M_1 and M_2 are configurations which could both have been reached from M. Thus rule Naive is not unsound, but incomplete or insufficiently permissive, at least with respect to what is possible in $\rho\pi$: if we were to undo actions in M'' step by step, using $\rho\pi$'s backward reductions, we could definitely reach all of M, M_1, and M_2. Note that the higher-order aspects do not matter here.

The main motivation to have a complete rule comes from the fact that, in an abstract semantics, one wants to be as liberal as possible, and not unduly restrict implementations. If we were to pick the Naive rule as our semantics for rollback, then a correct implementation would have to enforce the same restrictions with respect to states reachable from backward reductions, restrictions which, in the case of rule Naive, are both complex to characterize (in terms of conflicting rollbacks) and quite artificial since they do not correspond to any clear execution policy. In the next section, we present a *maximally permissive* semantics for rollback, using $\rho\pi$ as our benchmark for completeness.

3 The roll-π calculus and its high-level semantics

3.1 Syntax

Names, keys, and variables. We assume the existence of the following denumerable infinite mutually disjoint sets: the set $\mathcal{N}$ of *names*, the set $\mathcal{K}$ of *keys*, the

$$
\begin{aligned}
P, Q &::= \mathbf{0} \mid X \mid \nu a.\, P \mid (P \mid Q) \mid a\langle P \rangle \mid a(X) \triangleright_\gamma P \mid \mathsf{roll}\ k \mid \mathsf{roll}\ \gamma \\
M, N &::= \mathbf{0} \mid \nu u.\, M \mid (M \mid N) \mid \kappa : P \mid [\mu; k] \mid [\mu; k]^\bullet \\
\kappa &::= k \mid \langle h, \tilde{h} \rangle \cdot k \\
\mu &::= ((\kappa_1 : a\langle P \rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)) \\
& a \in \mathcal{N} \quad X \in \mathcal{V}_\mathcal{P} \quad \gamma \in \mathcal{V}_\mathcal{K} \quad u \in \mathcal{I} \quad h, k \in \mathcal{K}
\end{aligned}
$$

Fig. 2. Syntax of roll-π

set $\mathcal{V}_\mathcal{K}$ of *tag variables*, and the set $\mathcal{V}_\mathcal{P}$ of process variables. The set $\mathcal{I} = \mathcal{N} \cup \mathcal{K}$ is called the set of *identifiers*. We note $\mathbb{N}$ the set of natural integers. We let (together with their decorated variants): a, b, c range over $\mathcal{N}$; h, k, l range over $\mathcal{K}$; u, v, w range over $\mathcal{I}$; δ, γ range over $\mathcal{V}_\mathcal{K}$; X, Y, Z range over $\mathcal{V}_\mathcal{P}$. We note $\tilde{u}$ a finite set of identifiers $\{u_1, \ldots, u_n\}$.

Syntax. The syntax of the roll-π calculus is given in Figure 2 (we often add balanced parenthesis around roll-π terms to disambiguate them). *Processes*, given by the P, Q productions in Figure 2, are the standard processes of the asynchronous higher-order π-calculus, except for the presence of the roll primitive and the extra bound tag variable in triggers. A trigger in roll-π takes the form $a(X) \triangleright_\gamma P$, which allows the receipt of a message of the form $a\langle Q \rangle$ on channel a, and the capture of the tag of the receipt event with tag variable γ.

Processes in roll-π cannot directly execute, only *configurations* can. *Configurations* in roll-π are given by the M, N productions in Figure 2. A configuration is built up from *tagged processes* and *memories*.

In a tagged process $\kappa : P$ the tag κ is either a single key k or a pair of the form $\langle h, \tilde{h} \rangle \cdot k$, where $\tilde{h}$ is a set of keys with $h \in \tilde{h}$. A tag serves as an identifier for a process. As in $\rho\pi$ [9], tags and memories help capture the flow of causality in a computation.

A *memory* is a configuration of the form $[\mu; k]$, which keeps track of the fact that a configuration μ was reached during execution, that triggered the launch of a process tagged with the fresh tag k. In a memory $[\mu; k]$, we call μ the *configuration part* of the memory, and k the *tag* of the memory. The configuration part $\mu = (\kappa_1 : a\langle P \rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)$ of a memory records the message $a\langle P \rangle$ and the trigger $a(X) \triangleright_\gamma Q$ involved in the message receipt, together with their respective thread tags κ_1, κ_2. A *marked memory* is a configuration of the form $[\mu; k]^\bullet$, which just serves to indicate that a rollback operation targeting this memory has been initiated.

We note $\mathcal{P}$ the set of roll-π processes, and $\mathcal{C}$ the set of roll-π configurations. We call *agent* an element of the set $\mathcal{A} = \mathcal{P} \cup \mathcal{C}$. We let (together with their decorated variants) P, Q, R range over $\mathcal{P}$; L, M, N range over $\mathcal{C}$; and A, B, C range over $\mathcal{A}$. We call *thread*, a process that is either a message $a\langle P \rangle$, a trigger $a(X) \triangleright_\gamma P$, or a rollback instruction roll k. We let τ and its decorated variants range over threads.

Free identifiers and free variables. Notions of free identifiers and free variables in roll-π are usual. Constructs with binders are of the following forms: $\nu a.\, P$ binds the name a with scope P; $\nu u.\, M$ binds the identifier u with scope M; and $a(X) \triangleright_\gamma P$ binds the process variable X and the tag variable γ with scope P. We note $\mathtt{fn}(P)$, $\mathtt{fn}(M)$, and $\mathtt{fn}(\kappa)$ the set of free names, free identifiers, and free keys, respectively, of process P, of configuration M, and of tag κ. Note in particular that $\mathtt{fn}(\kappa : P) = \mathtt{fn}(\kappa) \cup \mathtt{fn}(P)$, $\mathtt{fn}(\mathsf{roll}\ k) = \{k\}$, $\mathtt{fn}(k) = \{k\}$ and $\mathtt{fn}(\langle h, \tilde{h}\rangle \cdot k) = \tilde{h} \cup \{k\}$. We say that a process P or a configuration M is *closed* if it has no free (process or tag) variable. We note $\mathcal{P}^{cl}$, $\mathcal{C}^{cl}$ and $\mathcal{A}^{cl}$ the sets of closed processes, configurations, and agents, respectively.

Initial and consistent configurations. Not all configurations allowed by the syntax in Figure 2 are meaningful. For instance, in a memory $[\mu; k]$, tags occurring in the configuration part μ must be different from the key k; if a tagged process $\kappa_1 : \mathsf{roll}\ k$ occurs in a configuration M, we expect a memory $[\mu; k]$ to occur in M as well. In the rest of the paper, we only will be considering well-formed, or *consistent*, closed configurations. A configuration is consistent if it can be derived using the rules of the calculus from an *initial* configuration. A configuration is initial if it does not contain memories, all the tags are distinct and simple (i.e., of the form k), and the argument of each roll is bound by a trigger.

We do not give here a syntactic characterization of consistent configurations as it is not essential to understand the developments in this paper (the interested reader may find some more details in [9], where a syntactic characterization of $\rho\pi$ consistent configurations is provided).

Remark 1. We have no construct for replicated processes or guarded choice in roll-π: as in HOπ, these can easily be encoded.

Remark 2. In the remainder of the paper, we adopt *Barendregt's Variable Convention*: if terms $t_1, \ldots, t_n$ occur in a certain context (e.g., definition, proof), then in these terms all bound identifiers and variables are chosen to be different from the free ones.

3.2 Operational semantics

The operational semantics of the roll-π calculus is defined via a reduction relation $\rightarrow$, which is a binary relation over closed configurations ($\rightarrow \subset \mathcal{C}^{cl} \times \mathcal{C}^{cl}$), and a structural congruence relation $\equiv$, which is a binary relation over processes and configurations ($\equiv \subset \mathcal{P}^2 \cup \mathcal{C}^2$). We define *evaluation contexts* as "configurations with a hole $\cdot$", given by the following grammar:

$$\mathbb{E} ::= \cdot \mid (M \mid \mathbb{E}) \mid \nu u.\, \mathbb{E}$$

General contexts $\mathbb{C}$ are just processes or configurations with a hole $\cdot$. A *congruence* on processes or configurations is an equivalence relation $\mathcal{R}$ that is closed for general contexts: $P\,\mathcal{R}\,Q \implies \mathbb{C}[P]\,\mathcal{R}\,\mathbb{C}[Q]$ or $M\,\mathcal{R}\,N \implies \mathbb{C}[M]\,\mathcal{R}\,\mathbb{C}[N]$.

The relation $\equiv$ is defined as the smallest congruence on processes and configurations that satisfies the rules in Figure 3. We note $t =_\alpha t'$ when terms

$$\text{(E.ParC)}\ A \mid B \equiv B \mid A \qquad \text{(E.ParA)}\ A \mid (B \mid C) \equiv (A \mid B) \mid C$$

$$\text{(E.ParN)}\ A \mid \mathbf{0} \equiv A \qquad \text{(E.NewN)}\ \nu u.\, \mathbf{0} \equiv \mathbf{0} \qquad \text{(E.NewC)}\ \nu u.\, \nu v.\, A \equiv \nu v.\, \nu u.\, A$$

$$\text{(E.NewP)}\ (\nu u.\, A) \mid B \equiv \nu u.\, (A \mid B) \qquad \text{(E.}\alpha\text{)}\ A =_\alpha B \implies A \equiv B$$

$$\text{(E.TagN)}\ \kappa : \nu a.\, P \equiv \nu a.\, \kappa : P$$

$$\text{(E.TagP)}\ k : \prod_{i=1}^{n} \tau_i \equiv \nu \tilde{h}.\, \prod_{i=1}^{n} (\langle h_i, \tilde{h} \rangle \cdot k : \tau_i) \quad \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2$$

Fig. 3. Structural congruence for roll-π

t, t' are equal modulo α-conversion. If $\tilde{u} = \{u_1, \ldots, u_n\}$, then $\nu\tilde{u}.\, A$ stands for $\nu u_1.\, \ldots \nu u_n.\, A$. We note $\prod_{i=1}^{n} A_i$ for $A_1 \mid \ldots \mid A_n$ (there is no need to indicate how the latter expression is parenthesized because the parallel operator is associative by rule E.ParA). In rule E.TagP, processes τ_i are threads. Recall the use of the variable convention in these rules: for instance, in the rule $(\nu u.\, A) \mid B \equiv \nu u.\, (A \mid B)$ the variable convention makes implicit the condition $u \notin \mathtt{fn}(B)$. The structural congruence rules are the usual rules for the π-calculus (E.ParC to E.α) without the rule dealing with replication, and with the addition of two new rules dealing with tags: E.TagN and E.TagP. Rule E.TagN is a scope extrusion rule to push restrictions to the top level. Rule E.TagP allows to generate unique tags for each thread in a configuration. An easy induction on the structure of terms provides us with a kind of normal form for configurations (by convention $\prod_{i \in I} A_i = \mathbf{0}$ if $I = \emptyset$, and $[\mu; k]^\circ$ stands for $[\mu; k]$ or $[\mu; k]^\bullet$):

Lemma 1 (Thread normal form). *For any configuration M, we have*

$$M \equiv \nu\tilde{u}.\, \prod_{i \in I} (\kappa_i : \rho_i) \mid \prod_{j \in J} [\mu_j; k_j]^\circ$$

with $\rho_i = \mathbf{0}$, $\rho_i = \mathsf{roll}\ k_i$, $\rho_i = a_i\langle P_i \rangle$, or $\rho_i = a_i(X_i) \triangleright_{\gamma_i} P_i$.

We say that a binary relation $\mathcal{R}$ on closed configurations is *evaluation-closed* if it satisfies the inference rules:

$$\text{(R.Ctx)}\ \frac{M\ \mathcal{R}\ N}{\mathbb{E}[M]\ \mathcal{R}\ \mathbb{E}[N]} \qquad \text{(R.Eqv)}\ \frac{M \equiv M' \qquad M'\ \mathcal{R}\ N' \qquad N' \equiv N}{M\ \mathcal{R}\ N}$$

The reduction relation $\to$ is defined as the union of two relations, the *forward* reduction relation $\twoheadrightarrow$ and the *backward* reduction relation $\rightsquigarrow$: $\to\ =\ \twoheadrightarrow \cup \rightsquigarrow$. Relations $\twoheadrightarrow$ and $\rightsquigarrow$ are defined to be the smallest evaluation-closed binary relations on closed configurations satisfying the rules in Figure 4 (note again the use of the variable convention: in rule H.Com the key k is fresh).

The rule for forward reduction H.Com is the standard communication rule of the higher-order π-calculus with three side effects: (i) the creation of a new

$$\text{(H.Com)}\quad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow \nu k.\, (k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(H.Start)}\quad (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^\bullet$$

$$\text{(H.Roll)}\quad \frac{N \blacktriangleright k \qquad \texttt{complete}(N \mid [\mu; k])}{N \mid [\mu; k]^\bullet \rightsquigarrow \mu \mid N \lightning k}$$

Fig. 4. Reduction rules for roll-π

memory to record the configuration that gave rise to it; (ii) the tagging of the continuation of the message receipt with the fresh key k; (iii) the passing of the newly created tag k as a parameter to the newly launched instance of the trigger's body Q.

Backward reduction is subject to the rules H.Roll and H.Start. Rule H.Roll is similar to rule Naive defined in the previous section, except that it relies on the presence of a marked memory instead of on the presence of the process $\kappa : \mathsf{roll}\ k$ to roll back a given configuration. Rule H.Start just marks a memory to enable rollback.

The definition of rule H.Roll exploits several predicates and relations which we define below.

Definition 1 (Causal dependence). *Let M be a configuration and let T_M be the set of tags occurring in M. The binary relation $>_M$ on T_M is defined as the smallest relation satisfying the following clauses:*

- $k >_M \langle h_i, \tilde{h}\rangle \cdot k$;
- $\kappa' >_M k$ *if* κ' *occurs in* μ *for some memory* $[\mu; k]^\circ$ *that occurs in* M.

The causal dependence relation $:>_M$ is the reflexive and transitive closure of $>_M$.

Relation $\kappa :>_M \kappa'$ reads "κ is a causal antecedent of κ' according to M". When configuration M is clear from the context, we write $\kappa :> \kappa'$ for $\kappa :>_M \kappa'$.

Definition 2 (κ dependence). *Let $M \equiv \nu\tilde{u}.\, \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J}[\mu_j; \kappa_j]^\circ$. Configuration M is κ-dependent, written $M \blacktriangleright \kappa$, if $\forall i \in I \cup J,\ \kappa :>_M \kappa_i$.*

We now define the *projection* operation on configurations $M \lightning \kappa$, that captures the parallel composition of all tagged processes that do not depend on κ occurring in memories in M.

Definition 3 (Projection). *Let $M \equiv \nu\tilde{u}.\, \prod_{i\in I}(\kappa_i : \rho_i) \mid \prod_{j\in J}[\mu_j; \kappa_j]^\circ$, with $\mu_j = \kappa'_j : R_j \mid \kappa''_j : T_j$. Then:*

$$M \lightning \kappa = \nu\tilde{u}.\, (\prod_{j'\in J'} \kappa'_{j'} : R_{j'}) \mid (\prod_{j''\in J''} \kappa''_{j''} : T_{j''})$$

where $J' = \{j \in J \mid \kappa :\not> \kappa'_j\}$ and $J'' = \{j \in J \mid \kappa :\not> \kappa''_j\}$.

Finally we define the notion of *complete* configuration, used in the premise of rule H.Roll.

Definition 4 (Complete configuration). *A configuration M* contains *a tagged process $\kappa : P$, written $\kappa : P \in M$, if $M \equiv \nu\tilde{u}.\,(\kappa : P) \mid N$ or $M \equiv \nu\tilde{u}.\,[\kappa : P \mid \kappa_1 : Q; k]^\circ \mid N$.*

A configuration M is complete*, noted* $\texttt{complete}(M)$*, if for each memory $[\mu; k]^\circ$ that occurs in M, one of the following holds:*

1. *There exists a process P such that $k : P \in M$.*
2. *There is $\tilde{h}$ such that for each $h_i \in \tilde{h}$ there exists a process P_i such that $\langle h_i, \tilde{h}\rangle \cdot k : P_i \in M$.*

Barbed bisimulation. The operational semantics of the roll-π calculus is completed classically by the definition of a contextual equivalence between configurations, which takes the form of a barbed congruence. We first define observables in configurations. We say that name a is *observable in configuration M*, noted $M \downarrow_a$, if $M \equiv \nu\tilde{u}.\,(\kappa : a\langle P\rangle) \mid N$, with $a \notin \tilde{u}$. Keys are not observable: this is because they are just an internal device used to support reversibility. We note $\Rightarrow$, $\twoheadrightarrow^*$, $\rightsquigarrow^*$ the reflexive and transitive closures of $\rightarrow$, $\twoheadrightarrow$, and $\rightsquigarrow$, respectively.

One of the aims of this paper is to define a low-level semantics for roll-π, and show that it is equivalent to the high-level one. We want to use weak barbed congruence for this purpose. Thus we need a definition of barbed congruence able to relate roll-π configurations executed under different semantics. These semantics will also rely on different runtime syntaxes. Thus, we define a family of relations, each labeled by the semantics to be used on the left and right components of its elements. We also label sets of configurations with the corresponding semantics, thus highlighting that the corresponding runtime syntax has to be included. However, contexts do not include runtime syntax, since we never add contexts at runtime.

Definition 5 (Barbed bisimulation and congruence). *A relation ${}_{s1}\mathcal{R}_{s2} \subseteq \mathcal{C}^{cl}_{s1} \times \mathcal{C}^{cl}_{s2}$ on closed consistent configurations is a* strong *(resp.* weak*)* barbed simulation *if whenever $M\ {}_{s1}\mathcal{R}_{s2}\ N$*

- *$M \downarrow_a$ implies $N \downarrow_a$ (resp. $N \Rightarrow_{s2}\downarrow_a$)*
- *$M \rightarrow_{s1} M'$ implies $N \rightarrow_{s2} N'$, with $M'\ {}_{s1}\mathcal{R}_{s2} N'$ (resp. $N \Rightarrow_{s2} N'$ with $M'\ {}_{s1}\mathcal{R}_{s2} N'$)*

A relation ${}_{s1}\mathcal{R}_{s2} \subseteq \mathcal{C}^{cl}_{s1} \times \mathcal{C}^{cl}_{s2}$ is a strong *(resp.* weak*)* barbed bisimulation *if ${}_{s1}\mathcal{R}_{s2}$ and $({}_{s1}\mathcal{R}_{s2})^{-1}$ are strong (resp. weak) barbed simulations. We call* strong *(resp.* weak*)* barbed bisimilarity *and note ${}_{s1}\sim_{s2}$ (resp. ${}_{s1}\approx_{s2}$) the largest strong (resp. weak) barbed bisimulation with respect to semantics $s1$ and $s2$.*

We say that two configurations M and N are strong *(resp.* weak*)* barbed congruent*, written ${}_{s1}\sim^c_{s2}$ (resp. ${}_{s1}\approx^c_{s2}$), if for each roll-π context $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are consistent, then $\mathbb{C}[M]\ {}_{s1}\sim_{s2}\ \mathbb{C}[N]$ (resp. $\mathbb{C}[M]\ {}_{s1}\approx_{s2}\ \mathbb{C}[N]$).*

3.3 Soundness and completeness of backward reduction in roll-π

We present in this section a Loop Theorem, that establishes the soundness of backward reduction in roll-π, and we prove the completeness (or maximal permissiveness) of backward reduction in roll-π.

Theorem 1 (Loop Theorem - Soundness of backward reduction). *For any (consistent) configurations M and M' with no marked memories, if $M \rightsquigarrow^* M'$, then $M' \twoheadrightarrow^* M$.*

To state the completeness result for backward reduction in roll-π, we define a family of functions $\phi_e : \mathcal{C}_{\text{roll-}\pi} \to \mathcal{C}_{\rho\pi}$, where $e \in \mathcal{N}$, mapping a roll-π configuration to a $\rho\pi$ configuration. Function ϕ_e is defined by induction as follows:

$$\begin{array}{ccc}
\phi_e(\nu u.\, A) = \nu u.\, \phi_e(A) & \phi_e(A \mid B) = \phi_e(A) \mid \phi_e(B) & \phi_e(\kappa : P) = \kappa : \phi_e(P) \\
\phi_e([\mu; k]^\circ) = [\phi_e(\mu); k] & \phi_e(\mathbf{0}) = \mathbf{0} & \phi_e(X) = X \\
\phi_e(\mathsf{roll}\ k) = e\langle \mathbf{0} \rangle & \phi_e(\mathsf{roll}\ \gamma) = e\langle \mathbf{0} \rangle & \phi_e(a\langle P \rangle) = a\langle \phi_e(P) \rangle \\
 & \phi_e(a(X) \triangleright_\gamma P) = a(X) \triangleright \phi_e(P) &
\end{array}$$

Note that roll instructions are transformed not into $\mathbf{0}$ but into a thread $e\langle \mathbf{0} \rangle$: this is to ensure a consistent roll-π configuration is transformed into a consistent $\rho\pi$ configuration (recall that $\mathbf{0}$ is not a thread, thus it may be collected by structural congruence and there would be no thread corresponding to the roll k process).

We now state that roll-π is maximally permissive: any subset of roll primitives in evaluation context may successfully be executed, unlike in the naive example of Section 2. Let $M = \nu \tilde{u}.\, [\mu; k] \mid (k : P) \mid N$ be a $\rho\pi$ configuration and $S = \{k_1, \ldots, k_n\}$ a set of keys. We note $M \rightsquigarrow_S M'$ if $M \rightsquigarrow_{\rho\pi} M'$, $M' = \nu \tilde{u}.\, \mu \mid N$, and $k_i :> k$ for some $k_i \in S$ (here k is the key of the memory $[\mu; k]$ consumed by the reduction). If $M' \not\rightsquigarrow_S$, we say that M' is *final with respect to* S. We note $\rightsquigarrow_S^*$ the reflexive and transitive closure of $\rightsquigarrow_S$. We assume here that reductions are name-preserving, i.e., existing keys are not α-converted (cf. [9] for a discussion on the topic).

Theorem 2 (Completeness of backward reduction). *Let M be a (consistent) roll-π configuration such that $M \equiv \nu \tilde{u}.\, \prod_{i=1}^n \kappa_i : \mathsf{roll}\ k_i \mid M_1$, let $S = \{k_1, \ldots, k_n\}$, and let $e \in \mathcal{N} \setminus \mathtt{fn}(M)$. Then for all $T \subseteq S$, if $\phi_e(M) \rightsquigarrow_T^* N$ and N is final with respect to T, there exists M' such that $N = \phi_e(M')$, and $M \rightsquigarrow_{\text{roll-}\pi}^* M'$.*

4 A distributed semantics for roll-π

The semantics defined in the previous section captures the behavior of rollback, but its H.Roll rule specifies an atomic action involving a configuration with an unbounded number of processes and relies on global checks on this configuration, for verifying that it is complete and κ-dependent. This makes it arduous to implement, especially in a distributed setting.

$$\text{(L.Com)}\quad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow_{LL} \nu k.\, (k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(L.Start)}\quad (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow_{LL} (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^\bullet \mid \mathsf{rl}\ k$$

$$\text{(L.Span)}\quad \mathsf{rl}\ \kappa_1 \mid [\kappa_1 : P \mid M; k]^\circ \rightsquigarrow_{LL} [\lfloor \kappa_1 : P \rfloor \mid M; k]^\circ \mid \mathsf{rl}\ k$$

$$\text{(L.Branch)}\quad \frac{\langle h_i, \tilde{h}\rangle \cdot k \text{ occurs in } M}{\mathsf{rl}\ k \mid M \rightsquigarrow_{LL} \prod_{h_i \in \tilde{h}} \mathsf{rl}\ \langle h_i, \tilde{h}\rangle \cdot k \mid M}$$

$$\text{(L.Up)}\quad \mathsf{rl}\ \kappa_1 \mid (\kappa_1 : P) \rightsquigarrow_{LL} \lfloor \kappa_1 : P \rfloor \qquad \text{(L.Stop)}\quad [\mu; k]^\circ \mid \lfloor k : P \rfloor \rightsquigarrow_{LL} \mu$$

Fig. 5. Reduction rules for LL

$$\text{(E.Gb1)}\quad \nu k.\, \mathsf{rl}\ k \equiv_{LL} 0 \qquad \text{(E.Gb2)}\quad \nu k.\, \prod_{h_i \in \tilde{h}} \mathsf{rl}\ \langle h_i, \tilde{h}\rangle \cdot k \equiv_{LL} 0$$

$$\text{(E.TagPfr)}\ \lfloor k : \prod_{i=1}^{n} \tau_i \rfloor \equiv_{LL} \nu \tilde{h}.\, \prod_{i=1}^{n} \lfloor (\langle h_i, \tilde{h}\rangle \cdot k : \tau_i) \rfloor \quad \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2$$

Fig. 6. Additional structural laws for LL

We thus present in this section a low-level (written LL) semantics, where the conditions above are verified incrementally by relying on the exchange of rl notifications. We show that the LL semantics captures the same intuition as the one introduced in Section 3 by proving that given a (consistent) configuration, its behaviors under the two semantics are weak barbed congruent according to Definition 5.

To avoid confusion between the two semantics, we use a subscript LL to identify all the elements (reductions, structural congruence, ...) referred to the low-level semantics presented here, and HL (for high-level) for the semantics described in Section 3.

The LL semantics $\rightarrow_{LL}$ of roll-π is defined as for the HL one (cf. Section 3.2), as $\rightarrow_{LL} = \twoheadrightarrow_{LL} \cup \rightsquigarrow_{LL}$, where relations $\twoheadrightarrow_{LL}$ and $\rightsquigarrow_{LL}$ are defined to be the smallest evaluation-closed binary relations on closed LL configurations satisfying the rules in Figure 5. The notion of structural congruence used in the definition of evaluation-closed is here the smallest congruence on LL processes and configurations that satisfies the rules in Figure 3 and in Figure 6.

LL configurations differ from HL configurations in two aspects. First, tagged processes (inside or outside memories) can be frozen, denoted $\lfloor \kappa : P \rfloor$, to indicate that they are participating to a rollback (rollback is no longer atomic). Second,

LL configurations include notifications of the form $\mathsf{rl}\ \kappa$, used to notify a tagged process with key κ to enter a rollback.

Let us describe the LL rules. Communication rule L.Com is as before. The main idea for rollback is that when a memory pointed by a roll is marked (rule L.Start), a notification $\mathsf{rl}\ k$ is generated. This notification is propagated by rules L.Span and L.Branch. Rule L.Span also freezes threads inside memories, specifying that they will be eventually removed by the rollback. Rule L.Branch (where the predicate "κ occurs in M" means that either $M = \kappa : P$ or $M = [\mu; k']^\circ$ with $\kappa : P \in M$) is used when the target configuration has been split into multiple threads: a notification has to be sent to each of them. Rule L.Up is similar to L.Span, but it applies to tagged processes outside memories. It also stops the propagation of the rl notification. The main idea is that by using rules L.Span, L.Branch, and L.Up one is able to tag all the causal descendants of a marked memory. Finally, rule L.Stop rolls back a single computation step by removing a frozen process and freeing the content of the memory created with it. In the LL semantics a rollback request is thus executed incrementally, while it was atomic in the HL semantics (rule H.Roll). The LL semantics also exploits an extended structural congruence, adding axioms E.Gb1 and E.Gb2 to garbage collect rl notifications when they are no more needed, and extending axiom E.TagP to deal with frozen threads (axiom E.TagPFr).

We now show an example to clarify the semantics (each reduction is labeled by the name of the axiom used to derive it). Let $M_0 = M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$, where $M_1 = (\kappa_0 : a\langle P\rangle) \mid (\kappa_1 : a(X) \triangleright_\gamma c\langle \mathsf{roll}\ \gamma\rangle)$. We have:

$$
\begin{aligned}
M_0 &\twoheadrightarrow \nu k.\, [M_1; k] \mid (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y) \\
\text{(L.Com)} \quad &\twoheadrightarrow \nu k, l.\, [M_1; k] \mid [M_2; l] \mid (l : \mathsf{roll}\ k) \\
\text{(L.Start)} \quad &\rightsquigarrow \nu k, l.\, [M_1; k]^\bullet \mid [M_2; l] \mid (l : \mathsf{roll}\ k) \mid \mathsf{rl}\ k \\
\text{(L.Span)} \quad &\rightsquigarrow \nu k, l.\, [M_1; k]^\bullet \mid [M_2'; l] \mid (l : \mathsf{roll}\ k) \mid \mathsf{rl}\ l \\
\text{(L.Up)} \quad &\rightsquigarrow \nu k, l.\, [M_1; k]^\bullet \mid [M_2'; l] \mid \lfloor (l : \mathsf{roll}\ k) \rfloor \\
\text{(L.Stop)} \quad &\rightsquigarrow \nu k.\, [M_1; k]^\bullet \mid M_2' \\
\text{(L.Stop)} \quad &\rightsquigarrow M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y)
\end{aligned}
$$

where:

$$
M_2 = (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y) \quad M_2' = \lfloor (k : c\langle \mathsf{roll}\ k\rangle) \rfloor \mid (\kappa_2 : c(Y) \triangleright_\delta Y)
$$

One can see that the rollback operation starts with the application of the rule L.Start, whose effects are (i) to mark the memory aimed by a roll process, and (ii) to generate a notification $\mathsf{rl}\ k$ to freeze its continuation. Since the continuation of the memory $[M_1; k]$ is contained in the memory $[M_2; l]$ then the rule L.Span is applied. So, the part of the memory containing the tag k gets frozen and a freeze notification $\mathsf{rl}\ l$ is generated. The notification eventually reaches the process $l : \mathsf{roll}\ k$ and freezes it (rule L.Up). Now, since there exists a memory whose continuation is a frozen process, we can apply the rule L.Stop, and free the configuration part of the memory (M_2'). Again, we have that the continuation

of $[M_1; k]$ is a frozen process and by applying the rule L.Stop we can free the configuration M_1, obtaining the initial configuration. In general, a rollback of a step whose memory is tagged by k is performed by executing a top-down visit of its causal descendants, freezing them, followed by a bottom-up visit undoing the steps one at the time.

We can now state the correspondence result between the two semantics.

Theorem 3 (Correspondence between HL and LL). *For each roll-π HL consistent configuration M, $M \,{}_{HL}{\approx}^{c}_{LL}\, M$.*

Proof. The proof is quite long and technical, and relies on a several additional semantics used as intermediate steps from HL to LL. It can be found in [8]. □

This result can be easily formulated as full abstraction. In fact, the encoding j from HL configurations to LL configurations defined by the injection (HL configurations are a subset of LL configurations) is fully abstract.

Corollary 1 (Full abstraction). *Let j be the injection from HL (consistent) configurations to LL configurations and let M, N be two HL configurations. Then we have $j(M) \,{}_{LL}{\approx}^{c}_{LL}\, j(N)$ iff $M \,{}_{HL}{\approx}^{c}_{HL}\, N$.*

Proof. From Theorem 3 we have $M \,{}_{HL}{\approx}^{c}_{LL}\, j(M)$ and $N \,{}_{HL}{\approx}^{c}_{LL}\, j(N)$. The thesis follows by transitivity. □

The results above ensure that the loss of atomicity in rollback preserves the reachability of configurations yet does not make undesired configurations reachable.

5 Related work and conclusion

We have introduced in this paper a fine-grained undo capability for the asynchronous higher-order π-calculus, in the form of a rollback primitive. We present a simple but non-trivial high-level semantics for rollback, and we prove it both sound (rolling back brings a concurrent program back to a state that is a proper antecedent of the current one) and complete (rolling back can reach all antecedent states of the current one). We also present a lower-level distributed semantics for rollback, which we prove to be fully abstract with respect to the high-level one. The reversibility apparatus we exploit to support our rollback primitive is directly taken from our reversible HOπ calculus [9].

Undo or rollback capabilities in programming languages have been the subject of numerous previous works and we do not have the space to review them here; see [10] for an early survey in the sequential setting. Among the recent works that have considered undo or rollback capabilities for concurrent program execution, we can single out [3] where logging primitives are coupled with a notion of process group to serve as a basis for defining transaction abstractions, [12] which introduces a checkpoint abstraction for functional programs, and [7] which extends the actor model with constructs to create globally-consistent

checkpoints. Compared to these works, our rollback primitive brings immediate benefits: it provides a general semantics for undo operations which is not provided in [3]; thanks to the fine-grained causality tracking implied by our reversible substrate, our roll-π calculus does not suffer from uncontrolled cascading rollbacks (domino effect) which may arise with [12], and, in contrast to [7], provides a built-in guarantee that, in failure-free computations, rollback is always possible and reaches a consistent state (soundness of backward reduction).

Our low-level semantics for rollback, being a first refinement towards an implementation, is certainly related to distributed checkpoint and rollback schemes, in particular to the causal logging schemes discussed in the survey [6]. A thorough analysis of this relationship must be left for further study, however, as it requires a proper modeling of site and communication failures, as well as an explicit model for persistent data.

References

1. A. Avizienis, J.C. Laprie, B. Randell, and C.E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1), 2004.
2. C.H. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 32(1), 1988.
3. T. Chothia and D. Duggan. Abstractions for fault-tolerant global computing. *Theor. Comput. Sci.*, 322(3), 2004.
4. V. Danos and J. Krivine. Reversible communicating systems. In *Proc. of CONCUR'04*, volume 3170 of *LNCS*. Springer, 2004.
5. V. Danos and J. Krivine. Transactions in RCCS. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*. Springer, 2005.
6. E.N. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3), 2002.
7. J. Field and C.A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *Proc. of POPL'05*. ACM, 2005.
8. I. Lanese, C.A. Mezzina, A. Schmitt, and J.B. Stefani. Controlling reversibility in higher-order pi (TR). `http://www.cs.unibo.it/~lanese/publications/fulltext/TR-rollpi.pdf.gz`.
9. I. Lanese, C.A. Mezzina, and J.B. Stefani. Reversing higher-order pi. In *Proc. of CONCUR 2010*, volume 6269 of *LNCS*. Springer, 2010.
10. G.B. Leeman. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1), 1986.
11. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, University of Edinburgh, 1992.
12. L. Ziarek and S. Jagannathan. Lightweight checkpointing for concurrent ML. *J. Funct. Program.*, 20(2), 2010.

Termination in a π-calculus with Subtyping

Ioana Cristescu Daniel Hirschkoff
ENS Lyon, Université de Lyon, CNRS, INRIA, France

Abstract

We present a type system to guarantee termination of π-calculus processes that exploits input/output capabilities and subtyping, as originally introduced by Pierce and Sangiorgi, in order to analyse the usage of channels.

We show that our system improves over previously existing proposals by accepting more processes as terminating. This increased expressiveness allows us to capture sensible programming idioms. We demonstrate how our system can be extended to handle the encoding of the simply typed λ-calculus, and discuss questions related to type inference.

1 Introduction

Although many concurrent systems, such as servers, are supposed to run forever, termination is an important property in a concurrent setting. For instance, one would like a request to a server to be eventually answered; similarly, the access to a shared resource should be eventually granted. Termination can be useful to guarantee in turn lock-freedom properties [8].

In this work, we study termination in the setting of the π-calculus: concurrent systems are specified as π-calculus processes, and we would like to avoid situations in which a process can perform an infinite sequence of internal communication steps. Despite its conciseness, the π-calculus can express complex behaviours, such as reconfiguration of communication topology, and dynamic creation of channels and threads. Guaranteeing termination is thus a nontrivial task.

More specifically, we are interested in methods that provide termination guarantees statically. There exist several type-based approaches to guarantee termination in the π-calculus [5, 15, 12, 3, 4]. In these works, any typable process is guaranteed to be reactive, in the sense that it cannot

enter an infinite sequence of internal communications: it eventually terminates computation, or ends up in a state where an interaction with the environment is required.

The type systems in the works mentioned above have different expressive powers. Analysing the expressiveness of a type system for termination amounts to studying the class of processes that are recognised as terminating. A type system for termination typically rules out some terminating terms, because it is not able to recognise them as such (by essence, an effective type system for termination defines an approximation of this undecidable property). When improving expressiveness, one is interested in making the type system more flexible: more processes should be deemed as terminating. An important point in doing so is also to make sure that (at least some of) the 'extra processes' make sense from the point of view of programming.

Type systems for termination in the π-calculus. Existing type systems for termination in the π-calculus build on simple types [13], whereby the type of a channel describes what kind of values it can carry. Two approaches, that we shall call 'level-based' and 'semantics-based', have been studied to guarantee termination of processes. We discuss below the first kind of methods, and return to semantics-based approaches towards the end of this section. Level-based methods for the termination of processes originate in [5], and have been further analysed and developed in [3]. They exploit a stratification of names, obtained by associating a *level* (given by a natural number) to each name. Levels are used to insure that at every reduction step of a given process, some well-founded measure defined on processes decreases.

Let us illustrate the level-based approach on some examples. In this paper, we work in the asynchronous π-calculus, and replication can occur only on input prefixes. As in previous work, adding features like synchrony or the sum operator to our setting does not bring any difficulty.

According to level-based type systems, the process $!a(x).\overline{b}\langle x\rangle$ is well-typed provided $\mathsf{lvl}(a)$, the level of a, is strictly greater than $\mathsf{lvl}(b)$. Intuitively, this process trades messages on a (that 'cost' $\mathsf{lvl}(a)$) for messages on b (that cost less). Similarly, $!a(x).(\overline{b}\langle x\rangle \mid \overline{b}\langle x\rangle)$ is also well-typed, because none of the two messages emitted on b will be liable to trigger messages on a ad infinitum. More generally, for a process of the form $!a(x).P$ to be typable, we must check that all messages occurring in P are transmitted on channels whose level is strictly smaller than $\mathsf{lvl}(a)$ (more accurately, we only take into

account those outputs that do not occur under a replication in P — see Section 2).

This approach rules out a process like $!a(x).\overline{b}\langle x\rangle \mid !b(y).\overline{a}\langle y\rangle$ (which generates the unsatisfiable constraint $\mathsf{lvl}(a) > \mathsf{lvl}(b) > \mathsf{lvl}(a)$), as well as the other obviously 'dangerous' term $!a(x).\overline{a}\langle x\rangle$ — note that neither of these processes is diverging, but they lead to infinite computations as soon as they are put in parallel with a message on a.

The limitations of simple types. The starting point of this work is the observation that since existing level-based systems rely on simple types, they rule out processes that are harmless from the point of view of termination, essentially because in simple types, all names transmitted on a given channel should have the same type, and hence, in our setting, the same level as well.

If we try for instance to type the process $P_0 \stackrel{def}{=} !a(x).\overline{x}\langle t\rangle$, the constraint is $\mathsf{lvl}(a) > \mathsf{lvl}(x)$, in other words, the level of the names transmitted on a must be smaller than a's level. It should therefore be licit to put P_0 in parallel with $\overline{a}\langle p\rangle \,|\, \overline{a}\langle q\rangle$, provided $\mathsf{lvl}(p) < \mathsf{lvl}(a)$ and $\mathsf{lvl}(q) < \mathsf{lvl}(a)$. Existing type systems enforce that p and q have *the same type* for this process to be typable: as soon as two names are sent on the same channel (here, a), their types are unified. This means that if for some reason (for instance, if the subterm $!p(z).\overline{q}\langle z\rangle$ occurs in parallel) we must have $\mathsf{lvl}(p) > \mathsf{lvl}(q)$, the resulting process is rejected, although it is terminating.

We would like to provide more flexibility in the handling of the level of names, by relaxing the constraint that p and q from the example above should have the same type. To do this while preserving soundness of the type system, it is necessary to take into account the way names are used in the continuation of a replicated input. In process P_0 above, x is used in output in the continuation, which allows one to send on a any name (of the appropriate simple type) of level *strictly smaller* than $\mathsf{lvl}(a)$. If, on the other hand, we consider process $P_1 \stackrel{def}{=} !b(y).!y(z).\overline{c}\langle z\rangle$, then typability of the subterm $!y(z).\overline{c}\langle z\rangle$ imposes $\mathsf{lvl}(y) > \mathsf{lvl}(c)$, which means that any name of level *strictly greater* than $\mathsf{lvl}(c)$ can be sent on b. In this case, P_1 uses the name y that is received along b in input. We can remark that divergent behaviours would arise if we allowed the reception of names having a bigger (resp. smaller) level in P_0 (resp. P_1).

Contributions of this work. These observations lead us to introduce a new type system for termination of mobile processes based on Pierce and Sangiorgi's system for *input/output types* (i/o-types) [11]. I/o-types are

based on the notion of *capability* associated to a channel name, which makes it possible to grant only the possibility of emitting (the output capability) or receiving (the input capability) on a given channel. A subtyping relation is introduced to express the fact that a channel for which both capabilities are available can be coerced to a channel where only one is used. Intuitively, being able to have a more precise description of how a name will be used can help in asserting termination of a process: in P_0, only the output capability on x is used, which makes it possible to send a name of smaller level on a; in P_1, symmetrically, y can have a bigger level than expected, as only the input capability on y is transmitted.

The overall setting of this work is presented in Section 2, together with the definition of our type system. This system is strictly more expressive than previously existing level-based systems. We show in particular that our approach yields a form of 'level polymorphism', which can be interesting in terms of programming, by making it possible to send several requests to a given *server* (represented as a process of the form $!f(x).P$, which corresponds to the typical idiom for functions or servers in the π-calculus) with arguments that must have different levels, because of existing dependencies between them.

In order to study more precisely the possibility to handle terminating functions (or servers) in our setting, we analyse an encoding of the λ-calculus in the π-calculus. We have presented in [3] a counterexample showing that existing level-based approaches are not able to recognise as terminating the image of the simply-typed λ-calculus (STλ) in the π-calculus (all processes computed using such an encoding terminate [13]). We show that this counterexample is typable in our system, but we exhibit a new counterexample, which is not. This shows that despite the increased expressiveness, level-based methods for the termination of π-calculus processes fail to capture terminating sequential computation as expressed in STλ.

To accommodate functional computation, we exploit the work presented in [4], where an *impure* π-calculus is studied. Impure means here that one distinguishes between two kinds of names. On one hand, *functional names* are subject to a certain discipline in their usage, which intuitively arises from the way names are used in the encoding of STλ in the π-calculus. On the other hand, *imperative names* do not obey such conditions, and are called so because they may lead to forms of stateful computation (for instance, an input on a certain name is available at some point, but not later, or it is always available, but leads to different computations at different points in the execution).

In [4], termination is guaranteed in an impure π-calculus by using a level-based approach for imperative names, while functional names are dealt with separately, using a semantics-based approach [15, 12]. We show that that type system, which combines both approaches for termination in the π-calculus, can be revisited in our setting. We also demonstrate that the resulting system improves in terms of expressiveness over [4], from several points of view.

Several technical aspects in the definition of our type systems are new with respect to previous works. First of all, while the works we rely on for termination adopt a presentation à la Church, where every name has a given type a priori, we define our systems à la Curry, in order to follow the approach for i/o-types in [11]. As we discuss below, this has some consequences on the soundness proof of our systems. Another difference is in the presentation of the impure calculus: [4] uses a specific syntactical construction, called `def`, and akin to a `let .. in` construct, to handle functional names. By a refinement of i/o-types, we are able instead to enforce the discipline of functional names without resorting to a particular syntactical construct, which allows us to keep a uniform syntax.

We finally discuss type inference, by focusing on the case of the *localised π-calculus* (Lπ). Lπ corresponds to a certain restriction on i/o-types. This restriction is commonly adopted in implementations of the π-calculus. We describe a sound and complete type inference procedure for our level-based system in Lπ. We also provide some remarks about inference for i/o-types in the general case.

Paper outline. Section 2 presents our type system, and shows that it guarantees termination. We study its expressiveness in Section 3. Section 4 discusses type inference, and we give concluding remarks in Section 5. For lack of space, several proofs are omitted from this version of the paper.

2 A Type System for Termination with Subtyping

2.1 Definition of the Type System

Processes and types. We work with an infinite set of *names*, ranged over using $a, b, c, \ldots, x, y, \ldots$. Processes, ranged over using $P, Q, R, \ldots$, are defined by the following grammar ($\star$ is a constant, and we use v for values):

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

The constructs of restriction and (possibly replicated) input are binding, and give rise to the usual notion of α-conversion. We write $\mathrm{fn}(P)$ for the set of free names of process P, and $P[b/x]$ stands for the process obtained by applying the capture-avoiding substitution of x with b in P.

We moreover implicitly assume, in the remainder of the paper, that all the processes we manipulate are written in such a way that all bound names are pairwise distinct and are different from the free names. This may in particular involve some implicit renaming of processes when a reduction is performed.

The grammar of types is given by:

$$T \quad ::= \quad \sharp^k T \mid \mathsf{i}^k T \mid \mathsf{o}^k T \mid \mathbb{U} \ ,$$

where k is a natural number that we call a *level*, and $\mathbb{U}$ stands for the `unit` type having $\star$ as only value. A name having type $\sharp^k T$ has level k, and can be used to send or receive values of type T, while type $\mathsf{i}^k T$ (resp. $\mathsf{o}^k T$) corresponds to having only the input (resp. output) capability.

Figure 1 introduces the subtyping and typing relations. We note $\leq$ both for the subtyping relation and for the inequality between levels, as no ambiguity is possible. We can remark that the input (resp. output) capability is covariant (resp. contravariant) w.r.t. $\leq$, but that the opposite holds for levels: input requires the supertype to have a smaller level.

Γ ranges over typing environments, which are partial maps from names to types – we write $\Gamma(a) = T$ if Γ maps a to T. $\mathrm{dom}(\Gamma)$, the domain of Γ, is the set of names for which Γ is defined, and $\Gamma, a : T$ stands for the typing environment obtained by extending Γ with the mapping from a to T, this operation being defined only when $a \notin \mathrm{dom}(\Gamma)$.

The typing judgement for processes is of the form $\Gamma \vdash P : w$, where w is a natural number called the *weight* of P. The weight corresponds to an upper bound on the maximum level of a channel that is used in output in P, without this output occurring under a replication. This can be read from the typing rule for output messages (notice that in the first premise, we require the output capability on a, which may involve the use of subtyping) and for parallel composition. As can be seen by the corresponding rules, non replicated input prefix and restriction do not change the weight of a process. The weight is controlled in the rule for replicated inputs, where we require that the level of the name used in input is strictly bigger than the weight of the continuation process. We can also observe that working in a synchronous calculus would involve a minor change: typing a synchronous output $\overline{a}\langle v\rangle.P$ would be done essentially like typing $\overline{a}\langle v\rangle \mid P$ in our setting (with no major modification in the correctness proof for our type system).

As an abbreviation, we shall omit the content of messages in prefixes, and write a and $\overline{a}$ for $a(x)$ and $\overline{a}\langle\star\rangle$ respectively, when a's type indicates that a is used to transmit values of type $\mathbb{U}$.

Example 1 *The process* $!a(x).\overline{x}\langle t\rangle \mid \overline{a}\langle p\rangle \mid \overline{a}\langle q\rangle \mid !p(z).\overline{q}\langle z\rangle$ *from Section 1 can be typed in our type system: we can set* $a : \sharp^3\mathsf{o}^2T, p : \sharp^2T, q : \mathsf{o}^1T$. *Subtyping on levels is at work in order to typecheck the subterm* $\overline{a}\langle q\rangle$. *We provide a more complex term, which can be typed using similar ideas, in Example 15 below.*

Subtyping $\leq$ is the least relation that is reflexive, transitive, and satisfies the following rules:

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

Typing values

$$\frac{}{\Gamma \vdash \star : \mathbb{U}} \qquad \frac{\Gamma(a) = T}{\Gamma \vdash a : T} \qquad \frac{\Gamma \vdash a : T \qquad T \leq U}{\Gamma \vdash a : U}$$

Typing processes

$$\frac{}{\Gamma \vdash \mathbf{0} : 0} \qquad \frac{\Gamma \vdash a : \mathsf{o}^k T \qquad \Gamma \vdash v : T}{\Gamma \vdash \overline{a}\langle v\rangle : k}$$

$$\frac{\Gamma \vdash a : \mathsf{i}^k T \qquad \Gamma, x : T \vdash P : w}{\Gamma \vdash a(x).P : w}$$

$$\frac{\Gamma \vdash a : \mathsf{i}^k T \qquad \Gamma, x : T \vdash P : w \qquad k > w}{\Gamma \vdash !a(x).P : 0} \qquad \frac{\Gamma, a : T \vdash P : w}{\Gamma \vdash (\boldsymbol{\nu} a)\, P : w}$$

$$\frac{\Gamma \vdash P_1 : w_1 \qquad \Gamma \vdash P_2 : w_2}{\Gamma \vdash P_1 | P_2 : \max(w_1, w_2)}$$

Figure 1: Typing and Subtyping Rules

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

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

Figure 2: Reduction of Processes

Reduction and termination. The definition of the operational semantics relies on a relation of structural congruence, noted $\equiv$, which is the smallest equivalence relation that is a congruence, contains α-conversion, and satisfies the following axioms:

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

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

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

Note in particular that there is no structural congruence law for replication.

Reduction, written $\longrightarrow$, is defined by the rules of Figure 2.

Definition 2 (Termination) *A process P* diverges *if there exists an infinite sequence of processes $(P_i)_{i\geq 0}$ such that $P = P_0$ and for any i, $P_i \longrightarrow P_{i+1}$. P* terminates *(or P is terminating) if P does not diverge.*

2.2 Properties of the Type System

We first state some (mostly standard) technical properties satisfied by our system.

Lemma 3 *If $\Gamma \vdash P : w$ and $w \neq 0$ then for any $w' \geq w$, $\Gamma \vdash P : w'$.*

Proposition 4 (Narrowing) *If $\Gamma, x : T \vdash P : w$ and $T' \leq T$, then $\Gamma, x : T' \vdash P : w'$ for some $w' \leq w$.*

Lemma 5 *If $P \equiv Q$, then $\Gamma \vdash P : w$ iff $\Gamma \vdash Q : w$.*

Lemma 6 *If $\Gamma, x : T \vdash P : w$, $\Gamma \vdash b : T'$ and $T' \leq T$ then $\Gamma \vdash P[b/x] : w'$, for some $w' \leq w$.*

Proof (sketch). *This is a consequence of Lemma 4, as we replace x by a name of smaller type.*

Theorem 7 (Subject reduction) *If $\Gamma \vdash P : w$ and $P \longrightarrow P'$, then $\Gamma \vdash P' : w'$ for some $w' \leq w$.*

Proof (sketch). *By induction over the derivation of $P \longrightarrow P'$. The most interesting case corresponds to the case where $P = !a(x).P_1 \mid \overline{a}\langle v\rangle \longrightarrow P' = !a(x).P_1 \mid P_1[v/x]$. By typability of P, we have $\Gamma \vdash !a(x).P_1 : 0$. Let $T_a = \Gamma(a)$. Typability of P gives $\Gamma, x : T \vdash P_1 : w_1$ for some T and w_1 such that $T_a \leq \mathsf{i}^k T$ and $w_1 < k \leq \mathsf{lvl}(a)$. Typability of $\overline{a}\langle v\rangle$ gives $T_a \leq \mathsf{o}^{k'} U$ for some $k' \geq \mathsf{lvl}(a)$, with $w = k'$ and $\Gamma \vdash v : U$. The two constraints on T_a entail $T \leq U$, and hence, by Lemma 6, $\Gamma \vdash P_1[v/x] : w_2$ for some $w_2 \leq w_1 \leq \mathsf{lvl}(a) \leq k'$. We then conclude $\Gamma \vdash P' : w_2$.*

Termination. Soundness of our type system, that is, that every typable process terminates, is proved by defining a measure on processes that decreases at each reduction step. A typing judgement $\Gamma \vdash P : w$ yields the weight w of process P, but this notion is not sufficient (for instance, $\overline{a} \mid \overline{a} \mid a \longrightarrow \overline{a}$, and the weight is preserved). We instead adapt the approach of [1], and define the measure as a multiset of natural numbers. This is done by induction over the derivation of a typing judgement for the process. We will use $\mathcal{D}$ to range over typing derivations, and write $\mathcal{D} : \Gamma \vdash P : w$ to mean that $\mathcal{D}$ is a derivation of $\Gamma \vdash P : w$.

To deduce termination, we rely on the multiset extension of the well-founded order on natural numbers, that we write $>_{mul}$. $M_2 >_{mul} M_1$ holds if $M_1 = N \uplus N_1$, $M_2 = N \uplus N_2$, N being the maximal such multiset ($\uplus$ is multiset union), and for all $e_1 \in N_1$ there is $e_2 \in N_2$ such that $e_1 < e_2$. The relation $>_{mul}$ is well-founded. We write $M_1 \geq_{mul} M_2$ if $M_1 >_{mul} M_2$ or $M_1 = M_2$.

Definition 8 *Suppose $\mathcal{D} : \Gamma \vdash P : w$. We define a multiset of natural*

numbers, noted $\mathcal{M}(\mathcal{D})$*, by induction over* $\mathcal{D}$ *as follows:*

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

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

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

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

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

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

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

Note that in the case of output in the above definition, we refer to $\mathsf{lvl}(a)$, which is the level of a according to Γ (that is, without using subtyping). We have that if $\Gamma \vdash P : w$, then $\forall k \in \mathcal{M}_\Gamma(P), k \leq w$.

Lemma 9 *Suppose* $\Gamma \vdash P : w$*,* $\Gamma(x) = T$*,* $\Gamma(v) = T'$ *and* $T' \leq T$*. Then* $\mathcal{M}_\Gamma(P) \geq_{mul} \mathcal{M}_\Gamma(P[v/x])$*.*

Proof. *Follows from Lemma 6, and by definition of* $\mathcal{M}_\Gamma(\cdot)$*.* □

Lemma 10 *If* $\Gamma \vdash P : w$ *and* $P \equiv Q$*, then* $\Gamma \vdash Q : w'$ *for some* w' *and* $\mathcal{M}_\Gamma(P) = \mathcal{M}_\Gamma(Q)$*.*

We are now able to derive the essential property of $\mathcal{M}_\Gamma(\cdot)$:

Lemma 11 *If* $\Gamma \vdash P : w$ *and* $P \longrightarrow P'$*, then* $\mathcal{M}_\Gamma(P) >_{mul} \mathcal{M}_\Gamma(P')$*.*

Theorem 12 (Soundness) *If* $\Gamma \vdash P : w$*, then* P *terminates.*

Proof. *Suppose that* P *diverges, i.e., there is an infinite sequence*$(P_i)_{i \in N}$*, where* $P_i \longrightarrow P_{i+1}$*,* $P = P_0$*. According to Theorem 7 every* P_i *is typable. Using Lemma 11 we have* $\mathcal{M}_\Gamma(P_i) >_{mul} \mathcal{M}_\Gamma(P_{i+1})$ *for all* i*, which yields a contradiction.* □

Remark 13 (à la Curry vs à la Church) *Our system is presented à la Curry. Existing systems for termination [5, 4] are à la Church, while the usual presentations of i/o-types [11] are à la Curry. The latter style of presentation is better suited to address type inference (see Section 4). This has however some technical consequences in our proofs. Most importantly, the measure on processes (Definition 8) would be simpler when working à la Church, because we could avoid to consider all possible derivations of a given judgement. We are not aware of Church-style presentations of i/o-types.*

3 Expressiveness of our Type System

For the purpose of the discussions in this section, we work in a polyadic calculus. The extension of our type system to handle polyadicity is rather standard, and brings no particular difficulty.

3.1 A More Flexible Handling of Levels

Our system is strictly more expressive than the original one by Deng and Sangiorgi [5], as expressed by the two following observations (Lemma 14 and Example 15):

Lemma 14 *Any process typable according to the first type system of [5] is typable in our system.*

Proof. *The presentation of [5] differs slightly from ours. The first system presented in that paper can be recast in our setting by working with the # capability only (thus disallowing subtyping), and requiring type $\sharp^k T$ for a in the first premise of the rules for output, finite input and replicated input. We write $\Gamma \vdash_D P : w$ for the resulting judgement. We establish that $\Gamma \vdash_D P : w$ implies $\Gamma \vdash P : w$ by induction over the derivation of $\Gamma \vdash_D P : w$.* □

We now present an example showing that the flexibility brought by subtyping can be useful to ease programming. We view replicated processes as servers, or functions. Our example shows that it is possible in our system to invoke a server by passing names having different levels, provided some form of coherence (as expressed by the subtyping relation) is guaranteed. This form of "polymorphism on levels" is not available in previous type systems for termination in the π-calculus.

Example 15 (Level-polymorphism) *Consider the following definitions (in addition to polyadicity, we accommodate the first-order type of natural*

numbers, with corresponding primitive operations):

$$\begin{array}{rcl} F_1 & = & !f_1(n,r).\overline{r}\langle n*n\rangle \\ F_2 & = & !f_2(m,r).(\boldsymbol{\nu} s)\left(\overline{f}_1\langle m+1,s\rangle \mid s(x).\overline{r}\langle x+1\rangle\right) \\ Q & = & !g(p,x,r).(\boldsymbol{\nu} s)\left(\overline{p}\langle x,s\rangle \mid s(y).\overline{p}\langle y,r\rangle\right) \end{array}$$

F_1 is a server, running at f_1, that returns the square of a integer on a continuation channel r (which is its second argument). F_2 is a server that computes similarly $(m+1)^2+1$, by making a call to F_1 to compute $(m+1)^2$. Both F_1 and F_2 can be viewed as implementations of functions of type `int -> int`*.*

Q is a "higher-order server": its first argument p is the address of a server acting as a function of type `int -> int`*, and Q returns the result of calling twice the function located at p on its argument (process Q thus somehow acts like Church numeral 2).*

Let us now examine how we can typecheck the process

$$F_1 \mid F_2 \mid Q \mid \overline{g}\langle f_1, 4, t_1\rangle \mid \overline{g}\langle f_2, 5, t_2\rangle \ .$$

F_2 contains a call to f_1 under a replicated input on f_2, which forces $\mathsf{lvl}(f_2) > \mathsf{lvl}(f_1)$. In the type systems of [5], this prevents us from typing the processes above, since f_1 and f_2 should have the same type (and hence in particular the same level), both being used as argument in the outputs on g. We can type this process in our setting, thanks to subtyping, for instance by assigning the following types: $g : \mathsf{o}^{k_g}(\mathsf{o}^{k_2}T, U, V)$, $f_2 : \sharp^{k_2}T$, $f_1 : \sharp^{k_1}T$, with $k_1 < k_2$.

It can be shown that this example cannot be typed using any of the systems of [5]. It can however be phrased (and hence recognised as terminating) in the "purely functional π-calculus" of [4], that is, using a semantics-based approach — see also Section 3.3. It should however not be difficult to present a variation on it that forces one to rely on levels-based type systems.

3.2 Encoding the Simply-Typed λ-calculus

We now push further the investigation of the ability to analyse terminating functional behaviour in the π-calculus using our type system, and study an encoding of the λ-calculus in the π-calculus.

We focus on the following *parallel call-by-value* encoding, but we believe that the analogue of the results we present here also holds for other encodings. A λ-term M is encoded as $[\![M]\!]_p$, where p is a name which acts as a

parameter in the encoding. The encoding is defined as follows:

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

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

We can make the following remarks:

- A simply-typed λ-term is encoded into a simply-typed process (see [13]). Typability for termination comes into play in the translation of λ-abstractions.
- The target of this encoding is $L\pi$, the *localised π-calculus* in which only the output capability is transmitted (see also Section 4.1).

[3] provides a counterexample to typability of this encoding for the first type system of [5] (the proof of this result also entails that typability according to the other, more expressive, type systems due to Deng and Sangiorgi also fails to hold). Let us analyse this example:

Example 16 (From [3]) *The λ-term $M_1 \stackrel{def}{=} f\ (\lambda x.(f\ u\ (u\ v))$ can be typed in the simply typed λ-calculus, in a typing context containing the hypotheses $f : (\sigma \longrightarrow \tau) \longrightarrow \tau \longrightarrow \tau, v : \sigma, u : \sigma \longrightarrow \tau$.*

Computing $[\![M_1]\!]_p$ yields the process:

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

If we try and type this term using the first type system of [5], we can reason as follows:

1. *By looking at the line corresponding to $[\![f\ u]\!]_{q_1}$, we deduce that the types of f and f_2 are unified, and similarly for z_2 and u.*
2. *Similarly, the next line ($[\![u\ v]\!]_{r_1}$) implies that the types of f_3 and u are unified.*

3. *The last line above entails that the types assigned to f and f' must be unified, and the same for the types of z and y (because of the output $\overline{r}\langle y\rangle$).*

If we write $\sharp^k\langle T_1, T_2\rangle$ for the type (simple) assigned to f, we have by remark 1 that u has type T_1, and the same holds for y by remark 3. In order to typecheck the replicated term, we must have $\mathsf{lvl}(y) > \mathsf{lvl}(f_3) = \mathsf{lvl}(u)$ by remark 2, which is impossible since y and u have the same type.

While $[\![M_1]\!]_p$ cannot be typed using the approach of [5], it can be using the system of Section 2. Indeed, in that setting y and u need not have the same levels, so that we can satisfy the constraint $\mathsf{lvl}(y) > \mathsf{lvl}(u)$. The last line above generates an output $\overline{f}\langle y, p\rangle$, which can be typed directly, without use of subtyping. To typecheck the output $\overline{f}\langle u, q_1\rangle$, we "promote" the level of u to the level of y thanks to subtyping, which is possible because only the output capability on u is transmitted along f.

It however appears that our system is not able to typecheck the image of STλ, as the following (new) counterexample shows:

Example 17 *We first look at the following rather simple π-calculus process:*

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

This process is not typable in our type system, although it terminates. Indeed, we can assign a type of the form $\sharp^k \mathsf{o}^n\mathbb{U}$ to u, and $\sharp^m\mathbb{U}$ to v. Type-checking the subterm $!v.\overline{u}\langle t\rangle$ imposes $k < m$, and type-checking $!u(x).\overline{x}$ imposes $k > n$. Finally, type-checking $\overline{u}\langle v\rangle$ gives $m \leq n$, which leads to an inconsistency.

We can somehow 'expand' this process into the encoding of a λ-term: consider indeed

$$M_2 \quad \stackrel{def}{=} \quad \left(\,\lambda u.\,((\lambda v.(u\ v))\ (\lambda y.(u\ t)))\,\right)\ \ (\lambda x.\,(x\ a)) \ .$$

We do not present the (rather complex) process corresponding to $[\![M_2]\!]_p$. We instead remark that there is a sequence of reductions starting from $[\![M_2]\!]_p$ and leading to

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

These first reduction steps correspond to 'administrative reductions' (which have no counterpart in the original λ-calculus term). We can now perform

some communications that correspond to β-reductions, and obtain a process which contains a subterm of the form

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

Some channel names appear in boldface in order to stress the similarity with the process seen above: for the same reasons, this term cannot be typed. By subject reduction (Theorem 7), a typable term can only reduce to a typable term. This allows us to conclude that $[\![M_2]\!]_p$ is not typable in our system.

3.3 Subtyping and Functional Names

In order to handle functional computation as expressed by STλ, we extend the system of Section 2 along the lines of [4]. The idea is to classify names into *functional* and *imperative* names. Intuitively, functional names arise through the encoding of STλ. For termination, these are dealt with using an appropriate method — the 'semantics-based' approaches discussed in Section 1, and introduced in [15, 12]. For imperative names, we resort to (an adaptation of) the rules of Section 2.

Our type system is à la Curry, and the kind of a name, functional or imperative, is fixed along the construction of a typing derivation. Typing environments are of the form $\Gamma \bullet f : \mathsf{o}^k T$ — the intuition is that we isolate a particular name, f. f is the name which can be used to build replicated inputs where f is treated as a functional name. The typing rules are given on Figure 3. There are two rules to typecheck a restricted process, according to whether we want to treat the restricted name as functional (in which case the isolated name changes) or imperative (in which case the typing hypothesis is added to the Γ part of the typing environment).

The typing rules of Figure 3 rely on i/o-capabilities and the isolated name to enforce the usage of functional names as expressed in [12]. In [4], a specific syntactical construct is instead used: we manipulate processes of the form $\mathtt{def}\ f\ =\ (x)P_1\ \mathtt{in}\ P_2$ (that can be read as $(\boldsymbol{\nu} f)\,(!f(x).P_1 \mid P_2)$), where f does not occur in P_1 and occurs in output position only in P_2.

Let us analyse how our system imposes these constraints. In the rule for restriction on a functional name, the name g, that occurs in 'isolated position' in the conclusion of the rule, is added in the 'non isolated' part of the typing environment in the premise, with a type allowing one to use it in output only.

In the rules for input on an imperative name (replicated or not), the typing environment is of the form $\Gamma \bullet -$ in the premise where we typecheck

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

$$\frac{\Gamma, f : \mathsf{o}^k T \vdash a : \mathsf{o}^n U \qquad \Gamma, f : \mathsf{o}^k T \vdash v : U}{\Gamma \bullet f : \mathsf{o}^k T \vdash \overline{a}\langle v \rangle : n}$$

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

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

$$\frac{\Gamma \bullet f : \mathsf{o}^k T \vdash P_1 \qquad \Gamma \bullet f : \mathsf{o}^k T \vdash P_2}{\Gamma \bullet f : \mathsf{o}^k T \vdash P_1 | P_2} \qquad \frac{\Gamma, g : \mathsf{o}^k T \bullet f : \mathsf{o}^n U \vdash P : w}{\Gamma \bullet g : \mathsf{o}^k T \vdash (\boldsymbol{\nu} f)\, P : w}$$

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

Figure 3: Typing Rules for an Impure Calculus

the continuation process: this has to be understood as $\Gamma \bullet d : \mathsf{o}^k T$, for some dummy name d that is not used in the process being typed. We write '$-$' to stress the fact that we disallow the construction of replicated inputs on functional names. The functional name f appears in the aforementioned premise in the 'non isolated' part of the typing environment, with only the output rights on it. Forbidding the creation of replicated inputs on functional names under input prefixes is necessary because of diverging terms like the following (c is imperative, f is functional):

$$c(x).!f(y).\overline{x}\langle y \rangle \;\; | \; \overline{c}\langle f \rangle \; | \; \overline{f}\langle v \rangle \;\; .$$

Note also that typing non replicated inputs (on imperative names) involves the same constraints as for replicated inputs, like in [4]: the relaxed control over functional names requires indeed to be more restrictive on all usages of imperative names.

The notation $\Gamma \bullet -$ is also used in the rule to type a replicated input on a functional name, and we can notice that in this case f cannot be used at

all in the premise, to avoid recursion.

In addition to the gain in expressiveness brought by subtyping, we can make the following remark:

Remark 18 (Expressiveness) *As in [4], our system allows one to typecheck the encoding of a STλ term, by treating all names as functional, and assigning them level* 0.

Moreover, our type system makes it possible to typecheck processes where several replicated inputs on the same functional name coexist, provided they occur 'at the same level' in the term. For instance, a term of the form $(\boldsymbol{\nu} f)\,(!f(x).P \mid !f(y).Q \mid R)$ *can be well-typed with* f *acting as a functional name. This is not possible using the* `def` *construct of [4].*

Another form of expressiveness brought by our system is given by typability of the following process: $!u(x).\overline{x} \mid !v.\overline{u}\langle t\rangle \mid \overline{u}\langle v\rangle \mid c(y).\overline{u}\langle c\rangle$. Here, name c must be imperative while name v must be functional, and both are emitted on u. This is impossible in [4], where every channel carries either a functional or an imperative name. In our setting, only the output capability on c is transmitted along u, so in a sense c is transmitted 'as a functional name'.

Because of the particular handling of restrictions on functional names, the analogue of Lemma 5 does not hold for this type system: typability is not preserved by structural congruence. Accordingly, the subject reduction property is stated in the following way:

Theorem 19 (Subject reduction) *If* $\Gamma \bullet f : \mathsf{o}^k T \vdash P : w$ *and* $P \longrightarrow P'$, *then there exist* Q *and* $w' \leq w$ *s.t.* $P' \equiv Q$ *and* $\Gamma \bullet f : \mathsf{o}^k T \vdash Q : w'$.

Theorem 20 (Soundness) *If* $\Gamma \bullet f : \mathsf{o}^k T \vdash P : w$, *then* P *terminates.*

Proof (sketch). *The proof has the same structure as the corresponding proof in [4]. An important aspect of that proof is that we exploit the termination property for the calculus where all names are functional without looking into it. To handle the imperative part, we must adapt the proof along the lines of the termination argument for Theorem 12.*

4 Type Inference

We now study type inference, that is, given a process P, the existence of Γ, w such that $\Gamma \vdash P : w$. There might a priori be several such Γ (and several w: see Lemma 3). Type inference for level-based systems has been studied

in [2], in absence of i/o-types. We first present a type inference procedure in a special case of our type system, and then discuss this question in the general case.

4.1 Type Inference for Termination in the Localised π-calculus

In this section, we concentrate on the *localised π-calculus*, Lπ, which is defined by imposing that channels transmit only the output capability on names: a process like $a(x).x(y).\mathbf{0}$ does not belong to Lπ, as it makes use of the input capability on x. From the point of view of implementations, the restriction to Lπ makes sense. For instance, the language JoCaml [10] implements a variant of the π-calculus that follows this approach: one can only use a received name in output. Similarly, the communication primitives in Erlang [9] can also be viewed as obeying to the discipline of Lπ: asynchronous messages can be sent to a PiD (process id), and one cannot create dynamically a receiving agent at that PiD: the code for the receiver starts running as soon as the PiD is allocated.

Technically, Lπ is introduced by allowing the transmission of o-types only. We write $\Gamma \vdash^{L\pi} P : w$ if $\Gamma \vdash P : w$ can be derived in such a way that in the derivation, whenever a type of the form $\eta^k \eta'^{k'} T$ occurs, we have $\eta' = \mathsf{o}$ (types of the form $\mathsf{i}^k T$ and $\sharp^k T$ appear only when typechecking input prefixes and restrictions). Obviously, typability for $\vdash^{L\pi}$ entails typability for $\vdash$, hence termination. It can also be remarked that in restricting to Lπ, we keep an important aspect of the flexibility brought by our system. In particular, the examples we have discussed in Section 3 — Example 15, and the encoding of the λ-calculus — belong to Lπ.

We now describe a type inference procedure for $\vdash^{L\pi}$. For lack of space, we do not provide all details and proofs.

We first check typability when levels are not taken into account. For this, we rely on a type inference algorithm for simple types [14], together with a simple syntactical check to verify that no received name is used in input. When this first step succeeds, we replace $\sharp T$ types with $\mathsf{o}T$ types appropriately in the outcome of the procedure for simple types (a type variable may be assigned to some names, as, e.g., to name x in process $a(x).\bar{b}\langle x\rangle$).

What remains to be done is to find out whether types can be decorated with levels in order to ensure termination. As mentioned above, we suppose w.l.o.g. that we have a term P in which all bound names are pairwise distinct, and distinct from all free names. We define the following sets of names:

- names(P) stands for the set of all names, free and bound, of P;

- $\mathrm{bn}(P)$ is the set of names that appear bound (either by restriction or by input) in P;
- $\mathrm{rcv}(P)$ is the set of names that are bound by an input prefix in P ($x \in \mathrm{rcv}(P)$ iff P has a subterm of the form $a(x).Q$ or $!a(x).Q$ for some a, Q);
- $\mathrm{res}(P)$ stands for the set of names that are restricted in P ($a \in \mathrm{res}(P)$ iff P has a subterm of the form $(\boldsymbol{\nu} a)\, Q$ for some Q).

We have $\mathrm{bn}(P) = \mathrm{rcv}(P) \uplus \mathrm{res}(P)$ (where $\uplus$ stands for disjoint union), and $\mathrm{names}(P) = \mathrm{bn}(P) \uplus \mathrm{fn}(P)$. Moreover, for any $x \in \mathrm{rcv}(P)$, there exists a unique $a \in \mathrm{fn}(P) \cup \mathrm{res}(P)$ such that P contains the prefix $a(x)$ or the prefix $!a(x)$: we write in this case $a = \mathrm{father}(x)$ ($a \in \mathrm{fn}(P) \cup \mathrm{res}(P)$, because we are in L$\pi$).

We build a graph as follows:

- For every name $n \in \mathrm{fn}(P) \cup \mathrm{res}(P)$, create a node labelled by n, and create a node labelled by $\mathrm{son}(n)$. Intuitively, if n has type $\sharp^k S$ of $\mathsf{o}^k S$, $\mathrm{son}(n)$ has type S. In case type inference for simple types returns a type of the form α, where α is a type variable, for n, we just create the node n.
- For every $x \in \mathrm{rcv}(P)$, let $a = \mathrm{father}(x)$, add x as a label to $\mathrm{son}(a)$.

Example 21 *We associate to the process* $P = a(x).(\boldsymbol{\nu} b)\, \overline{x}\langle b\rangle \mid !a(y).(\overline{c}\langle y\rangle \mid d(z).\overline{y}\langle z\rangle)$ *the following set of 8 nodes with their labels:*

$\{a\}, \{\mathrm{son}(a), x, y\}, \{b\}, \{\mathrm{son}(b)\}, \{c\}, \{\mathrm{son}(c), y\}, \{d\}, \{\mathrm{son}(d), z\}$.

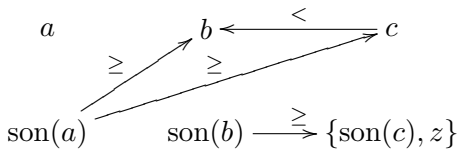
The next step is to insert edges in our graph, to represent the constraints between levels.

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

Example 22 *The graph associated to process* $!c(z).\overline{b}\langle z\rangle \mid \overline{a}\langle c\rangle \mid \overline{a}\langle b\rangle$ *has nodes*

$$\{a\}, \{\mathrm{son}(a)\}, \{b\}, \{\mathrm{son}(b)\}, \{c\}, \{\mathrm{son}(c), z\}\ ,$$

and can be depicted as follows:



The last phase of the type inference procedure consists in looking for an assignment of levels on the graph: this is possible as long as there are no cycles involving at least one $\xrightarrow{>}$ edge in the graph.

At the beginning, all nodes of the graph are unlabelled; we shall label them using natural numbers.

1. We go through all nodes of the graph, and collect those that have no outgoing edge leading to an unlabelled node in a set $\mathcal{S}$.

2. If $\mathcal{S}$ is not empty, we label every node n in $\mathcal{S}$ as follows: we start by setting n's label to 0.

   We then examine all outgoing edges of n. For every $n \xrightarrow{\geq} m$, we replace n's label, say k, with $\max(k, k')$, where k' is m's label, and similarly for $n \xrightarrow{>} m$ edges, with $\max(k, k' + 1)$.

   We then empty $\mathcal{S}$, and start again at step 1.

3. If $\mathcal{S} = \emptyset$, then either all nodes of the graph are labelled, in which case the procedure terminates, or the graph contains at least one oriented cycle. If this cycle contains at least one $\xrightarrow{>}$ edge, the procedure stops and reports failure. Otherwise, the cycle involves only $\xrightarrow{\geq}$ edges: we compute the level of each node of the cycle along the lines of step 2 (not taking into account nodes of the cycle among outgoing edges), and then assign the maximum of these labels to all nodes in the cycle. We start again at step 1.

This procedure terminates, since each time we go back to step 1, strictly more nodes are labelled.

Example 23 *On the graph of Example 22, the procedure first assigns level* 0 *to nodes* a, b *and* $\{\mathrm{son}(c), z\}$. *In the second iteration,* $\mathcal{S} = \{\mathrm{son}(b), c\}$*; level* 0 *is assigned to* $\mathrm{son}(b)$*, and* 1 *to* c. *Finally, level* 1 *is assigned to* $\mathrm{son}(a)$. *This yields the typing* $b : \mathsf{o}^0\mathsf{o}^0 T, c : \sharp^1\mathsf{o}^0 T, a : \mathsf{o}^0\mathsf{o}^1\mathsf{o}^0 T$ *for the process of Example 22.*

As announced above, for lack of space we have described only the main steps of our type inference procedure. Establishing that the latter has the

desired properties involves the introduction of an auxiliary typing judgement (that characterises $\vdash^{L\pi}$), and explaining how types are reconstructed at the end of the procedure. This finally leads to the following result:

Theorem 24 *There is a type inference procedure that given a process P, returns Γ, w s.t. $\Gamma \vdash^{L\pi} P : w$ iff there exists Γ', w' s.t. $\Gamma' \vdash^{L\pi} P : w'$.*

4.2 Discussion: Inferring i/o-Types

If we consider type inference for the whole system of Section 2, the situation is more complex. We start by discussing type inference without taking the levels into account. If a process is typable using simple types (that is, with only types of the form $\sharp T$), one is interested in providing a more informative typing derivation, where input and output capabilities are used.

For instance, the process $a(x).\overline{x}\langle t\rangle$ can be typed using different assignments for a: $\mathsf{io}T$, $\sharp\mathsf{o}T$, $\mathsf{i}\sharp T$, and $\sharp\sharp T$ — if we suppose $t : T$. Among these, $\mathsf{io}T$ is the most informative (intuitively, types featuring 'less #' seem preferable because they are more precise). Moreover, it is a supertype of all other types, thus acting as a 'candidate' if we were to look for a notion of principal typing. Actually, in order to infer i/o-types, one must be able to compute lubs and glbs of types, using equations like $glb(\mathsf{i}T, \mathsf{i}U) = \mathsf{i}\, glb(T, U)$, $glb(\mathsf{i}T, \mathsf{o}U) = \sharp glb(T, U)$, and $glb(\mathsf{o}T, \mathsf{o}U) = \mathsf{o}\, lub(T, U)$. The contravariance of o suggests the introduction of an additional capability, that we shall note $\uparrow$, which builds a supertype of input and output capabilities (more formally, we add the axioms $\mathsf{i}T \leq \uparrow T$ and $\mathsf{o}T \leq \uparrow T$).

[7] presents a type inference algorithm for (an enrichment of) i/o-types, where such a capability $\uparrow$ is added to the system of [11] (the notations are different, but we adapt them to our setting for the sake of readability). The use of $\uparrow$ can be illustrated on the following example process:

$$Q_1 \stackrel{def}{=} a(t).b(u).\big(\; !t(z).\overline{u}\langle z\rangle \mid \overline{c}\langle t\rangle \mid \overline{c}\langle u\rangle \;\big) \;.$$

To typecheck Q_1, we can see that the input (resp. output) capability on t (resp. u) needs to be received on a (resp. b), which suggests the types $a : \mathsf{ii}T, b : \mathsf{io}T$. Since t and u are emitted on the same channel c, and because of contravariance of output, we compute a *supertype* of $\mathsf{i}T$ and $\mathsf{o}T$, and assign type $\mathsf{o} \uparrow T$ to c.

Operationally, the meaning of $\uparrow$ is "no i/o-capability at all" (note that this does not prevent from comparing names, which may be useful to study behavioural equivalences [6]): in the typing we just described, since we only have the input capability on t and the output capability on u, we must

renounce to all capabilities, and t and u are sent without the receiver to be able to do anything with the name except passing it along. Observe also that depending on how the context uses c, a different typing can be introduced. For instance, Q_1 can be typed by setting $a : \mathsf{i}\sharp T, b : \mathsf{io}T, c : \mathsf{oo}T$. This typing means that the output capability on u is received, used, and transmitted on c, and both capabilities on t are received, the input capability being used locally, while the output capability is transmitted on c.

The first typing, which involves $\uparrow$, is the one that is computed by the procedure of [7]. It is "minimal", in the terminology of [7]. Depending on the situations, a typing like the second one (or the symmetrical case, where the input capability is transmitted on c) might be preferable.

If we take levels into account, and try and typecheck Q_1 (which contains a replicated subterm), the typings mentioned above can be adapted as follows: we can set $a : \mathsf{i}^0\sharp^1 T, b : \mathsf{i}^0\mathsf{o}^0 T, c : \mathsf{o}^0\mathsf{o}^1 T$, in which case subtyping on levels is used to deduce $u : \mathsf{o}^1 T$ in order to typecheck $\overline{c}\langle u\rangle$. Symmetrically, we can also set $a : \mathsf{i}^0\mathsf{i}^1 T, b : \mathsf{i}^0\sharp^0 T, c : \mathsf{o}^0\mathsf{i}^0 T$, and typecheck $\overline{c}\langle t\rangle$ using subsumption to deduce $t : \mathsf{i}^0 T$.

It is not clear to us how levels should be handled in relation with the $\uparrow$ capability. One could think that since $\uparrow$ prevents any capability to be used on a name, levels have no use, and one could simply adopt the subtyping axioms $\mathsf{i}^k T \leq \uparrow T$ and $\mathsf{o}^k T \leq \uparrow T$. This would indeed allow us to typecheck Q_1.

Further investigations on a system for i/o-types with $\uparrow$ and levels is left for future work, as well as the study of inference for such a system.

5 Concluding Remarks

In this paper, we have demonstrated how Pierce and Sangiorgi's i/o-types can be exploited to refine the analysis of the simplest of type systems for termination of processes presented in [5]. Other, more complex systems are presented in that work, and it would be interesting to study whether they would benefit from the enrichment with capabilities and subtyping. One could also probably refine the system of Section 2 by distinguishing between *linear* and *replicated input capabilities*, as only the latter must be controlled for termination (if a name is used in linear input only, its level is irrelevant).

The question of type inference for our type systems (differently from existing proposals, these are presented à la Curry, which is better suited for the study of type inference) can be studied further. It would be interesting to analyse how the procedure of Section 4.1 could be ported to programming

languages that obey the discipline of Lπ for communication, like Erlang or JoCaml. For the moment, we only have preliminary results for a type inference procedure for the system of Section 2, and we would like to explore this further. Type inference for the system of Section 3.3 is a challenging question, essentially because making the distinction between functional and imperative names belongs to the inference process (contrarily to the setting of [4], where the syntax of processes contains this information).

Acknowledgements. Romain Demangeon, as well as anonymous referees, have provided insightful comments and suggestions on this work. We also acknowledge support by ANR projects ANR-08-BLANC-0211-01 "COMPLICE", ANR-2010-BLANC-0305-02 "PiCoq" and CNRS PEPS "COGIP".

References

[1] R. Demangeon (2010): *Terminaison des systèmes concurrents*. Ph.D. thesis, ENS Lyon.

[2] R. Demangeon, D. Hirschkoff, N. Kobayashi & D. Sangiorgi (2007): *On the Complexity of Termination Inference for Processes*. In: *Proc. of TGC'07*, *LNCS* 4912, Springer, pp. 140–155, doi:10.1007/978-3-540-78663-4_11.

[3] R. Demangeon, D. Hirschkoff & D. Sangiorgi (2009): *Mobile Processes and Termination*. In: *Semantics and Algebraic Specification*, *LNCS* 5700, Springer, pp. 250–273, doi:10.1007/978-3-642-04164-8_13.

[4] R. Demangeon, D. Hirschkoff & D. Sangiorgi (2010): *Termination in Impure Concurrent Languages*. In: *Proc. of CONCUR'10*, *LNCS* 6269, Springer, pp. 328–342, doi:10.1007/978-3-642-15375-4_23.

[5] Y. Deng & D. Sangiorgi (2006): *Ensuring termination by typability*. *Inf. Comput.* 204(7), pp. 1045–1082, doi:10.1016/j.ic.2006.03.002.

[6] M. Hennessy & J. Rathke (2004): *Typed behavioural equivalences for processes in the presence of subtyping*. *Math. Str. in Comp. Sc.* 14(5), pp. 651–684, doi:10.1017/S0960129504004281.

[7] A. Igarashi & N. Kobayashi (2000): *Type Reconstruction for Linear -Calculus with I/O Subtyping*. *Inf. Comput.* 161(1), pp. 1–44, doi:10.1006/inco.2000.2872.

[8] N. Kobayashi & D. Sangiorgi (2010): *A hybrid type system for lock-freedom of mobile processes*. *ACM Trans. Program. Lang. Syst.* 32(5), doi:http://doi.acm.org/10.1145/1745312.1745313.

[9] Ericsson Computer Science Laboratory (2011): *Erlang Programming Language Website*. http://www.erlang.org.

[10] L. Mandel & L. Maranget (2010): *The JoCaml programming language*. http://jocaml.inria.fr/.

[11] B. C. Pierce & D. Sangiorgi (1996): *Typing and Subtyping for Mobile Processes*. *Math. Structures in Comput. Sci.* 6(5), pp. 409–453.

[12] D. Sangiorgi (2006): *Termination of Processes*. *Math. Structures in Comput. Sci* 16(1), pp. 1–39, doi:10.1017/S0960129505004810.

[13] D. Sangiorgi & D. Walker (2001): *The π-calculus: a Theory of Mobile Processes*. Cambridge Univ. Press.

[14] V. T. Vasconcelos & K. Honda (1993): *Principal Typing Schemes in a Polyadic pi-Calculus*. In: *Proc. of CONCUR'93*, *Lecture Notes in Computer Science* 715, Springer, pp. 524–538, doi:10.1007/3-540-57208-2_36.

[15] N. Yoshida, M. Berger & K. Honda (2004): *Strong Normalisation in the Pi-Calculus*. *Information and Computation* 191(2), pp. 145–202, doi:10.1016/j.ic.2003.08.004.

Strong Normalisation in λ-calculi with References

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

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

Abstract. We present a method for ensuring termination of lambda-calculi with references. This method makes it possible to combine term rewriting measure-based techniques for termination of imperative languages with traditional approaches to termination in purely functional languages, such as logical relations. More precisely, the method lifts any termination proof for the purely functional simply-typed lambda-calculus to a termination proof for the lambda-calculus with references. The method can be made parametric on the termination technique employed for the functional core.

1 Motivations

This paper studies strong normalisation in $\lambda_{\mathtt{ref}}$, a call-by-value λ-calculus with (higher-order) references. It is well-known that, even in the simply-typed calculus, the problem is difficult, because references allow one to program loops "via the memory". We refer to Boudol's [Bou07] for a discussion on existing works in this direction.

Boudol [Bou07] has proposed a type and effect system for a calculus whose core is very similar to $\lambda_{\mathtt{ref}}$; the system guarantees termination by means of the realisability technique. That work is revisited and generalised in [Ama09], where the closely related technique of reducibility candidates is exploited to establish soundness of the type and effect system. In both these works, the type and effect system relies on a stratification of memory into regions; the stratification is used to control interactions between the functional and the imperative constructs, in order to prevent "loops via the memory". The stratification plays also a key role in the structure of the soundness proof, to support the induction argument. Boudol's approach has also been investigated by Tranquilli [Tra10], who proposes an analysis of the stratification imposed by the type and effect system, by means of a monadic translation. The target of this translation, in the general case, is a lambda-calculus with recursive types. Tranquilli however shows that when applying the translation to well-typed source terms, one can avoid the use of recursive types. By combining this observation with a simulation result, the author concludes that well-typed terms terminate.

In this paper, we propose a different proof strategy for strong normalisation in $\lambda_{\mathtt{ref}}$. Our approach is adapted from [DHS10], where we introduced a type

system for termination of mobile processes. The crux in defining types in that work is to distinguish between functional and imperative channels, and to exploit a stratification of imperative channel names. Soundness of the type system is established by defining a projection of an *impure calculus*, that is, a calculus featuring imperative and functional features, into a purely functional core calculus (in the context of the π-calculus, the functional subcalculus is given, intuitively, by the image of the encoding of the λ-calculus in the π-calculus). The proof then relies on termination of the functional core, which is treated like a "black box" in the proof: since our projection function preserves divergences, and the target calculus is terminating, we can reason by contradiction to show that the source of the translation only consists of terminating terms.

In the present paper we show that we can transport the strategy from [DHS10] onto $\lambda_{\mathtt{ref}}$. In contrast with π-calculus, $\lambda_{\mathtt{ref}}$ is purely sequential and higher-order (it involves substitutions of variables with terms); both these features have a substantial impact on the details of the technique. In this sense, another goal of the paper is to show that the technique in [DHS10] is not specific to a concurrent scenario, and can be used on different kinds of impure languages. The "black box" property for the purely functional subcalculus in [DHS10] remains: the technique for $\lambda_{\mathtt{ref}}$ is essentially parametric on the method employed for ensuring termination of the pure λ-calculus (realisability, reducibility candidates or other methods). The present paper is devoted to the presentation of our technique in a rather simple setting, where the core functional language is the simply typed λ-calculus.

With respect to [DHS10], several modifications have to be made in order to handle $\lambda_{\mathtt{ref}}$. Many of them are related to the definition of the projection function, which in the present work maps λ-terms with references to purely functional terms. In the π-calculus the projection acts on prefixed terms, simply by replacing some of them with the inactive process $\mathbf{0}$; this crucially relies on the operators of parallel composition and $\mathbf{0}$ of the process calculus. In the λ-calculus the situation is more intricate. Consider for instance a $\lambda_{\mathtt{ref}}$ term of the form $T = (\lambda z.\star)\ (\mathtt{ref}\ M)$, where $\star$ is the unique element of type $\mathbb{1}$ (the unit type), and $\mathtt{ref}\ M$ denotes the allocation of a reference holding the value of M (the – slightly more involved – syntax and operational semantics of $\lambda_{\mathtt{ref}}$ will be introduced formally below). The idea is to project T into some purely functional term T', in such a way that: (i) if T is typable in our type and effect system, then T' is typable according to simple types; (ii) the projection function is defined compositionally on the structure of terms, and preserves divergences. In a call-by-value strategy, if the evaluation of M terminates, the evaluation of T yields $\star$. In order to preserve divergences, because of a potential divergence in M, we cannot define T' by simply erasing the subterm $\mathtt{ref}\ M$. Instead, we set $T' \stackrel{\text{def}}{=} (\lambda x_1.\lambda x_2.\, x_1)\ \star\ M'$, where M' is the purely functional term obtained by applying recursively the projection to M. This way, T' diverges if M' does so, and eventually returns $\star$, in case M' converges. This shows how the projection acts at an operational level. In the proof, we also take care of condition (i)

above, by defining the translation both on terms and on types, in such a way as to preserve typability.

Building on the projection function, we derive soundness of the type and effect system by contradiction: suppose a well-typed $\lambda_{\mathtt{ref}}$ term T diverges, then its projection T' is diverging too, which contradicts the fact that T' belongs to a terminating calculus. Termination of T' is obtained by an external argument, namely the strong normalisation proof for the functional subcalculus (here, the simply typed call by value λ-calculus).

Other technical differences with respect to the technique in the setting of the π-calculus [DHS10] are discussed later in the paper.

Comparison with [Bou07,Ama09,Tra10]. As we hinted above, the question we address in this paper has been studied in a very similar setting in other works. In contrast with the works by Boudol and Amadio, where soundness of the type system is obtained by a 'semantic' approach (be it realisability or reducibility candidates), which is applied to the whole (impure) calculus, we somehow factor out the imperative part of the calculus, which allows us to lift a termination proof of $\lambda_{\mathbf{ST}}$ to a termination proof of $\lambda_{\mathtt{ref}}$.

Tranquilli [Tra10] proceeds similarly, in two steps: a translation into a purely functional calculus, followed by a termination argument about the latter. However, technically, our approach and his differ considerably, in particular because we project into a subcalculus, using a translation function which seems unrelated to Tranquilli's.

Outline. We introduce $\lambda_{\mathtt{ref}}$ and its type and effect system in Section 2. Section 3 is devoted to the soundness proof, where we present in particular the projection function on $\lambda_{\mathtt{ref}}$. In Section 4, we discuss how the proof can be extended to calculi richer than simple types.

2 $\lambda_{\mathtt{ref}}$: a λ-calculus with References

2.1 Syntax and semantics for $\lambda_{\mathtt{ref}}$

We now define the calculi we manipulate in this work. The standard, simply-typed, λ-calculus with the constant $\star$ and the base type $\mathbb{1}$ is called $\lambda_{\mathbf{ST}}$ in the following. The reduction relation in $\lambda_{\mathbf{ST}}$ is full β-reduction, and is denoted using $\twoheadrightarrow$.

$\lambda_{\mathtt{ref}}$ is a call-by-value λ-calculus extended with imperative operations (read, write and update) acting on a store (sometimes called a *memory* in the following). The store is stratified into regions, which are referred to using natural numbers, i.e., we suppose that the store is divided into a finite number of regions, and that there exists an enumeration of these regions. Constructs of the language involving imperative operations are annotated by a region — thus, by a natural number. For instance, $\mathtt{deref}_n(M)$ is the operator that reads the value stored at the address which is returned by the evaluation of M; n denotes the fact that this address belongs to the region n of the memory.

$$
\begin{array}{rcl}
M & ::= & (M\ M) \mid x \mid \lambda x.\, M \mid \star \\
& & \mid \mathtt{ref}_n\ M \mid \mathtt{deref}_n(M) \mid M \mathtt{:=}_n M \mid u_{(n,T)} \\
T & ::= & \mathbb{1} \mid T\ \mathtt{ref}_n \mid T \to^n T \\
V & ::= & \lambda x.\, M \mid x \mid u_{(n,T)} \mid \star \\
R & ::= & (\lambda x.\, M)\ V \\
& & \mid \mathtt{deref}_n(u_{(n,T)}) \mid \mathtt{ref}_n\ V \mid u_{(n,T)} \mathtt{:=}_n V \\
\mathbf{E} & ::= & [\,] \mid V\ \mathbf{E} \mid \mathbf{E}\ M \\
& & \mid \mathtt{deref}_n(\mathbf{E}) \mid \mathtt{ref}_n\ \mathbf{E} \mid \mathbf{E} \mathtt{:=}_n M \mid V \mathtt{:=}_n \mathbf{E}
\end{array}
$$

Fig. 1. Syntax for terms, types, values, redexes and evaluation contexts

To define terms of $\lambda_{\mathtt{ref}}$, we rely on a set of *addresses*, which are distinct from the variables used in the syntax of the standard λ-calculus. Addresses are written $u_{(n,T)}$: they are explicitly associated both to a region n and to a type T (types are described below). These annotations are not mandatory in order to obtain the results we state in this paper, but they improve the readability of our proofs. Note in passing that values of different types can be stored in the same region. We suppose that there exists an infinite number of addresses for a given pair consisting of a type and a region.

Stores, ranged over using δ, are formally defined as partial mappings from addresses to values. The (finite) support of δ is written $\mathrm{supp}(\delta)$, $\emptyset$ is the empty store ($\mathrm{supp}(\emptyset) = \emptyset$), and $\delta\langle u_{(n,T)} \rightsquigarrow V\rangle$ denotes the store δ' defined by $\delta'(u_{(n,T)}) = V$ and $\delta'(v) = \delta(v)$ for every $v \in \mathrm{supp}(\delta)$ such that $v \neq u_{(n,T)}$.

Figure 1 presents the grammar definitions for (respectively) terms, types, values, redexes and evaluation contexts.

The standard λ-calculus syntax is extended with the unit value ($\star$), addresses and three imperative operators. $\mathtt{ref}_n\ M$ stands for the creation of a new cell in the store, at region n, and containing the result of the evaluation of M; $\mathtt{deref}_n(M)$ yields the value that is stored at the address given by the evaluation of M (in region n); finally, $M \mathtt{:=}_n N$ updates the value stored at the address given by the evaluation of M with the value of N.

Types extend the simple types of $\lambda_{\mathbf{ST}}$ with unit ($\mathbb{1}$) and a reference type: $T\ \mathtt{ref}_n$ is the type of an address in region n containing values of type T. To record the latent effect of a function, arrow types are annotated with regions: intuitively, $T_1 \to^n T_2$ is the type of a function taking arguments of type T_1, returning a term of type T_2, and such that evaluation of the body accesses regions in the memory *lower than* the region n.

Stratification. We impose a well-formedness condition on types that reflects the stratification of the store: a term acting at region n cannot be stored in a region smaller than $n + 1$. For this, we define $\mathrm{reg}(T)$, an integer describing the set of

regions associated to a type T, by:

$$\mathrm{reg}(\mathbb{1}) = 0 \qquad \mathrm{reg}(T\ \mathtt{ref}_n) = \max(n, \mathrm{reg}(T))$$
$$\mathrm{reg}(T_1 \to^n T_2) = \max(n, \mathrm{reg}(T_2))$$

Definition 1 (Well-formed types) *A type T is well-formed if for all its subtypes of the form $T'\ \mathtt{ref}_n$, we have $\mathrm{reg}(T') < n$.*

In the following, we shall implicitly assume that all types we manipulate are well-formed. Well-formedness of types is the condition that ensures the termination of the imperative part of a term. This in particular ensures that each time we reduce a redex $\mathtt{deref}_n(u_{(n,T)})$, the obtained value does not create new operations acting at region n.

Comparison with [Bou07]. The type system we present in the next section is actually very close to the one given in [Bou07], which in turn is close to the one of [Ama09].

In our presentation, regions, defined in [Bou07] as abstract parts of the store, are denoted by natural numbers. The two presentations are equivalent. In [Bou07], when the stratification condition (which is inductively defined on sets of regions) is met, a partial order between regions can be extracted, and thus integers can be assigned to regions so that each typable term can be given a well-formed type using our definitions. Conversely, from a set of regions indexed by natural numbers we can derive easily a set of corresponding abstract regions satisfying the stratification condition.

Another difference between the two settings is that our well-formedness condition for types is actually looser than the one found in [Bou07], allowing us to typecheck more terms. Indeed, in Definition 1, in the case of an arrow type, we do not impose the well-formedness condition in the type of the argument, making terms like $(\lambda x.(\mathtt{deref}_2(x)\ u_{(3,\mathbb{1})}))\ (\mathtt{ref}_2\ \lambda y.\star)$ acceptable in our setting, while they are not in [Bou07]. In this example, x has type $\mathbb{1}\ \mathtt{ref}_3 \to^0 \mathbb{1}$ (detailed typing rules can be found in the following section), which gives type $(\mathbb{1}\ \mathtt{ref}_3 \to^0 \mathbb{1})\ \mathtt{ref}_2$ for $\mathtt{ref}_2\ \lambda y.\star$. Note that this example is phrased using natural numbers for regions: it is not difficult to translate it into Boudol's framework, and insert the term in an appropriate context in order to enforce that the (abstract) region corresponding to 3 dominates the region corresponding to 2.

We think that the works [Bou07] and [Ama09] can easily be adapted with this small refinement in our definition of well-formedness in order to obtain the same expressiveness as our system.

2.2 Types and Reduction

Typing. Figure 2 defines two typing judgements, of the form $\Gamma \vdash M : (T, n)$ for terms and $\Gamma \vdash \delta$ for stores. Our type system is presented *à la Church*, and we write $\Gamma(x) = T$ when variable x has type T according to type environment Γ.

Typing rules for terms

$$(\textbf{App})\ \frac{\Gamma \vdash M : (T_1 \to^n T_2, m) \qquad \Gamma \vdash N : (T_1, k)}{\Gamma \vdash M\ N : (T_2, \max(m,n,k))}$$

$$(\textbf{Abs})\ \frac{\Gamma \vdash M : (T_2, n) \qquad \Gamma(x) = T_1}{\Gamma \vdash \lambda x.\, M : (T_1 \to^n T_2, 0)}$$

$$(\textbf{Ref})\ \frac{\Gamma \vdash M : (T_1, m)}{\Gamma \vdash \mathtt{ref}_n\ M : (T_1\ \mathtt{ref}_n, \max(n,m))} \qquad (\textbf{Var})\ \frac{\Gamma(x) = T_1}{\Gamma \vdash x : (T_1, 0)}$$

$$(\textbf{Uni})\frac{}{\Gamma \vdash \star : (\mathbb{1}, 0)} \qquad (\textbf{Add})\ \frac{}{\Gamma \vdash u_{(n,T_1)} : (T_1\ \mathtt{ref}_n, 0)}$$

$$(\textbf{Asg})\ \frac{\Gamma \vdash M : (T_1\ \mathtt{ref}_n, m) \qquad \Gamma \vdash N : (T_1, k)}{\Gamma \vdash M \mathtt{:=}_n N : (\mathbb{1}, \max(m,n,k))}$$

$$(\textbf{Drf})\ \frac{\Gamma \vdash M : (T\ \mathtt{ref}_n, m)}{\Gamma \vdash \mathtt{deref}_n(M) : (T, \max(m,n))}$$

Typing rules for stores

$$(\textbf{Emp})\frac{}{\Gamma \vdash \emptyset} \qquad (\textbf{Sto})\frac{\Gamma \vdash \delta \qquad \Gamma \vdash V : (T, 0)}{\Gamma \vdash \delta\langle u_{(n,T)} \rightsquigarrow V\rangle}$$

Fig. 2. $\lambda_{\mathtt{ref}}$: Type and Effect System

In a typing judgement $\Gamma \vdash M : (T, n)$, n defines a bound on the *effect* of the evaluation of M, which intuitively corresponds to the highest region accessed when evaluating M. Effects can be thought of as sets of regions (the part of the store manipulated by the evaluation of a term), and are denoted by a single natural number, which stands for the maximum region in the effect.

As explained above, in type $T_1 \to^n T_2$, n refers to the effect of the body of the function. As a consequence, in rule (**App**), the effect of the application $M\ N$ where M has type $T_1 \to^n T_2$ is the maximum between the effect of M, the effect of N, and n. Indeed the maximum region accessed during the evaluation of $M\ N$ is accessed during either the evaluation of M to some function $\lambda x.M_2$, or the evaluation of N to some value V_1, or during the evaluation of $M_2\{V_1/x\}$, whose effect is n.

We notice that values have an effect 0: values cannot reduce and, as explained above, the effect of a term stands for the maximum region accessed during its evaluation.

We extend typing to evaluation contexts by treating the hole as a term variable which can be given any type and has effect 0.

$$(\beta)\frac{}{(\lambda x.\, M\ V, \delta) \mapsto (M\{V/x\}, \delta)}$$

$$(\mathbf{ref})\frac{u_{(n,T)} \notin \mathrm{supp}(\delta) \qquad \Gamma \vdash V : (T, \_)}{(\mathtt{ref}_n\ V, \delta) \mapsto (u_{(n,T)},\ \delta\langle u_{(n,T)} \rightsquigarrow V\rangle)}$$

$$(\mathbf{deref})\frac{\delta(u_{(n,T)}) = V}{(\mathtt{deref}_n(u_{(n,T)}), \delta) \mapsto (V, \delta)}$$

$$(\mathbf{store})\frac{\Gamma \vdash V : (T, \_)}{(u_{(n,T)} \mathtt{:=}_n V, (\delta)) \mapsto (\star,\ \delta\langle u_{(n,T)} \rightsquigarrow V\rangle)}$$

$$(\mathbf{context})\frac{(M, \delta) \mapsto (M', \delta')}{(\mathbf{E}[M], \delta) \mapsto (\mathbf{E}[M'], \delta')}$$

Fig. 3. $\lambda_{\mathtt{ref}}$: Reduction Rules

Reduction. The execution of programs is given by a reduction relation, written $\mapsto$, relating *states* (a state is given by a pair consisting of a term and a store), and which is defined on Figure 3. We write $\mapsto^n_{\mathrm{F}}$ for a *functional* reduction, obtained using rule (β); n refers to the effect of the β-redex, that is, in this call-by-value setting, the region that decorates the type of the function being triggered. In other words, we suppose in rule (β) that $\Gamma \vdash \lambda x.\, M : (T_V \to^n T, m)$ holds for some T_V, T, m. We introduce similarly *imperative* reductions, noted $\mapsto^n_{\mathrm{I}}$, for reductions obtained using rules (**ref**), (**deref**) or (**store**) (in these cases, the accessed region n appears explicitly in the rules of Figure 3). We will call a reduction according to $\mapsto^n_{\mathrm{F}}$ (resp. $\mapsto^n_{\mathrm{I}}$) "a functional reduction on level n" (resp. "an imperative reduction on level n").

Definition 2 *We define an* infinite computation starting from M *as an infinite sequence* $(M_i, \delta_i)_{0 \le i}$ *such that* $M_0 = M$, $\delta_0 = \emptyset$ *and* $\forall i, (M_i, \delta_i) \mapsto (M_{i+1}, \delta_{i+1})$.

We say that a term M diverges *when there exists an infinite sequence starting from* M *and that* M terminates *when it does not diverge.*

The following result will be useful to prove Proposition 5. It says that we can replace a term inside an evaluation context with a term of the same type but with a smaller effect and preserve typability. The effect of the whole term can decrease (in the case where $\mathbf{E} = [\,]$ for instance).

Lemma 3 *If* $\begin{cases} \Gamma \vdash \mathbf{E}[M] : (T, n) \\ \Gamma \vdash M : (T_0, m) \\ \Gamma \vdash M' : (T_0, m') \\ m' \le m \end{cases}$ *then* $\Gamma \vdash \mathbf{E}[M'] : (T, n')$ *with* $n' \le n$.

Our type and effect system enjoys the two standard properties of subject substitution and subject reduction. Notice that in the statement of Lemma 4, the

effect associated to $M\{V/x\}$ is the same as the one associated to M. This holds as the term V is a value and thus does not introduce new operations on the memory which are not handled by the type system. Should we have used a call-by-name setting, the statement of this proposition would have been: "If $\Gamma \vdash M : (T, n)$, $\Gamma(x) = T'$ and $\Gamma \vdash N : (T', m)$ then $\Gamma \vdash M\{N/x\} : (T, \max(m, n))$".

Lemma 4 (Subject substitution)
If $\Gamma \vdash M : (T, n)$, $\Gamma(x) = T'$ and $\Gamma \vdash V : (T', m)$ then $\Gamma \vdash M\{V/x\} : (T, n)$.

We only sketch proofs for some results. The proof for Lemma 4, as well as detailed proofs for all other results, can be found in [Dem10].

Proposition 5 (Subject reduction)
$\Gamma \vdash M : (T, n)$, $\Gamma \vdash \delta$ and $(M, \delta) \mapsto (M', \delta')$ entail that $\Gamma \vdash \delta'$ and $\Gamma \vdash M' : (T, n')$ for some $n' \leq n$.

Proof (Sketch). The proof is done by induction on the derivation of $(M, \delta) \mapsto (M', \delta')$. If the rule (**context**) is used, we rely on Lemma 3. If the rule (**beta**) is used, we use the subject substitution. Cases (**ref**) and (**store**) are easy. Case (**deref**) is done using the hypothesis that δ is well-typed.

3 Termination of $\lambda_{\mathtt{ref}}$ Programs

3.1 Defining a projection from $\lambda_{\mathtt{ref}}$ to $\lambda_{\mathbf{ST}}$

The technique of projection and simulation works as follows. First, we define a projection function, parametrised upon a region p (we will refer to a "projection on level p"), which strips a $\lambda_{\mathtt{ref}}$ term from its imperative constructs (and some of its functional parts), in order to obtain a $\lambda_{\mathbf{ST}}$ term.

Then, we prove a simulation result (Lemma 14 below), stating that when a well-typed state (M, δ) reduces to (M', δ') by a functional reduction on level p, the projection on level p of M reduces in at least one step to the projection on level p of M' and when (M, δ) reduces to (M', δ') by another type of reduction then either the projections on level p of M and M' are equal, or the projection of M reduces in at least one step to the projection of M'. This result is what makes the projection function divergence preserving, as announced in Section 1.

With these results at hand, we suppose, toward a contradiction, the existence of a diverging process M_0, and we show the existence of a region p such that an infinite computation starting from M_0 contains an infinite number of functional reductions on level p. Using the simulation lemma, we obtain by projection a diverging $\lambda_{\mathbf{ST}}$ term (as a functional reduction on level p is mapped to at least one step of reduction), which contradicts strong normalisation of $\lambda_{\mathbf{ST}}$.

Before turning to the formal definition of the projection function, let us explain informally how it acts on $\mathtt{deref}_n(M)$ — we already gave some ideas about the projection of $\mathtt{ref}_n\ M$ in Section 1. Again, the purpose of the projection is to remove the imperative command. Because we cannot just throw away M

(this would invalidate the simulation lemma), we apply the projection function recursively to M. Once the projected version of M is executed, we replace the result with a value of the appropriate type, which we call a *generic value*.

More precisely, generic values are canonical terms that are used to replace a given subterm *once we know that no divergence can arise due to the evaluation of the subterm* (this would correspond either to a divergence of the subterm, or to a contribution to a more general divergence). They are defined as follows:

Definition 6 *Given a type T without the* `ref` *construct, the* generic value $\mathtt{V}_T$ of type T *is defined by:* $\mathtt{V}_{T\ \mathtt{ref}_n} = \mathtt{V}_{\mathbb{1}} = \star$, *and* $\mathtt{V}_{T_1 \to^n T_2} = \lambda x.\mathtt{V}_{T_2}$ *(x being of type T_1).*

In order to program the evaluation of a projected subterm and its replacement with a generic value, the definition of projection makes use of the following (families of) projectors:

$$\Pi^{(1,2)} = \lambda x.\lambda y.\, x \qquad\qquad \Pi^{(1,3)} = \lambda x.\lambda y.\lambda z.\, x \ .$$

In the following, we shall use these projectors in a well-typed fashion (that is, we pick the appropriate instance in the corresponding family).

Finally, in order to present the definition of the projection function, we need a last notion, that conveys the intuition that a given term M can be involved in a reduction on level p. This can be the case for two reasons. Either M is able to perform (maybe after some preliminary reduction steps) a reduction on level p, in which case, by the typing rules, the effect of M is greater than p, or M is a function that can receive some arguments and eventually perform a reduction on level p, in which case the type system ensures that its type T satisfies $\mathrm{reg}(T) \geq p$.

Definition 7 *Suppose $\Gamma \vdash M : (T, n)$. We say that M is* related to p *if either $n \geq p$ or $\mathrm{reg}(T) \geq p$. In the former (resp. latter) case, we say that M is related to p via its effect (resp. via its type).*

We extend this notion to evaluation contexts by treating the hole like a term variable, for a given typing derivation for a context (this is useful in particular in the statement of Lemma 13).

Notice that a term containing a subterm whose effect is p is not necessarily related to p: for instance, we can derive $\Gamma \vdash (\lambda x.\star)\ \lambda y.\mathtt{deref}_3(u_{(3,\mathbb{1})}) : (\mathbb{1}, 0)$ for an appropriate Γ, but this term is not related to 3 — one can easily check that this term cannot be used to trigger a reduction on level 3.

Definition 8 *Given a typable M of type T, we define the projection on level p of M, written $\mathsf{pr}^p_\Gamma(M)$, as follows:*

$$
\begin{array}{rl}
& \textit{If } M \textit{ is not related to } p\textit{:} \\
\mathsf{pr}^p_\Gamma(M) & = \mathtt{V}_T \\
\textit{Otherwise:} & \\
\mathsf{pr}^p_\Gamma(M_1\ M_2) & = \mathsf{pr}^p_\Gamma(M_1)\ \mathsf{pr}^p_\Gamma(M_2) \\
\mathsf{pr}^p_\Gamma(x) & = x \\
\mathsf{pr}^p_\Gamma(\lambda x.M_1) & = \lambda x.\mathsf{pr}^p_\Gamma(M_1) \\
\mathsf{pr}^p_\Gamma(\mathtt{ref}_n\ M_1) & = (\Pi^{(1,2)}\ \star\ \mathsf{pr}^p_\Gamma(M_1)) \\
\mathsf{pr}^p_\Gamma(\mathtt{deref}_n(M_1)) & = (\Pi^{(1,2)}\ \mathtt{V}_T\ \mathsf{pr}^p_\Gamma(M_1)) \\
\mathsf{pr}^p_\Gamma(M_1 \mathtt{:=}_n M_2) & = (\Pi^{(1,3)}\ \star\ \mathsf{pr}^p_\Gamma(M_1)\ \mathsf{pr}^p_\Gamma(M_2)) \\
\mathsf{pr}^p_\Gamma(u_{(n,T_1)}) & = \star
\end{array}
$$

We extend this definition to evaluation contexts in the following way: we always propagate the projection inductively in a context $\mathbf{E}$, without checking if the context is related to p or not. For instance, $\mathsf{pr}^p_\Gamma(\mathbf{E}_1\ M) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)\ \mathsf{pr}^p_\Gamma(M)$ even if $(\mathbf{E}_1\ M)$ is not related to p.

The projection function maps $\lambda_{\mathtt{ref}}$ terms to $\lambda_{\mathbf{ST}}$ terms, where $\lambda_{\mathbf{ST}}$ is the simply typed λ-calculus: this is stated in Lemma 10.

Definition 9 *We extend the projection function to act on types as follows:*

$$\mathsf{pr}^p_\Gamma(\mathbb{1}) = \mathbb{1} \qquad \mathsf{pr}^p_\Gamma(T\ \mathtt{ref}_n) = \mathbb{1} \qquad \mathsf{pr}^p_\Gamma(T_1 \to^n T_2) = \mathsf{pr}^p_\Gamma(T_1) \to \mathsf{pr}^p_\Gamma(T_2)\ .$$

Observe that for any type T, $\mathsf{pr}^p_\Gamma(T)$ is a simple type, and $\mathtt{V}_T$ is a simply-typed λ-term of type $\mathsf{pr}^p_\Gamma(T)$.

Lemma 10 *Take $p \in \mathbb{N}$, and suppose $\Gamma \vdash M : (T, n)$. Then $\mathsf{pr}^p_\Gamma(M)$ belongs to $\lambda_{\mathbf{ST}}$, and has type $\mathsf{pr}^p_\Gamma(T)$.*

Proof (Sketch). We reason by induction on the typing judgement in $\lambda_{\mathtt{ref}}$. If M is not related to p, the result follows directly from the remarks above. Otherwise, we reason by cases on the last rule used to type M and conclude using the induction hypothesis.

3.2 Simulation Result

In order to reason about the transitions of projected terms, the first step is to understand how projection interacts with the decomposition of a term into an evaluation context and a redex.

The lemma below explains how the projection function is propagated within a term of the form $\mathbf{E}[M]$. There are, intuitively, two possibilities, depending only on the context and on the level (p) of the projection:

– either $\mathbf{E}$ is such that $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$ for all M, that is, the projection is always propagated in the hole to M,

– or it is not the case and the context is such that, if the effect of M is too small, the projection inserts a generic value before reaching the hole in $\mathbf{E}$. In this case $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[V]$, where $\mathbf{E}_1$ is an 'initial part' of $\mathbf{E}$, and this equality holds independently from M (as long as, like said above, the effect of M is sufficiently small in some sense).

In the former case, the projection is propagated inductively inside the context to the hole, no matter the effect of M, whereas in the latter case, if the effect of M is small enough, the projection does not stop before reaching the hole in $\mathbf{E}$.

Lemma 11 *Take $p \in \mathbb{N}$, and consider a well-typed context $\mathbf{E}$. We have:*

1. *Either for all well-typed process M, $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$,*
2. *or there exist $\mathbf{E}_1$ and $\mathbf{E}_2 \neq []$ s.t. $\mathbf{E} = \mathbf{E}_1[\mathbf{E}_2]$ and, for all M, if k stands for the effect of M, we are in one of the two following cases:*
   (a) If $k \geq p$, then $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E})[\mathsf{pr}^p_\Gamma(M)]$.
   (b) If $k < p$, then $\mathsf{pr}^p_\Gamma(\mathbf{E}[M]) = \mathsf{pr}^p_\Gamma(\mathbf{E}_1)[\mathtt{V}_{T''}]$ (where T'' is the type of $\mathbf{E}_2$).

Proof (Sketch). We proceed by structural induction on $\mathbf{E}$ and distinguish two cases:

1. Either the context is not related to p. This means that $\mathbf{E}_1 = [\,]$ and $\mathbf{E}_2 = \mathbf{E}$. If $k < p$ then the whole projection gives a generic value. If $k \geq p$ then we discuss on the structure of $\mathbf{E}$, use the induction hypothesis and the definition of projection.
2. If the context is related to p we discuss on the structure of the context and use the induction hypothesis, constructing at each step the outer context $\mathbf{E}_1$. When we reach a context not related to p, we conclude using case 1.

The properties we now establish correspond to the situation, in the previous lemma, where M is an imperative redex acting on region p. The typing rules of Figure 2 insure that firing the redex yields a term which is not related to p via its effect: depending on the kind of imperative operator that is executed, this term might either be related to p via its type, or not related to p at all.

In the latter case, we are able to show that the projected versions of the two terms are related by $\twoheadrightarrow^+$ (the transitive closure of reduction in $\lambda_{\mathbf{ST}}$), which allows us to establish a simulation property.

Fact 12 *If $\mathbf{E}_2$ is not related to p, then:*

1. *If $\mathbf{E}_2 = (V_3\ \mathbf{E}_3)$ then V_3 is not related to p.*
2. *If $\mathbf{E}_2 = (\mathbf{E}_3\ M_3)$ then $\mathbf{E}_3$ is not related to p.*

Lemma 13 *If $\Gamma \vdash \mathbf{E}_2 : (T'', m)$ and $\mathbf{E}_2$ is not related to p, then for any well-typed M, M',*

1. *$\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,2)}\ \mathtt{V}_T\ M)] \twoheadrightarrow^+ \mathtt{V}_{T''}$;*
2. *$\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,3)}\ \mathtt{V}_T\ M\ M')] \twoheadrightarrow^+ \mathtt{V}_{T''}$.*

Proof (Sketch). We proceed by structural induction on $\mathbf{E}_2$. Fact 12 is necessary: for instance, if $\mathbf{E}_2 = \mathbf{E}_3\ M_3$, we have

$$\mathsf{pr}^p_\Gamma(\mathbf{E}_2)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)] = (\mathsf{pr}^p_\Gamma(\mathbf{E}_3)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)]\ \mathsf{pr}^p_\Gamma(M_3))$$

with $\mathbf{E}_3$ of type $T_3 \to T''$. Thus, we use Fact 12 and the induction hypothesis on $\mathbf{E}_3$ to get $\mathsf{pr}^p_\Gamma(\mathbf{E}_3)[(\Pi^{(1,2)}\ \mathtt{V}_T\ N)] \twoheadrightarrow^+ \mathtt{V}_{T_3 \to T''}$, from which we conclude.

Lemmas 11 and 13 allow us to derive the desired simulation property for $\lambda_{\mathtt{ref}}$, the main point being that a functional reduction on level p is projected into one reduction in the target calculus (case 4 below).

Lemma 14 (Simulation) *Consider $p \in \mathbb{N}$, and suppose $\Gamma \vdash M : (T, m)$.*

1. *If $(M, \delta) \mapsto^n_{\mathrm{I}} (M', \delta')$ and $n < p$, then $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$.*
2. *If $(M, \delta) \mapsto^p_{\mathrm{I}} (M', \delta')$, then $\mathsf{pr}^p_\Gamma(M) \twoheadrightarrow^+ \mathsf{pr}^p_\Gamma(M')$.*
3. *If $(M, \delta) \mapsto^n_{\mathrm{F}} (M', \delta')$ and $n < p$, then $\mathsf{pr}^p_\Gamma(M) = \mathsf{pr}^p_\Gamma(M')$.*
4. *If $(M, \delta) \mapsto^p_{\mathrm{F}} (M', \delta')$, then $\mathsf{pr}^p_\Gamma(M) \twoheadrightarrow \mathsf{pr}^p_\Gamma(M')$.*

Proof (Sketch). The structure of the proof is as follows. For cases 1 and 2, terms are decomposed in the same way but the arguments invoked are different. In case 1, we use the definition of projection on terms not related to p to conclude; in case 2, projection yields an "actual term" (not a generic value) and we use Lemma 13 to conclude.

For both cases, the proof for rules (**ref**) and (**deref**) differ, as in the former case the more complex term appears before the reduction (we have $\mathtt{ref}_n\ V$ which reduces to $u_{(n,T)}$) whereas in the latter case the more complex term appears after the reduction (we have $\mathtt{deref}_n(u_{(n,T)})$ which reduces to V).

Cases 3 and 4 are treated along the lines of cases 1 and 2, except that Lemma 13 is not required.

3.3 Deriving soundness

To obtain soundness, we need to show that a diverging term performs an infinite number of functional reductions on level p, for some p. For this we introduce a measure that decreases along imperative reductions on level p and does not increase along reductions on level $< p$. The measure is given by counting the *active imperative operators* of a term, which are the imperative operators (reference creations, dereferencings and assignments) that do not occur under a λ.

Definition 15 *Take M in $\lambda_{\mathtt{ref}}$. The number of active imperative operators on region p in M, written $\mathbf{Ao}^p(M)$ is defined inductively as follows:*

$$\mathbf{Ao}^p(x) = \mathbf{Ao}^p(\lambda x.M) = \mathbf{Ao}^p(u_{(n,T)}) = 0 \qquad \mathbf{Ao}^p(M\ N) = \mathbf{Ao}^p(M) + \mathbf{Ao}^p(N)$$

$$\begin{array}{l} \mathbf{Ao}^p(\mathtt{deref}_n(M)) = \mathbf{Ao}^p(\mathtt{ref}_n\ M) = \mathbf{Ao}^p(M) \qquad \text{if } n \neq p \\ \mathbf{Ao}^p(\mathtt{deref}_p(M)) = \mathbf{Ao}^p(\mathtt{ref}_p\ M) = 1 + \mathbf{Ao}^p(M) \end{array}$$

$$\begin{array}{l} \mathbf{Ao}^p(M :=_n N) = \mathbf{Ao}^p(M) + \mathbf{Ao}^p(N) \qquad \text{if } n \neq p \\ \mathbf{Ao}^p(M :=_p N) = 1 + \mathbf{Ao}^p(M) + \mathbf{Ao}^p(N) \end{array}$$

$\mathbf{Ao}^p(M)$ and the effect of M are related as follows:

Lemma 16 *If $\Gamma \vdash M : (T, m)$ and $m < p$ then $\mathbf{Ao}^p(M) = 0$.*

We are finally able to show that $\mathbf{Ao}^p(M)$ yields the measure we need.

Lemma 17 *If $\Gamma \vdash M : (T, m)$ then:*

1. *if $(M, \delta) \mapsto^n_{\mathrm{F}} (M', \delta')$ with $n < p$ then $\mathbf{Ao}^p(M') \leq \mathbf{Ao}^p(M)$,*
2. *if $(M, \delta) \mapsto^n_{\mathrm{I}} (M', \delta')$ with $n < p$ then $\mathbf{Ao}^p(M') \leq \mathbf{Ao}^p(M)$,*
3. *and if $(M, \delta) \mapsto^p_{\mathrm{I}} (M', \delta')$ then $\mathbf{Ao}^p(M') < \mathbf{Ao}^p(M)$.*

Proof (Sketch). We examine each reduction rule and use Lemma 16 to observe that new imperative operators on region p can only be generated by functional reductions on level $\geq p$ or by imperative reductions on level $> p$, and that every imperative reduction on level p erases one active imperative operator on region p.

The following lemma states that there exists a maximum region p on which an infinite number of reductions takes place. With the previous result, we are able to show that an infinite number of functional reductions take place on level p.

Lemma 18 *Suppose that $\Gamma \vdash M : (T, l)$, and that there exists $(M_i, \delta_i)_{i \in \mathbb{N}}$, an infinite reduction sequence starting from M. Then:*

1. *For all i, M_i is typable.*
2. *There exist p and i_o s.t.*
   (a) *if $i > i_0$ and $(M_i, \delta_i) \mapsto^n_{\mathrm{I}} (M_{i+1}, \delta_{i+1})$ then $n \leq p$,*
   (b) *if $i > i_0$ and $(M_i, \delta_i) \mapsto^n_{\mathrm{F}} (M_{i+1}, \delta_{i+1})$ then $n \leq p$,*
   (c) *There exists an infinite set of indexes $\mathcal{I}$ s.t. for each $i \in \mathcal{I}$, either $(M_i, \delta_i) \mapsto^p_{\mathrm{F}} (M_{i+1}, \delta_{i+1})$ or $(M_i, \delta_i) \mapsto^p_{\mathrm{I}} (M_{i+1}, \delta_{i+1})$.*
   (d) *There are infinitely many $i \in \mathcal{I}$ s.t. $(M_i, \delta_i) \mapsto^p_{\mathrm{F}} (M_{i+1}, \delta_{i+1})$.*

Proof (Sketch).

1. Follows from Proposition 5.
2. The set of different regions is finite, so we easily find a p satisfying 2a, 2b and 2c. Lemma 17 ensures that 2d holds.

Theorem 19 (Soundness) *If $\Gamma \vdash M : (T, m)$ then M terminates.*

Proof. Consider, by absurd, an infinite computation $\{M_i\}_i$ starting from $M = M_0$. By Lemma 18, all the M_i's are well-typed, and there is a maximal p s.t. for infinitely many i, $(M_i, \delta_i) \mapsto^p_{\mathrm{F}} (M_{i+1}, \delta_{i+1})$ and an index i_0 exists such that every reduction on an index greater than i_0 is performed on region $n \leq p$. Consider the sequence $(\mathsf{pr}^p_\Gamma(M_i))_{i > i_0}$. By Lemma 14, we obtain that for every $i > i_0$, $\mathsf{pr}^p_\Gamma(M_i) \twoheadrightarrow^* \mathsf{pr}^p_\Gamma(M_{i+1})$. Moreover, $\mathsf{pr}^p_\Gamma(M_i) \twoheadrightarrow^+ \mathsf{pr}^p_\Gamma(M_{i+1})$ for an infinite number of i. Thus $\mathsf{pr}^p_\Gamma(M_{i_0})$ is diverging. This contradicts the termination of $\lambda_{\mathbf{ST}}$.

Remark 20 (Raising the effect) *The results we present in this paper still hold if we add the rule:*

$$(\textbf{Sub})\frac{\Gamma \vdash M : (T, n) \qquad n \leq n'}{\Gamma \vdash M : (T, n')}$$

to the type system.

This rule allows us to be more liberal when typing term and to reach a greater expressiveness. For instance, it allows one to store at the same address functions whose bodies do not have the same effect.

Example 21 (Landin's trick) *The standard example of diverging term in λ_{ref}, known as* Landin's trick*, is given by:*

$$(\lambda f.[(\lambda t.(\mathtt{deref}_1(f)\ \star))\ (f :=_1 \lambda z.(\mathtt{deref}_1(f)\ z))])\ (\mathtt{ref}_1\ \lambda x.x)\ .$$

In order to try and type this term, we are bound to manipulate non well-formed types.

In the call by value setting of λ_{ref}, a first address $u_{(1,\mathbb{1}\to^1\mathbb{1})}$ (we use Remark 20 here, as the identity has no effect) is created when evaluating the argument $(\mathtt{ref}_1\ \lambda x.x)$ and instantiates f in the body of the outer function. Then $u_{(1,\mathbb{1}\to^1\mathbb{1})}$ is instantiated by the function $\lambda z.(\mathtt{deref}_1(f)\ z)$ whose type is $\mathbb{1} \to^1 \mathbb{1}$ then the term enters a loop. It is easy to see that the type of $u_{(1,\mathbb{1}\to^1\mathbb{1})}$ (which is also the type of f) is $(\mathbb{1} \to^1 \mathbb{1})\ \mathtt{ref}_1$ and is not well-formed, as $\mathrm{reg}(\mathbb{1} \to^1 \mathbb{1}) = 1 \not< 1$.

On the other hand, consider the following terminating term:

$$(\lambda f.[(\lambda t.(\mathtt{deref}_1(f)\ \star))\ (f :=_1 \lambda z.(\Pi^{(1,2)}\ I\ (\lambda y.\mathtt{deref}_1(f)\ y))\ z])\ (\mathtt{ref}_1\ \lambda x.x)$$

where $I = \lambda t.\, t$. This term is close to the example given above, except that $\lambda z.(\mathtt{deref}_1(f)\ z)$ is replaced with $\lambda z.(\Pi^{(1,2)}\ I\ (\lambda y.\mathtt{deref}_1(f)\ y)\ z$. This new subterm, stored at address f, contains a dereferencing of f. Yet the term terminates because the read in memory never comes in redex position; the term $(\lambda z.(\Pi^{(1,2)}\ I\ (\lambda y.\mathtt{deref}_1(f)\ y))\ z)$ reduces to $(\Pi^{(1,2)}\ I\ (\lambda y.\mathtt{deref}_1(f)\ y))$ which, in turn, reduces in two steps to I.

Here the type system asigns to $(\Pi^{(1,2)}\ I\ (\lambda y.\mathtt{deref}_1(f)\ y))$ the type $\mathbb{1} \to^0 \mathbb{1}$ and the effect 0. Thus the type of x is $\mathbb{1} \to^0 \mathbb{1}\ \mathtt{ref}_1$, which is well-formed.

4 Parametricity

As is the case in [DHS10] for the π-calculus, the method we have presented for the λ-calculus with references is parametric with respect to a terminating purely functional core, and does not examine the corresponding termination proof. Other core calculi could be considered. Moreover, if the functional calculus corresponds to a subset of the simply typed terms, then the result holds directly.

We believe that it is possible to extend our work to polymorphic types, although this extension is not trivial if we consider adding region polymorphism:

for instance, we would have to guarantee that a type like $(\forall A.A \to^0 A)\ \mathtt{ref}_n$ cannot have its A component instantiated with a type containing a region strictly greater than n.

Another idea is to apply this termination technique to a language containing both references and a recursion operator on integers. By restricting the use of the latter (in order not to create loops based on recursion), we think that one could be able to build a type system ensuring termination. Yet, this system would involve more technical difficulties than the one we presented here, as one would have to ensure that the two features do not collaborate to create divergences.

By taking as functional core a λ-calculus with complexity bounds (such as, for instance, [ABM10]), we believe that one can use our technique in order to lift complexity bounds for impure languages. The main idea is to rely on the projection function to provide bounds on the number of reductions a terminating typed term can make.

Note, to conclude, that references can be encoded in a standard way in the π-calculus (as well as the call-by-value λ-calculus). One could then wonder if the method presented in [DHS10] can recognise as terminating the subset of π-processes corresponding to encodings of $\lambda_{\mathtt{ref}}$ terms. The question is challenging, as weight-based methods for termination in π [DS06] cannot be used to prove termination of the encoding of $\lambda_{\mathbf{ST}}$.

Acknowledgements. Support from the french ANR projects "CHoCo", "AEOLUS" and "Complice" (ANR-08-BLANC-0211-01), and by the European Project "HATS" (contract number 231620) is acknowledged.

References

[ABM10] Roberto M. Amadio, Patrick Baillot, and Antoine Madet. An affine-intuitionistic system of types and effects: confluence and termination. *CoRR*, abs/1005.0835, 2010.

[Ama09] Roberto M. Amadio. On Stratified Regions. In *Proc. of APLAS*, volume 5904 of *LNCS*, pages 210–225. Springer, 2009.

[Bou07] Gérard Boudol. Fair Cooperative Multithreading. In *Proc. of CONCUR*, volume 4703 of *LNCS*, pages 272–286. Springer, 2007.

[Dem10] Romain Demangeon. *Termination for Concurrent Systems.* PhD thesis, Ecole Normale Superieure de Lyon, 2010.
Available from `http://perso.ens-lyon.fr/romain.demangeon/phd.pdf`.

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

[DS06] Yuxin Deng and Davide Sangiorgi. Ensuring Termination by Typability. *Information and Computation*, 204(7):1045–1082, 2006.

[Tra10] Paolo Tranquilli. Translating types and effects with state monads and linear logic. submitted, 2010.