Projet PiCoq

Deliverable D123

December 2013

Checking NFA equivalence with bisimulations up to congruence

Filippo Bonchi Damien Pous
CNRS, ENS Lyon, Université de Lyon, LIP (UMR 5668)
{filippo.bonchi,damien.pous}@ens-lyon.fr

Abstract

We introduce *bisimulation up to congruence* as a technique for proving language equivalence of non-deterministic finite automata. Exploiting this technique, we devise an optimisation of the classical algorithm by Hopcroft and Karp [16]. We compare our approach to the recently introduced antichain algorithms, by analysing and relating the two underlying coinductive proof methods. We give concrete examples where we exponentially improve over antichains; experimental results moreover show non negligible improvements.

Keywords Language Equivalence, Automata, Bisimulation, Coinduction, Up-to techniques, Congruence, Antichains.

1. Introduction

Checking language equivalence of finite automata is a classical problem in computer science, which finds applications in many fields ranging from compiler construction to model checking.

Equivalence of deterministic finite automata (DFA) can be checked either via minimisation [9, 15] or through Hopcroft and Karp's algorithm [2, 16], which exploits an instance of what is nowadays called a *coinduction proof principle* [24, 27, 29]: two states recognise the same language if and only if there exists a *bisimulation* relating them. In order to check the equivalence of two given states, Hopcroft and Karp's algorithm creates a relation containing them and tries to build a bisimulation by adding pairs of states to this relation: if it succeeds then the two states are equivalent, otherwise they are different.

On the one hand, minimisation algorithms have the advantage of checking the equivalence of all the states at once (while Hopcroft and Karp's algorithm only check a given pair of states). On the other hand, they have the disadvantage of needing the whole automata from the beginning[1], while Hopcroft and Karp's algorithm can be executed "on-the-fly" [12], on a lazy DFA whose transitions are computed on demand.

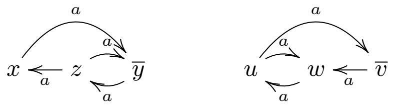
This difference is fundamental for our work and for other recently introduced algorithms based on *antichains* [1, 33]. Indeed, when starting from non-deterministic finite automata (NFA), the powerset construction used to get deterministic automata induces an exponential factor. In contrast, the algorithm we introduce in this work for checking equivalence of NFA (as well as those in [1, 33]) usually does not build the whole deterministic automaton, but just a small part of it. We write "usually" because in few bad cases, the algorithm still needs exponentially many states of the DFA.

Our algorithm is grounded on a simple observation on determinised NFA: for all sets X and Y of states of the original NFA, the union (written $+$) of the language recognised by X (written $[\![X]\!]$) and the language recognised by Y ($[\![Y]\!]$) is equal to the language recognised by the union of X and Y ($[\![X+Y]\!]$). In symbols:

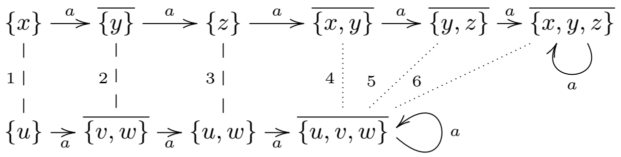
$$[\![X+Y]\!] = [\![X]\!] + [\![Y]\!] \tag{1}$$

This fact leads us to introduce a sound and complete proof technique for language equivalence, namely *bisimulation up to context*, that exploits both *induction* (on the operator $+$) and *coinduction*: if a bisimulation R equates both the (sets of) states X_1, Y_1 and X_2, Y_2, then $[\![X_1]\!] = [\![Y_1]\!]$ and $[\![X_2]\!] = [\![Y_2]\!]$ and, by (1), we can immediately conclude that also $X_1 + X_2$ and $Y_1 + Y_2$ are language equivalent. Intuitively, bisimulations up to context are bisimulations which *do not need to relate* $X_1 + X_2$ and $Y_1 + Y_2$ when X_1 (resp. X_2) and Y_1 (resp. Y_2) are already related.

To illustrate this idea, let us check the equivalence of states x and u in the following NFA. (Final states are overlined, labelled edges represent transitions.)



The determinised automaton is depicted below.



Each state is a set of states of the NFA, final states are overlined: they contain at least one final state of the NFA. The numbered lines show a relation which is a bisimulation containing x and u. Actually, this is the relation that is built by Hopcroft and Karp's algorithm (the numbers express the order in which pairs are added).

The dashed lines (numbered by 1, 2, 3) form a smaller relation which is not a bisimulation, but a bisimulation up to context: the equivalence of states $\{x, y\}$ and $\{u, v, w\}$ could be immediately deduced from the fact that $\{x\}$ is related to $\{u\}$ and $\{y\}$ to $\{v, w\}$, without the need of further exploring the determinised automaton.

Bisimulations up-to, and in particular bisimulations up to context, have been introduced in the setting of concurrency theory [24,

[1] There are few exceptions, like [19] which minimises labelled transition systems w.r.t. bisimilarity rather than trace equivalence.

25, 28] as a proof technique for bisimilarity of CCS or π-calculus processes. As far as we know, they have never been used for proving language equivalence of NFA.

Among these techniques one should also mention *bisimulation up to equivalence*, which, as we show in this paper, is implicitly used in the original Hopcroft and Karp's algorithm. This technique can be briefly explained by noting that not all bisimulations are equivalence relations: it might be the case that a bisimulation relates (for instance) X and Y, Y and Z but not X and Z. However, since $[\![X]\!] = [\![Y]\!]$ and $[\![Y]\!] = [\![Z]\!]$, we can immediately conclude that X and Z recognise the same language. Analogously to bisimulations up to context, a bisimulation up to equivalence *does not need to relate* X and Z when they are both related to some Y.

The techniques of up-to equivalence and up-to context can be combined resulting in a powerful proof technique which we call *bisimulation up to congruence*. Our algorithm is in fact just an extension of Hopcroft and Karp's algorithm that attempts to build a bisimulation up to congruence instead of a bisimulation up to equivalence. An important consequence, when using up to congruence, is that we do not need to build the whole deterministic automata, but just those states that are needed for the bisimulation up-to. For instance, in the above NFA, the algorithm stops after equating z and $u + v$ and does not build the remaining four states. Despite their use of the up to equivalence technique, this is not the case with Hopcroft and Karp's algorithm, where all accessible subsets of the deterministic automata have to be visited at least once.

The ability of visiting only a small portion of the determinised automaton is also the key feature of the antichain algorithm [33] and its optimisation exploiting similarity [1]. The two algorithms are designed to check *language inclusion* rather than equivalence, but we can relate these approaches by observing that the two problems are equivalent ($[\![X]\!] = [\![Y]\!]$ iff $[\![X]\!] \subseteq [\![Y]\!]$ and $[\![Y]\!] \subseteq [\![X]\!]$; and $[\![X]\!] \subseteq [\![Y]\!]$ iff $[\![X]\!] + [\![Y]\!] = [\![Y]\!]$ iff $[\![X + Y]\!] = [\![Y]\!]$).

In order to compare with these algorithms, we make explicit the coinductive up-to technique underlying the antichain algorithm [33]. We prove that this technique can be seen as a restriction of up to congruence, for which *symmetry* and *transitivity* are not allowed. As a consequence, the antichain algorithm usually needs to explore more states than our algorithm. Moreover, we show how to integrate the optimisation proposed in [1] in our setting, resulting in an even more efficient algorithm.

Summarising, the contributions of this work are

(1) the observation that Hopcroft and Karp implicitly use bisimulations up to equivalence (Section 2),

(2) an efficient algorithm for checking language equivalence (and inclusion), based on a powerful up to technique (Section 3),

(3) a comparison with antichain algorithms, by recasting them into our coinductive framework (Sections 4 and 5).

Outline

Section 2 recalls Hopcroft and Karp's algorithm for DFA, showing that it implicitly exploits bisimulation up to equivalence. Section 3 describes the novel algorithm, based on bisimulations up to congruence. We compare this algorithm with the antichain one in Section 4, and we show how to exploit similarity in Section 5. Section 6 is devoted to benchmarks. Sections 7 and 8 discuss related and future works. Omitted proofs can be found in the Appendix.

Notation

We denote sets by capital letters $X, Y, S, T \dots$ and functions by lower case letters $f, g, \dots$ Given sets X and Y, $X \times Y$ is their Cartesian product, $X \uplus Y$ is the disjoint union and X^Y is the set of functions $f\colon Y \to X$. Finite iterations of a function $f\colon X \to X$ are denoted by f^n (formally, $f^0(x) = x$, $f^{n+1}(x) = f(f^n(x))$). The collection of subsets of X is denoted by $\mathcal{P}(X)$. The (omega) iteration of a function $f\colon \mathcal{P}(X) \to \mathcal{P}(X)$ is denoted by f^ω (formally, $f^\omega(Y) = \bigcup_{n\geq 0} f^n(Y)$). For a set of letters A, $A^\star$ denotes the set of all finite words over A; ϵ the empty word; and $w_1 w_2$ the concatenation of words $w_1, w_2 \in A^\star$. We use 2 for the set $\{0, 1\}$ and $2^{A^\star}$ for the set of all languages over A.

2. Hopcroft and Karp's algorithm for DFA

A deterministic finite automaton (DFA) over the alphabet A is a triple (S, o, t), where S is a finite set of states, $o\colon S \to 2$ is the output function, which determines if a state $x \in S$ is final ($o(x) = 1$) or not ($o(x) = 0$), and $t\colon S \to S^A$ is the transition function which returns, for each state x and for each letter $a \in A$, the next state $t_a(x)$. For $a \in A$, we write $x \xrightarrow{a} x'$ to mean that $t_a(x) = x'$. For $w \in A^\star$, we write $x \xrightarrow{w} x'$ for the least relation such that (1) $x \xrightarrow{\epsilon} x$ and (2) $x \xrightarrow{aw'} x'$ iff $x \xrightarrow{a} x''$ and $x'' \xrightarrow{w'} x'$.

For any DFA, there exists a function $[\![-]\!]\colon S \to 2^{A^\star}$ mapping states to languages, defined for all $x \in S$ as follows:

$$[\![x]\!](\epsilon) = o(x)\ , \qquad [\![x]\!](aw) = [\![t_a(x)]\!](w)\ .$$

The language $[\![x]\!]$ is called the language accepted by x. Given two automata (S_1, o_1, t_1) and (S_2, o_2, t_2), the states $x_1 \in S_1$ and $x_2 \in S_2$ are said to be *language equivalent* (written $x_1 \sim x_2$) iff they accept they same language.

Remark 1. *In the following, we will always consider the problem of checking the equivalence of states of one single and fixed automaton (S, o, t). We do not loose generality since for any two automata (S_1, o_1, t_1) and (S_2, o_2, t_2) it is always possible to build an automaton $(S_1 \uplus S_2, o_1 \uplus o_2, t_1 \uplus t_2)$ such that the language accepted by every state $x \in S_1 \uplus S_2$ is the same as the language accepted by x in the original automaton (S_i, o_i, t_i). For this reason, we also work with automata without explicit initial states: we focus on the equivalence of two arbitrary states of a fixed DFA.*

2.1 Proving language equivalence via coinduction

We first define bisimulation. We make explicit the underlying notion of progression which we need in the sequel.

Definition 1 (Progression, Bisimulation). *Given two relations $R, R' \subseteq S \times S$ on states,* R progresses to R'*, denoted $R \rightarrowtail R'$, if whenever $x\ R\ y$ then*

1. *$o(x) = o(y)$ and*
2. *for all $a \in A$, $t_a(x)\ R'\ t_a(y)$.*

A bisimulation *is a relation R such that $R \rightarrowtail R$.*

As expected, bisimulation is a sound and complete proof technique for checking language equivalence of DFA:

Proposition 1 (Coinduction). *Two states are language equivalent iff there exists a bisimulation that relates them.*

2.2 Naive algorithm

Figure 1 shows a naive version of Hopcroft and Karp's algorithm for checking language equivalence of the states x and y of a deterministic finite automaton (S, o, t). Starting from x and y, the algorithm builds a relation R that, in case of success, is a bisimulation. In order to do that, it employs the set (of pairs of states) $todo$ which, intuitively, at any step of the execution, contains the pairs (x', y') that must be checked: if (x', y') already belongs to R, then it has already been checked and nothing else should be done. Otherwise, the algorithm checks if x' and y' have the same outputs (i.e., if both are final or not). If $o(x') \neq o(y')$, then x and y are different.

```
                    Naive(x, y)
(1) R is empty; todo is empty;
(2) insert (x, y) in todo;
(3) while todo is not empty, do {
 (3.1)  extract (x', y') from todo;
 (3.2)  if (x', y') ∈ R then skip;
 (3.3)  if o(x') ≠ o(y') then return false;
 (3.4)  for all a ∈ A,
          insert (t_a(x'), t_a(y')) in todo;
 (3.5)  insert (x', y') in R;
(4) return true;
```

Figure 1. Naive algorithm for checking the equivalence of states x and y of a DFA (S, o, t); R and *todo* are sets of pairs of states. The code of $\texttt{HK}(x, y)$ is obtained by replacing step 3.2 with `if` $(x', y') \in e(R)$ `then skip`.

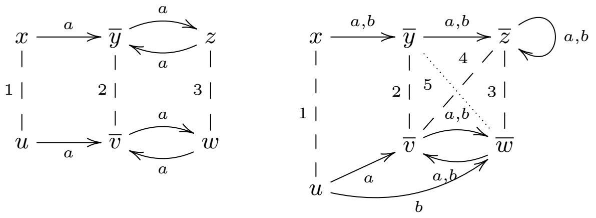


Figure 2. Checking for DFA equivalence.

If $o(x') = o(y')$, then the algorithm inserts (x', y') in R and, for all $a \in A$, the pairs $(t_a(x'), t_a(y'))$ in *todo*.

Proposition 2. *For all $x, y \in S$, $x \sim y$ iff* $\texttt{Naive}(x, y)$.

Proof. We first observe that if $\texttt{Naive}(x, y)$ returns true then the relation R that is built before arriving to step 4 is a bisimulation. Indeed, the following proposition is an invariant for the loop corresponding to step 3:

$$R \rightarrowtail R \cup todo$$

This invariant is preserved since at any iteration of the algorithm, a pair (x', y') is removed from *todo* and inserted in R after checking that $o(x') = o(y')$ and adding $(t_a(x'), t_a(y'))$ for all $a \in A$ in *todo*. Since *todo* is empty at the end of the loop, we eventually have $R \rightarrowtail R$, i.e., R is a bisimulation. By Proposition 1, $x \sim y$.

We now prove that if $\texttt{Naive}(x, y)$ returns false, then $x \not\sim y$. Note that for all (x', y') inserted in *todo*, there exists a word $w \in A^\star$ such that $x \overset{w}{\to} x'$ and $y \overset{w}{\to} y'$. Since $o(x') \neq o(y')$, then $[\![x']\!](\epsilon) \neq [\![y']\!](\epsilon)$ and thus $[\![x]\!](w) = [\![x']\!](\epsilon) \neq [\![y']\!](\epsilon) = [\![y]\!](w)$, that is $x \not\sim y$. □

Since both Hopcroft and Karp's algorithm and the one we introduce in Section 3 are simple variations of this naive one, it is important to illustrate its execution with an example. Consider the DFA with input alphabet $A = \{a\}$ in the left-hand side of Figure 2, and suppose we want to check that x and u are language equivalent.

During the initialisation, (x, u) is inserted in *todo*. At the first iteration, since $o(x) = 0 = o(u)$, (x, u) is inserted in R and (y, v) in *todo*. At the second iteration, since $o(y) = 1 = o(v)$, (y, v) is inserted in R and (z, w) in *todo*. At the third iteration, since $o(z) = 0 = o(w)$, (z, w) is inserted in R and (y, v) in *todo*. At the fourth iteration, since (y, v) is already in R, the algorithm does nothing. Since there are no more pairs to check in *todo*, the relation R is a bisimulation and the algorithm terminates returning true.

These iterations are concisely described by the numbered dashed lines in Figure 2. The line i means that the connected pair is inserted in R at iteration i. (In the sequel, when enumerating iterations, we ignore those where a pair from *todo* is already in R so that there is nothing to do.)

Remark 2. *Unless it finds a counter-example,* $\texttt{Naive}$ *constructs the* smallest *bisimulation that relates the two starting states (see Proposition 8 in Appendix A). On the contrary, minimisation algorithms [9, 15] are designed to compute the* largest *bisimulation relation for a given automaton. For instance, taking automaton on the left of Figure 2, they would equate the states x and w which are language equivalent, while* $\texttt{Naive}(x, u)$ *does not relate them.*

2.3 Hopcroft and Karp's algorithm

The naive algorithm is quadratic: a new pair is added to R at each non-trivial iteration, and there are only n^2 such pairs, where $n = |S|$ is the number of states of the DFA. To make this algorithm (almost) linear, Hopcroft and Karp actually record a set of *equivalence classes* rather than a set of visited pairs. As a consequence, their algorithm may stop earlier, when an encountered pair of states is not already in R but in its reflexive, symmetric, and transitive closure. For instance in the right-hand side example from Figure 2, we can stop when we encounter the dotted pair (y, w), since these two states already belong to the same equivalence class according to the four previous pairs.

With this optimisation, the produced relation R contains at most n pairs (two equivalence classes are merged each time a pair is added). Formally, and ignoring the concrete data structure to store equivalence classes, Hopcroft and Karp's algorithm consists in simply replacing step 3.2 in Figure 1 with

```
(3.2)  if (x', y') ∈ e(R) then skip;
```

where $e\colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ is the function mapping each relation $R \subseteq S \times S$ into its symmetric, reflexive, and transitive closure. We hereafter refer to this algorithm as $\texttt{HK}$.

2.4 Bisimulations up-to

We now show that the optimisation used by Hopcroft and Karp corresponds to exploiting an "up-to technique".

Definition 2 (Bisimulation up-to)**.** *Let $f\colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ be a function on relations on S. A relation R is a* bisimulation up to f *if $R \rightarrowtail f(R)$, i.e., whenever $x\ R\ y$ then*

1. *$o(x) = o(y)$ and*
2. *for all $a \in A$, $t_a(x)\ f(R)\ t_a(y)$.*

With this definition, Hopcroft and Karp's algorithm just consists in trying to build a bisimulation up to e. To prove the correctness of the algorithm it suffices to show that any bisimulation up to e is contained in a bisimulation. We use for that the notion of compatible function [26, 28]:

Definition 3 (Compatible function)**.** *A function $f\colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ is* compatible *if it is monotone and it preserves progressions: for all $R, R' \subseteq S \times S$,*

$$R \rightarrowtail R' \text{ entails } f(R) \rightarrowtail f(R') .$$

Proposition 3. *Let f be a compatible function. Any bisimulation up to f is contained in a bisimulation.*

Proof. Suppose that R is a bisimulation up to f, i.e., that $R \rightarrowtail f(R)$. Using compatibility of f and by a simple induction on n, we get $\forall n, f^n(R) \rightarrowtail f^{n+1}(R)$. Therefore, we have

$$\bigcup_n f^n(R) \rightarrowtail \bigcup_n f^n(R) ,$$

in other words, $f^\omega(R) = \bigcup_n f^n(R)$ is a bisimulation. This latter relation trivially contains R, by taking $n = 0$. □

We could prove directly that e is a compatible function; we however take a detour to ease our correctness proof for the algorithm we propose in Section 3.

Lemma 1. *The following functions are compatible:*

- id: *the identity function;*
- $f \circ g$: *the composition of compatible functions f and g;*
- $\bigcup F$: *the pointwise union of an arbitrary family F of compatible functions: $\bigcup F(R) = \bigcup_{f\in F} f(R)$;*
- f^ω: *the (omega) iteration of a compatible function f.*

Lemma 2. *The following functions are compatible:*

- *the constant reflexive function: $r(R) = \{(x, x) \mid \forall x \in S\}$;*
- *the converse function: $s(R) = \{(y, x) \mid x\ R\ y\}$;*
- *the squaring function: $t(R) = \{(x, z) \mid \exists y, x\ R\ y\ R\ z\}$.*

Intuitively, given a relation R, $(s \cup \mathrm{id})(R)$ is the symmetric closure of R, $(r \cup s \cup \mathrm{id})(R)$ is its reflexive and symmetric closure, and $(r \cup s \cup t \cup \mathrm{id})^\omega(R)$ is its symmetric, reflexive and transitive closure: $e = (r \cup s \cup t \cup \mathrm{id})^\omega$. Another way to understand this decomposition of e is to recall that for a given R, $e(R)$ can be defined inductively by the following rules:

$$\frac{}{x\ e(R)\ x}\ r \qquad \frac{x\ e(R)\ y}{y\ e(R)\ x}\ s \qquad \frac{x\ e(R)\ y \quad y\ e(R)\ z}{x\ e(R)\ z}\ t \qquad \frac{x\ R\ y}{x\ e(R)\ y}\ \mathrm{id}$$

Theorem 1. *Any bisimulation up to e is contained in a bisimulation.*

Proof. By Proposition 3, it suffices to show that e is compatible, which follows from Lemma 1 and Lemma 2. □

Corollary 1. *For all $x, y \in S$, $x \sim y$ iff* $\mathtt{HK}(x, y)$.

Proof. Same proof as for Proposition 2, by using the invariant $R \rightarrowtail e(R) \cup todo$. We deduce that R is a bisimulation up to e after the loop. We conclude with Theorem 1 and Proposition 1. □

Returning to the right-hand side example from Figure 2, Hopcroft and Karp's algorithm constructs the relation

$$R_{\mathtt{HK}} = \{(x, u), (y, v), (z, w), (z, v)\}$$

which is not a bisimulation, but a bisimulation up to e: it contains the pair (x, u), whose b-transitions lead to (y, w), which is not in $R_{\mathtt{HK}}$ but in its equivalence closure, $e(R_{\mathtt{HK}})$.

3. Optimised algorithm for NFA

We now move from DFA to non-deterministic automata (NFA). We start with standard definitions about semi-lattices, determinisation, and language equivalence for NFA.

A *semi-lattice* $(X, +, 0)$ consists of a set X and a binary operation $+: X \times X \to X$ which is associative, commutative, idempotent (ACI), and has $0 \in X$ as identity. Given two semi-lattices $(X_1, +_1, 0_1)$ and $(X_2, +_2, 0_2)$, an *homomorphism* of semi-lattices is a function $f: X_1 \to X_2$ such that for all $x, y \in X_1$, $f(x +_1 y) = f(x) +_2 f(y)$ and $f(0_1) = 0_2$. The set $2 = \{0, 1\}$ is a semi-lattice when taking $+$ to be the ordinary Boolean or. Also the set of all languages $2^{A^\star}$ carries a semi-lattice where $+$ is the union of languages and 0 is the empty language. More generally, for any set X, $\mathcal{P}(X)$ is a semi-lattice where $+$ is the union of sets and 0 is the empty set. In the sequel, we indiscriminately use 0 to denote the element $0 \in 2$, the empty language in $2^{A^\star}$, and the empty set in $\mathcal{P}(X)$. Similarly, we use $+$ to denote the Boolean or in 2, the union of languages in $2^{A^\star}$, and the union of sets in $\mathcal{P}(X)$.

A non-deterministic finite automaton (NFA) over the input alphabet A is a triple (S, o, t), where S is a finite set of states, $o: S \to 2$ is the output function (as for DFA), and $t: S \to \mathcal{P}(S)^A$ is the transition relation, which assigns to each state $x \in S$ and input letter $a \in A$ a set of possible successor states.

The *powerset construction* transforms any NFA (S, o, t) in the DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ where $o^\sharp: \mathcal{P}(S) \to 2$ and $t^\sharp: \mathcal{P}(S) \to \mathcal{P}(S)^A$ are defined for all $X \in \mathcal{P}(S)$ and $a \in A$ as follows:

$$o^\sharp(X) = \begin{cases} o(x) & \text{if } X = \{x\} \text{ with } x \in S \\ 0 & \text{if } X = 0 \\ o^\sharp(X_1) + o^\sharp(X_2) & \text{if } X = X_1 + X_2 \end{cases}$$

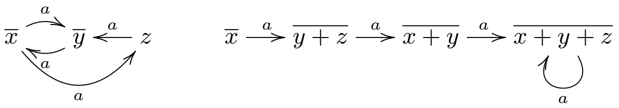
$$t^\sharp_a(X) = \begin{cases} t_a(x) & \text{if } X = \{x\} \text{ with } x \in S \\ 0 & \text{if } X = 0 \\ t^\sharp_a(X_1) + t^\sharp_a(X_2) & \text{if } X = X_1 + X_2 \end{cases}$$

Observe that in $(\mathcal{P}(S), o^\sharp, t^\sharp)$, the states form a semi-lattice $(\mathcal{P}(S), +, 0)$, and $o^\sharp$ and $t^\sharp$ are, by definition, semi-lattices homomorphisms. These properties are fundamental for the up-to technique we are going to introduce; in order to highlight the difference with generic DFA (which usually do not carry this structure), we introduce the following definition.

Definition 4. *A* determinised NFA *is a DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ obtained via the powerset construction of some NFA (S, o, t).*

Hereafter, we use a new notation for representing states of determinised NFA: in place of the singleton $\{x\}$ we just write x and, in place of $\{x_1, \dots, x_n\}$, we write $x_1 + \dots + x_n$.

For an example, consider the NFA (S, o, t) depicted below (left) and part of the determinised NFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ (right).



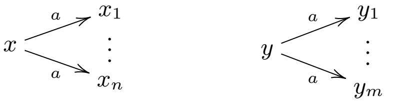
In the determinised NFA, x makes one single a-transition going into $y + z$. This state is final: $o^\sharp(y + z) = o^\sharp(y) + o^\sharp(z) = o(y) + o(z) = 1 + 0 = 1$; it makes an a-transition into $t^\sharp_a(y + z) = t^\sharp_a(y) + t^\sharp_a(z) = t_a(y) + t_a(z) = x + y$.

The language accepted by the states of a NFA (S, o, t) can be conveniently defined via the powerset construction: the language accepted by $x \in S$ is the language accepted by the singleton $\{x\}$ in the DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$, in symbols $[\![\{x\}]\!]$. Therefore, in the following, instead of considering the problem of language equivalence of states of the NFA, we focus on language equivalence of *sets* of states of the NFA: given two sets of states X and Y in $\mathcal{P}(S)$, we say that X and Y are language equivalent ($X \sim Y$) iff $[\![X]\!] = [\![Y]\!]$. This is exactly what happens in standard automata theory, where NFA are equipped with sets of initial states.

3.1 Extending coinduction to NFA

In order to check if two sets of states X and Y of an NFA (S, o, t) are language equivalent, we can simply employ the bisimulation proof method on $(\mathcal{P}(S), o^\sharp, t^\sharp)$. More explicitly, a bisimulation for a NFA (S, o, t) is a relation $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ on sets of states, such that whenever $X\ R\ Y$ then (1) $o^\sharp(X) = o^\sharp(Y)$, and (2) for all $a \in A$, $t^\sharp_a(X)\ R\ t^\sharp_a(Y)$. Since this is just the old definition of bisimulation (Definition 1) applied to $(\mathcal{P}(S), o^\sharp, t^\sharp)$, we get that $X \sim Y$ iff there exists a bisimulation relating them.

Remark 3 (Linear time v.s. branching time)**.** *It is important not to confuse these bisimulation relations with the standard Milner-and-Park bisimulations [24] (which* strictly *imply language equivalence): in a standard bisimulation R, if the following states x and y of an NFA are in R,*



then each x_i should be in R with some y_j (and vice-versa). Here, instead, we first transform the transition relation into

$$x \xrightarrow{a} x_1 + \cdots + x_n \qquad y \xrightarrow{a} y_1 + \cdots + y_m \ ,$$

using the powerset construction, and then we require that the sets $x_1 + \cdots + x_n$ and $y_1 + \cdots + y_m$ are related by R.

3.2 Bisimulation up to congruence

The semi-lattice structure $(\mathcal{P}(S), +, 0)$ carried by determinised NFA makes it possible to introduce a new up-to technique, which is not available with plain DFA: *up to congruence*. This technique relies on the following function.

Definition 5 (Congruence closure)**.** *Let $u\colon \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S)) \to \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S))$ be the function on relations on sets of states defined for all $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ as:*

$$u(R) = \{(X_1 + X_2,\ Y_1 + Y_2) \mid X_1\ R\ Y_1 \text{ and } X_2\ R\ Y_2\}\,.$$

The function $c = (r \cup s \cup t \cup u \cup \mathrm{id})^\omega$ is called the congruence closure *function.*

Intuitively, $c(R)$ is the smallest equivalence relation which is closed with respect to $+$ and which includes R. It could alternatively be defined inductively using the rules r, s, t, and id from the previous section, and the following one:

$$\frac{X_1\ c(R)\ Y_1 \qquad X_2\ c(R)\ Y_2}{X_1 + X_2\ c(R)\ Y_1 + Y_2}\ u$$

We call bisimulations up to congruence the bisimulations up to c. We report the explicit definition for the sake of clarity:

Definition 6 (Bisimulation up to congruence)**.** *A bisimulation up to congruence for a NFA (S, o, t) is a relation $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ on sets of states, such that whenever $X\ R\ Y$ then*

1. *$o^\sharp(X) = o^\sharp(Y)$ and*
2. *for all $a \in A$, $t_a^\sharp(X)\ c(R)\ t_a^\sharp(Y)$.*

We then show that bisimulations up to congruence are sound, using the notion of compatibility:

Lemma 3. *The function u is compatible.*

Proof. We assume that $R \rightarrowtail R'$, and we prove that $u(R) \rightarrowtail u(R')$. If $X\ u(R)\ Y$, then $X = X_1 + X_2$ and $Y = Y_1 + Y_2$ for some X_1, X_2, Y_1, Y_2 such that $X_1\ R\ Y_1$ and $X_2\ R\ Y_2$. By assumption, we have $o^\sharp(X_1) = o^\sharp(Y_1)$, $o^\sharp(X_2) = o^\sharp(Y_2)$, and for all $a \in A$, $t_a^\sharp(X_1)\ R'\ t_a^\sharp(Y_1)$ and $t_a^\sharp(X_2)\ R'\ t_a^\sharp(Y_2)$. Since $o^\sharp$ and $t^\sharp$ are homomorphisms, we deduce $o^\sharp(X_1 + X_2) = o^\sharp(Y_1 + Y_2)$, and for all $a \in A$, $t_a^\sharp(X_1 + X_2)\ u(R')\ t_a^\sharp(Y_1 + Y_2)$. □

Theorem 2. *Any bisimulation up to congruence is contained in a bisimulation.*

Proof. By Proposition 3, it suffices to show that c is compatible, which follows from Lemmas 1, 2 and 3. □

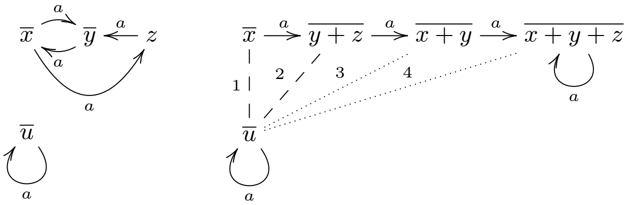


Figure 3. Bisimulations up to congruence, on a single letter NFA.

In the Introduction, we already gave an example of bisimulation up to context, which is a particular case of bisimulation up to congruence (up to context corresponds to use just the function $(r \cup u \cup \mathrm{id})^\omega$, without closing under s and t).

A more involved example illustrating the use of all ingredients of the congruence closure function (c) is given in Figure 3. The relation R expressed by the dashed numbered lines (formally $R = \{(x,\ u), (y + z,\ u)\}$) is neither a bisimulation, nor a bisimulation up to equivalence, since $y + z \overset{a}{\to} x + y$ and $u \overset{a}{\to} u$, but $(x{+}y,\ u) \notin e(R)$. However, R is a bisimulation up to congruence. Indeed, we have $(x + y,\ u) \in c(R)$:

$$\begin{aligned} x + y\ \ c(R)\ & u + y && ((x,u) \in R)\\ c(R)\ & y + z + y && ((y+z,u) \in R)\\ =\ \ & y + z && \\ c(R)\ & u && ((y+z,u) \in R) \end{aligned}$$

In contrast, we need four pairs to get a bisimulation up to e containing (x, u): this is the relation depicted with both dashed and dotted lines in Figure 3.

Note that we can deduce many other equations from R; in fact, $c(R)$ defines the following partition of sets of states:

$$\{0\}, \{y\}, \{z\}, \{x, u, x{+}y, x{+}z, \text{ and the 9 remaining subsets}\}.$$

3.3 Optimised algorithm for NFA

Algorithms for NFA can be obtained by computing the determinised NFA on-the-fly [12]: starting from the algorithms for DFA (Figure 1), it suffices to work with sets of states, and to inline the powerset construction. The corresponding code is given in Figure 4. The naive algorithm (`Naive`) does not use any up to technique, Hopcroft and Karp's algorithm (`HK`) reasons up to equivalence in step `3.2`, and the optimised algorithm, referred as `HKC` in the sequel, relies on up to congruence: step `3.2` becomes

```
(3.2)  if $(X', Y') \in c(R \cup todo)$ then skip;
```

Observe that we use $c(R \cup todo)$ rather than $c(R)$: this allows us to skip more pairs, and this is safe since all pairs in $todo$ will eventually be processed.

Corollary 2. *For all $X, Y \in \mathcal{P}(S)$, $X \sim Y$ iff* $\mathtt{HKC}(X, Y)$.

Proof. Same proof as for Proposition 2, by using the invariant $R \rightarrowtail c(R \cup todo)$ for the loop. We deduce that R is a bisimulation up to congruence after the loop. We conclude with Theorem 2 and Proposition 1. □

The most important point about these three algorithms is that they compute the states of the determinised NFA lazily. This means that only *accessible* states need to be computed, which is of practical importance since the determinised NFA can be exponentially large. In case of a negative answer, the three algorithms stop even before all accessible states have been explored; otherwise, if a bisimulation (possibly up-to) is found, it depends on the algorithm:

```
                    Naive$(X,Y)$
(1) $R$ is empty; $todo$ is empty;
(2) insert $(X,Y)$ in $todo$;
(3) while $todo$ is not empty, do {
 (3.1)  extract $(X',Y')$ from $todo$;
 (3.2)  if $(X',Y') \in R$ then skip;
 (3.3)  if $o^\sharp(X') \neq o^\sharp(Y')$ then return $false$;
 (3.4)  for all $a \in A$,
          insert $(t_a^\sharp(X'), t_a^\sharp(Y'))$ in $todo$;
 (3.5)  insert $(X',Y')$ in $R$;
(4) return $true$;
```

Figure 4. On-the-fly naive algorithm, for checking the equivalence of sets of states X and Y of a NFA (S, o, t). The code for on-the-fly $\mathtt{HK}(X,Y)$ is obtained by replacing the test in step `3.2` with $(X',Y') \in e(R)$; the code for $\mathtt{HKC}(X,Y)$ is obtained by replacing this test with $(X',Y') \in c(R \cup todo)$.

- with `Naive`, all accessible states need to be visited, by definition of bisimulation;
- with `HK`, the only case where some accessible states can be avoided is when a pair (X,X) is encountered: the algorithm skips this pair so that the successors of X are not necessarily computed (this situation rarely happens in practice—it actually never happens when starting with disjoint automata). In the other cases where a pair (X,Y) is skipped, then X and Y are necessarily already related to some other states in R, so that their successors will eventually be explored;
- with `HKC`, only a small portion of the accessible states is built (check the experiments in Section 6). To see a concrete example, let us execute `HKC` on the NFA from Figure 3. After two iterations, $R = \{(x,u),(y+z,u)\}$. Since $x+y\ c(R)\ u$, the algorithm stops without building the states $x+y$ and $x+y+z$. Similarly, in the example from the Introduction, `HKC` does not construct the four states corresponding to pairs 4, 5, and 6.

This ability of `HKC` to ignore parts of the determinised NFA comes from the up to congruence technique, which allows one to infer properties about states that were not necessarily encountered before. As we shall see in Section 4 the efficiency of antichains algorithms [1, 33] also comes from their ability to skip large parts of the determinised NFA.

3.4 Computing the congruence closure

For the optimised algorithm to be effective, we need a way to check whether some pairs belong to the congruence closure of some relation (step `3.2`). We present here a simple solution based on set rewriting; the key idea is to look at each pair (X,Y) in a relation R as a pair of rewriting rules:

$$X \to X+Y \qquad\qquad Y \to X+Y \ ,$$

which can be used to compute normal forms for sets of states. Indeed, by idempotence, $X\ R\ Y$ entails $X\ c(R)\ X+Y$.

Definition 7. *Let $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ be a relation on sets of states. We define $\rightsquigarrow_R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ as the smallest irreflexive relation that satisfies the following rules:*

$$\frac{X\ R\ Y}{X \rightsquigarrow_R X+Y} \qquad \frac{X\ R\ Y}{Y \rightsquigarrow_R X+Y} \qquad \frac{Z \rightsquigarrow_R Z'}{U+Z \rightsquigarrow_R U+Z'}$$

Lemma 4. *For all relations R, the relation $\rightsquigarrow_R$ is convergent.*

In the sequel, we denote by $X{\downarrow_R}$ the normal form of a set X w.r.t. $\rightsquigarrow_R$. Intuitively, the normal form of a set is the largest set of its equivalence class. Recalling the example from Figure 3, the common normal form of $x+y$ and u can be computed as follows (R is the relation $\{(x,u),(y+z,u)\}$):

$$\begin{tikzcd}[column sep=tiny]
x+y \arrow[dr, squiggly] & & & & u \arrow[dl, squiggly] \\
& x+y+u \arrow[dr, squiggly] & & x+u \arrow[dl, squiggly] & \\
& & x+y+z+u & &
\end{tikzcd}$$

Theorem 3. *For all relations R, and for all $X, Y \in \mathcal{P}(S)$, we have $X{\downarrow_R} = Y{\downarrow_R}$ iff $(X,Y) \in c(R)$.*

Thus, in order to check if $(X,Y) \in c(R \cup todo)$ we only have to compute the normal form of X and Y with respect to $\rightsquigarrow_{R \cup todo}$. Note that each pair of $R \cup todo$ may be used only once as a rewriting rule, but we do not know in advance in which order to apply these rules. Therefore, the time required to find one rule that applies is in the worst case rn where $r = |R \cup todo|$ is the size of the relation $R \cup todo$, and $n = |S|$ is the number of states of the NFA (assuming linear time complexity for set-theoretic union and containment of sets of states). Since we cannot apply more than r rules, the time for checking whether $(X,Y) \in c(R \cup todo)$ is bounded by $r^2 n$.

We tried other solutions, notably by using binary decision diagrams [8]. We have chosen to keep the presented rewriting algorithm for its simplicity and because it behaves well in practice.

3.5 Complexity hints

The complexity of `Naive`, `HK` and `HKC` is closely related to the size of the relation that they build. Hereafter, we use $v = |A|$ to denote the number of letters in A.

Lemma 5. *The three algorithms require at most $1 + v \cdot |R|$ iterations, where $|R|$ is the size of the produced relation; moreover, this bound is reached whenever they return true.*

Therefore, we can conveniently reason about $|R|$.

Lemma 6. *Let $R_{\mathtt{Naive}}$, $R_{\mathtt{HK}}$, and $R_{\mathtt{HKC}}$ denote the relations produced by the three algorithms. We have*

$$|R_{\mathtt{HKC}}|, |R_{\mathtt{HK}}| \leq m \qquad\qquad |R_{\mathtt{Naive}}| \leq m^2 \ , \tag{2}$$

where $m \leq 2^n$ is the number of accessible states in the determinised NFA and n is the number of states of the NFA. If the algorithms returned true, we moreover have

$$|R_{\mathtt{HKC}}| \leq |R_{\mathtt{HK}}| \leq |R_{\mathtt{Naive}}| \ . \tag{3}$$

As shown below in Section 4.2.4, $R_{\mathtt{HKC}}$ can be exponentially smaller than $R_{\mathtt{HK}}$. Notice however that the problem of deciding NFA language equivalence is PSPACE-complete [23], and that none of the algorithms presented here is in PSPACE: all of them store a set of visited pairs, and in the worst case, this set can become exponentially large with all of them. (This also holds for the antichain algorithms [1, 33] which we describe in Section 4.) Instead, the standard PSPACE algorithm does not store any set of visited pairs: it checks all words of length smaller than 2^n. While this can be done in polynomial space, this systematically requires exponential time.

3.6 Using `HKC` for checking language inclusion

For NFA, language inclusion can be reduced to language equivalence in a rather simple way. Since the function $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^\star}$ is a semi-lattice homomorphism (see Theorem 7 in Appendix A), for any given sets of states X and Y, $[\![X+Y]\!] = [\![Y]\!]$ iff $[\![X]\!] + [\![Y]\!] = [\![Y]\!]$ iff $[\![X]\!] \subseteq [\![Y]\!]$. Therefore, it suffices to run $\mathtt{HKC}(X+Y, Y)$ to check the inclusion $[\![X]\!] \subseteq [\![Y]\!]$.

In such a situation, all pairs that are eventually manipulated by `HKC` have the shape $(X'+Y', Y')$ for some sets X', Y'. The step `3.2` of `HKC`, where it checks whether the current pair belongs to the congruence closure of the relation, can thus be simplified. First, the pairs in the current relation can only be used to rewrite from right to left. Second, the following lemma allows one to avoid unnecessary normal form computations:

Lemma 7. *For all sets X, Y and for all relations R, we have $X{+}Y\ c(R)\ Y$ iff $X \subseteq Y{\downarrow_R}$.*

Proof. We first prove that for all $X, Y,\ X{\downarrow_R} = Y{\downarrow_R}$ iff $X \subseteq Y{\downarrow_R}$ and $Y \subseteq X{\downarrow_R}$, using the fact that the normalisation function ${\downarrow_R} \colon\ X \mapsto X{\downarrow_R}$ is monotone and idempotent. The announced result follows by Theorem 3, since $Y \subseteq (X+Y){\downarrow_R}$ is always true and $X{+}Y \subseteq Y{\downarrow_R}$ iff $X \subseteq Y{\downarrow_R}$. □

However, as shown below, checking an equivalence by decomposing it into two inclusions cannot be more efficient than checking the equivalence directly.

Lemma 8. *Let X, Y be two sets of states; let $R_\subseteq$ and $R_\supseteq$ be the relations computed by `HKC`$(X{+}Y, Y)$ and `HKC`$(X{+}Y, X)$, respectively. If $R_\subseteq$ and $R_\supseteq$ are bisimulations up to congruence, then the following relation is a bisimulation up to congruence:*

$$R_= = \{(X', Y') \mid (X'{+}Y', Y') \in R_\subseteq \text{ or } (X'{+}Y', X') \in R_\supseteq\}.$$

On the contrary, checking the equivalence directly actually allows one to skip some pairs that cannot be skipped when reasoning by double inclusion. As an example, consider the DFA on the right of Figure 2. The relation computed by `HKC`(x, u) contains only four pairs (because the fifth one follows from transitivity). Instead, the relations built by `HKC`$(x, x{+}u)$ and `HKC`$(u{+}x, u)$ would both contain five pairs: transitivity cannot be used since our relations are now oriented (from $y \le v$, $z \le v$ and $z \le w$, we cannot deduce $y \le w$). Another example, where we get an exponential factor by checking the equivalence directly rather than through the two inclusions, can be found in Section 4.2.4.

In a sense, the behaviour of the coinduction proof method here is similar to that of standard proofs by induction, where one often has to strengthen the induction predicate to get a (nicer) proof.

4. Antichain algorithm

In [33], De Wulf et al. have proposed the *antichain* approach for checking language inclusion of NFA. We show that this approach can be explained in terms of *simulations up to upward-closure* that, in turn, can be seen as a special case of bisimulations up to congruence. Before doing so, we recall the standard notion of antichain and we describe the antichain algorithm (`AC`).

Given a partial order $(X, \sqsubseteq)$, an *antichain* is a subset $Y \subseteq X$ containing only incomparable elements (that is, for all $y_1, y_2 \in Y$, $y_1 \not\sqsubseteq y_2$ and $y_2 \not\sqsubseteq y_1$). `AC` exploits antichains over the set $S \times \mathcal{P}(S)$, where the ordering is given by $(x_1, Y_1) \sqsubseteq (x_2, Y_2)$ iff $x_1 = x_2$ and $Y_1 \subseteq Y_2$.

In order to check $[\![X]\!] \subseteq [\![Y]\!]$ for two sets of states X, Y of an NFA (S, o, t), `AC` maintains an antichain of pairs (x', Y'), where x' is a state of the NFA and Y' is a state of the determinised automaton. More precisely, the automaton is explored non-deterministically (via t) for obtaining the first component of the pair and deterministically (via $t^\sharp$) for the second one. If a pair such that x' is accepting ($o(x') = 1$) and Y' is not ($o^\sharp(Y') = 0$) is encountered, then a counter-example has been found. Otherwise all derivatives of the pair along the automata transitions have to be inserted into the antichain, so that they will be explored. If one these pairs p is larger than a previously encountered pair p' ($p' \sqsubseteq p$) then the language inclusion corresponding to p is subsumed by p' so that p can be skipped; otherwise, if $p \sqsubseteq p_1, \dots, p_n$ for some pairs $p_1, \dots, p_n$ that are already in the antichain, then one can safely remove these pairs: they are subsumed by p and, by doing so, the set of visited pairs remains an antichain.

Remark 4. *An important difference between* `HKC` *and* `AC` *consists in the fact that the former inserts pairs in todo without checking whether they are redundant (this check is performed when the pair is processed), while the latter removes all redundant pairs whenever a new one is inserted. Therefore, the cost of an iteration with* `HKC` *is merely the cost of the corresponding congruence check, while the cost of an iteration with* `AC` *is merely that of inserting all successors of the corresponding pair and simplifying the antichain.*

Note that the above description corresponds to the “forward” antichain algorithm, as described in [1]. Instead, the original antichain algorithm, as first described in [33], is “backward” in the sense that the automata are traversed in the reversed way, from accepting states to initial states. The two versions are dual [33] and we could similarly define the backward counterpart of `HKC` and `HK`. We however stick to the forward versions for the sake of clarity.

4.1 Coinductive presentation

Leaving apart the concrete data structures used to manipulate antichains, we can rephrase this algorithm using a coinductive framework, like we did for Hopcroft and Karp’s algorithm.

First define a notion of *simulation*, where the left-hand side automaton is executed non-deterministically:

Definition 8 (Simulation)**.** *Given two relations $T, T' \subseteq S \times \mathcal{P}(S)$, T* s-progresses *to T', denoted $T \rightarrowtail_s T'$, if whenever $x\ T\ Y$ then*

1. *$o(x) \le o^\sharp(Y)$ and*
2. *for all $a \in A$, $x' \in t_a(x)$, $x'\ T'\ t^\sharp_a(Y)$.*

A simulation *is a relation T such that $T \rightarrowtail_s T$.*

As expected, we obtain the following coinductive proof principle:

Proposition 4 (Coinduction)**.** *For all sets X, Y, we have $[\![X]\!] \subseteq [\![Y]\!]$ iff there exists a simulation T such that for all $x \in X$, $x\ T\ Y$.*

(Note that like for our notion of bisimulation, the above notion of simulation is weaker than the standard one from concurrency theory [24], which *strictly* entails language inclusion—Remark 3.)

To account for the antichain algorithm, where we can discard pairs using the preorder $\sqsubseteq$, it suffices to define the *upward closure* function $\uparrow \colon \mathcal{P}(S \times \mathcal{P}(S)) \to \mathcal{P}(S \times \mathcal{P}(S))$ as

$$\uparrow\! T = \{(x, Y) \mid \exists (x', Y') \in T \text{ s.t. } (x', Y') \sqsubseteq (x, Y)\} \ .$$

A pair belongs to the upward closure $\uparrow\! T$ of a relation $T \subseteq S \times \mathcal{P}(S)$, if and only if this pair is subsumed by some pair in T. In fact, rather than trying to construct a simulation, `AC` attempts to construct a simulation up to upward closure.

Like for `HK` and `HKC`, this method can be justified by defining the appropriate notion of s-compatible function, showing that any simulation up to an s-compatible function is contained in a simulation, and showing that the upward closure function ($\uparrow$) is s-compatible.

Theorem 4. *Any simulation up to $\uparrow$ is contained in a simulation.*

Corollary 3. *For all $X, Y \in \mathcal{P}(S)$, $[\![X]\!] \subseteq [\![Y]\!]$ iff* `AC`(X, Y).

4.2 Comparing `HKC` and `AC`

The efficiency of the two algorithms strongly depends on the number of pairs that they need to explore. In the following (Sections 4.2.3 and 4.2.4), we show that `HKC` can explore far fewer pairs than `AC`, when checking language inclusion of automata that share some states, or when checking language equivalence. We would also like to formally prove that (a) `HKC` never explores more than `AC`, and

(b) when checking inclusion of disjoint automata, `AC` never explores more than `HKC`. Unfortunately, the validity of these statements highly depends on numerous assumptions about the two algorithms (e.g., on the exploration strategy) and their potential proofs seem complicated and not really informative. For these reasons, we preferred to investigate the formal correspondence at the level of the coinductive proof techniques, where it is much cleaner.

4.2.1 Language inclusion: `HKC` can mimic `AC`

As explained in Section 3.6, we can check the language inclusion of two sets X, Y by executing $\texttt{HKC}(X{+}Y, Y)$. We now show that for any simulation up to upward closure that proves the inclusion $[\![X]\!] \subseteq [\![Y]\!]$, there exists a bisimulation up to congruence of the same size which proves the same inclusion. For $T \subseteq S \times \mathcal{P}(S)$, let $\widehat{T} \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ denote the relation $\{(x+Y, Y) \mid x\ T\ Y\}$.

Lemma 9. *We have* $\widehat{\uparrow T} \subseteq c(\widehat{T})$.

Proof. If $(x+Y, Y) \in \widehat{\uparrow T}$, then there exists $Y' \subseteq Y$ such that $(x, Y') \in T$. By definition, $(x+Y', Y') \in \widehat{T}$ and $(Y, Y) \in c(\widehat{T})$. By the rule (u), $(x+Y'+Y, Y'+Y) \in c(\widehat{T})$ and since $Y' \subseteq Y$, $(x+Y, Y) \in c(\widehat{T})$. □

Proposition 5. *If T is a simulation up to $\uparrow$, then $\widehat{T}$ is a bisimulation up to c.*

Proof. First observe that if $T \rightarrowtail_s T'$, then $\widehat{T} \rightarrowtail u^\omega(\widehat{T'})$. Therefore, if $T \rightarrowtail_s \uparrow T$, then $\widehat{T} \rightarrowtail u^\omega(\widehat{\uparrow T})$. By Lemma 9, $\widehat{T} \rightarrowtail u^\omega(c(\widehat{T})) = c(\widehat{T})$. □

(Note that transitivity and symmetry are not used in the above proofs: the constructed bisimulation up to congruence is actually a bisimulation up to context $(r \cup u \cup id)^\omega$.)

The relation $\widehat{T}$ is not the one computed by `HKC`, since the former contains pairs of the shape $(x+Y, Y)$, while the latter has pairs of the shape $(X+Y, Y)$ with X possibly not a singleton. However, note that manipulating pairs of the two kinds does not change anything since by Lemma 7, $(X+Y, Y) \in c(R)$ iff for all $x \in X$, $(x+Y, Y) \in c(R)$.

4.2.2 Inclusion: `AC` can mimic `HKC` on disjoint automata

As shown in Section 4.2.3 below, `HKC` can be faster than `AC`, thanks to the up to transitivity technique. However, in the special case where the two automata are disjoint, transitivity cannot help, and the two algorithms actually match each other.

Suppose that the automaton (S, o, t) is built from two disjoint automata (S_1, o_1, t_1) and (S_2, o_2, t_2) as described in Remark 1. Let R be the relation obtained by running $\texttt{HKC}(X_0{+}Y_0, Y_0)$ with $X_0 \subseteq S_1$ and $Y_0 \subseteq S_2$. All pairs in R are necessarily of the shape $(X{+}Y,\ Y)$ with $X \subseteq S_1$ and $Y \subseteq S_2$. Let $\overline{R} \subseteq S \times \mathcal{P}(S)$ denote the relation $\{(x, Y) \mid \exists X,\ x \in X \text{ and } X{+}Y\ R\ Y\}$.

Lemma 10. *If S_1 and S_2 are disjoint, then* $\overline{c(R)} \subseteq \uparrow(\overline{R})$.

Proof. Suppose that $x\ \overline{c(R)}\ Y$, i.e., $x \in X$ with $X + Y\ c(R)\ Y$. By Lemma 7, we have $X \subseteq Y{\downarrow_R}$, and hence, $x \in Y{\downarrow_R}$. By definition of R the pairs it contains can only be used to rewrite from right to left; moreover, since S_1 and S_2 are disjoint, such rewriting steps cannot enable new rewriting rules, so that all steps can be performed in parallel: we have $Y{\downarrow_R} = \sum_{X'+Y' R Y' \subseteq Y} X'$. Therefore, there exists some X', Y' with $x \in X'$, $X'{+}Y'\ R\ Y'$, and $Y' \subseteq Y$. It follows that $(x, Y') \in \overline{R}$, hence $(x, Y) \in \uparrow(\overline{R})$. □

Proposition 6. *If S_1 and S_2 are disjoint, and if R is a bisimulation up to congruence, then $\overline{R}$ is a simulation up to upward closure.*

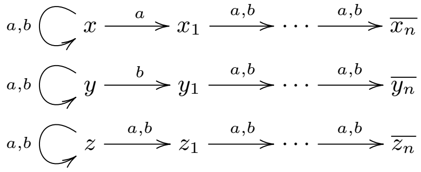


Figure 5. Family of examples where `HKC` exponentially improves over `AC` and `HK`; we have $x + y \sim z$.

Proof. First observe that for all relations R, R', if $R \rightarrowtail R'$, then $\overline{R} \rightarrowtail_s \overline{R'}$. Therefore, if $R \rightarrowtail c(R)$, then $\overline{R} \rightarrowtail_s \overline{c(R)}$. We deduce $\overline{R} \rightarrowtail_s \uparrow(\overline{R})$ by Lemma 10. □

4.2.3 Inclusion: `AC` cannot mimic `HKC` on merged automata

The containment of Lemma 10 does not hold when S_1 and S_2 are not disjoint, since c can exploit transitivity, while $\uparrow$ cannot. For a concrete grasp, take $R = \{(x+y, y), (y+z, z)\}$ and observe that $(x, z) \in \overline{c(R)}$ but $(x, z) \notin \uparrow(\overline{R})$. This difference makes it possible to find bisimulations up to c that are much smaller than the corresponding simulations up to $\uparrow$, and for `HKC` to be more efficient than `AC`. Such an example, where `HKC` is exponentially better than `AC` for checking language inclusion of automata sharing some states, is given in [6].

4.2.4 Language equivalence: `AC` cannot mimic `HKC`.

`AC` can be used to check language equivalence, by checking the two underlying inclusions. However, checking equivalence directly can be better, even in the disjoint case. To see this on a simple example, consider the DFA on the right-hand side of Figure 2. If we use `AC` twice to prove $x \sim u$, we get the following antichains

$$T_1 = \{(x,u), (y,v), (y,w), (z,v), (z,w)\} \ ,$$
$$T_2 = \{(u,x), (v,y), (w,y), (v,z), (w,z)\} \ ,$$

containing five pairs each. Instead, four pairs are sufficient with `HK` or `HKC`, thanks to up to symmetry and up to transitivity.

For a more interesting example, consider the family of NFA given in Figure 5, where n is an arbitrary natural number. Taken together, the states x and y are equivalent to the state z: they recognise the language $(a{+}b)^\star(a{+}b)^{n+1}$. Alone, the state x (resp. y) recognises the language $(a{+}b)^\star a(a{+}b)^n$ (resp. $(a{+}b)^\star b(a{+}b)^n$).

For $i \leq n$, let $X_i = x{+}x_1{+}\ldots{+}x_i$, $Y_i = y{+}y_1{+}\ldots{+}y_i$, and $Z_i = z{+}z_1{+}\ldots{+}z_i$; for $N \subseteq [1..i]$, furthermore set

$$X_i^N = x + \sum_{j \in N} x_j \ , \qquad \overline{Y}_i^N = y + \sum_{j \in [1..n] \setminus N} y_j \ .$$

In the determinised NFA, $x + y$ can reach all the states of the shape $X_i^N {+} \overline{Y}_i^N$, for $i \leq n$ and $N \subseteq [1..i]$. For instance, for $n{=}i{=}2$, we have $x{+}y \overset{aa}{\to} x{+}y{+}x_1{+}x_2$, $x{+}y \overset{ab}{\to} x{+}y{+}y_1{+}x_2$, $x{+}y \overset{ba}{\to} x{+}y{+}x_1{+}y_2$, and $x{+}y \overset{bb}{\to} x{+}y{+}y_1{+}y_2$. Instead, z reaches only $n{+}1$ distinct states, those of the form Z_i.

The smallest bisimulation relating $x + y$ and z is

$$R = \{(X_i^N + \overline{Y}_i^N,\ Z_i) \mid i \leq n, N \subseteq [1..i]\},$$

which contains $2^{n+1}{-}1$ pairs. This is the relation computed by $\texttt{Naive}(x, y)$ and $\texttt{HK}(x, y)$—the up to equivalence technique (alone) does not help in `HK`. With `AC`, we obtain the antichains $T_x + T_y$ (for

$[\![x+y]\!] \subseteq [\![z]\!]$) and T_z (for $[\![x+y]\!] \supseteq [\![z]\!]$), where:

$$\begin{aligned} T_x &= \{(x_i,\ Z_i) \mid i \leq n\}, \\ T_y &= \{(y_i,\ Z_i) \mid i \leq n\}, \\ T_z &= \{(z_i,\ X_i^N + \overline{Y}_i^N) \mid i \leq n, N \subseteq [1..i]\}. \end{aligned}$$

Note that T_x and T_y have size $n+1$, and T_z has size $2^{n+1}-1$.

The language recognised by x or y are known for having a minimal DFA with 2^n states [17]. So, checking $x+y \sim z$ via minimisation (e.g., [9, 15]) would also require exponential time.

This is not the case with HKC, which requires only polynomial time in this case. Indeed, HKC$(x+y, z)$ builds the relation

$$\begin{aligned} R' = \{&(x+y,\ z)\} \\ \cup\, \{&(x+Y_i+y_{i+1},\ Z_{i+1}) \mid i < n\} \\ \cup\, \{&(x+Y_i+x_{i+1},\ Z_{i+1}) \mid i < n\} \end{aligned}$$

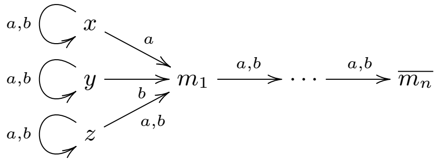
which is a bisimulation up to congruence and which only contains $2n+1$ pairs. To see that this is a bisimulation up to congruence, consider the pair $(x{+}y{+}x_1{+}y_2,\ Z_2)$ obtained from $(x{+}y,\ z)$ after reading the word ba. This pair does not belong to R' but to its congruence closure. Indeed, we have

$$\begin{aligned} x{+}y{+}x_1{+}y_2\ c(R')\ & Z_1{+}y_2 && (x{+}y{+}x_1\ R'\ Z_1) \\ c(R')\ & x{+}y{+}y_1{+}y_2 && (x{+}y{+}y_1\ R'\ Z_1) \\ c(R')\ & Z_2 && (x{+}y{+}y_1{+}y_2\ R'\ Z_2) \end{aligned}$$

(Check Lemma 18 in Appendix D for a complete proof.)

5. Exploiting Similarity

Looking at the example in Figure 5, a natural idea would be to first quotient the automaton by graph isomorphism. By doing so, we would merge the states x_i, y_i, z_i, and we would obtain the following automaton, for which checking $x{+}y \sim z$ is much easier.



As shown by Abdulla et al. [1], one can actually do better with the antichain algorithm, by exploiting any preorder contained in language inclusion (e.g., similarity [24]). In this section, we rephrase this technique for antichains in our coinductive framework, and we show how this idea can be embedded in HKC, resulting in an even stronger algorithm.

5.1 AC with similarity: AC'

For the sake of clarity, we fix the preorder to be *similarity*, which can be computed in quadratic time [13]:

Definition 9 (Similarity)**.** Similarity *is the largest relation on states* $\preceq\ \subseteq S \times S$ *such that* $x \preceq y$ *entails:*

1. $o(x) \leq o(y)$ *and*
2. *for all* $a \in A, x' \in S$ *such that* $x \overset{a}{\to} x'$, *there exists some* y' *such that* $y \overset{a}{\to} y'$ *and* $x' \preceq y'$.

One extends similarity to a preorder $\preceq^{\forall\exists}\ \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ on sets of states, and to a preorder $\sqsubseteq^{\preceq}\ \subseteq (S \times \mathcal{P}(S)) \times (S \times \mathcal{P}(S))$ on antichain pairs, as:

$$\begin{aligned} X \preceq^{\forall\exists} Y \quad & \text{if} \quad \forall x \in X, \exists y \in Y, x \preceq y \ \ , \\ (x', Y') \sqsubseteq^{\preceq} (x, Y) \quad & \text{if} \quad x \preceq x' \text{ and } Y' \preceq^{\forall\exists} Y \ \ . \end{aligned}$$

The new antichain algorithm [1], which we call AC', is similar to AC, but the antichain is now taken w.r.t. the new preorder $\sqsubseteq^{\preceq}$. Formally, let $\curlywedgeuparrow\colon \mathcal{P}(S \times \mathcal{P}(S)) \to \mathcal{P}(S \times \mathcal{P}(S))$ be the function defined for all relations $T \subseteq S \times \mathcal{P}(S)$, as

$$\begin{aligned} \curlywedgeuparrow T = \{(x, Y) \mid x \preceq^{\forall\exists} Y, \text{ or} & \\ \exists (x', Y') \in T \text{ s.t. } (x', Y') \sqsubseteq^{\preceq} (x, Y)\}. & \end{aligned}$$

While AC consists in trying to build a simulation up to $\uparrow$, AC' tries to build a simulation up to $\curlywedgeuparrow$, i.e., it skips a pair (x, Y) if either (a) it is subsumed by another pair of the antichain or (b) $x \preceq^{\forall\exists} Y$.

Theorem 5. *Any simulation up to* $\curlywedgeuparrow$ *is contained in a simulation.*

Corollary 4. *The antichain algorithm proposed in [1] is sound and complete: for all sets* X, Y, $[\![X]\!] \subseteq [\![Y]\!]$ *iff* AC'(X, Y).

Optimisation 1(a) and optimisation 1(b) in [1] are simply (a) and (b), as discussed above. Another optimisation, called Optimisation 2, is presented in [1]: if $y_1 \preceq y_2$ and $y_1, y_2 \in Y$ for some pair (x, Y), then y_1 can be safely removed from Y. Note that while this is useful to store smaller sets, it does not allow one to explore less, since the pairs encountered with or without optimisation 2 are always equivalent w.r.t. the ordering $\sqsubseteq^{\preceq}$: $Y \preceq^{\forall\exists} Y \setminus y_1$ and, for all $a \in A$, $t_a^\sharp(Y) \preceq^{\forall\exists} t_a^\sharp(Y \setminus y_1)$.

5.2 HKC with similarity: HKC'

Although HKC is primarily designed to check language equivalence, we can also extend it to exploit the similarity preorder. It suffices to notice that for any similarity pair $x \preceq y$, we have $x{+}y \sim y$.

Let $\overline{\preceq}$ denote the relation $\{(x{+}y,\ y) \mid x \preceq y\}$, let r' denote the constant to $\overline{\preceq}$ function, and let $c' = (r' {\cup} s {\cup} t {\cup} u {\cup} id)^\omega$. Accordingly, we call HKC' the algorithm obtained from HKC (Figure 4) by replacing $(X, Y) \in c(R \cup \mathit{todo})$ with $(X, Y) \in c'(R \cup \mathit{todo})$ in step 3.2. Notice that the latter test can be reduced to rewriting thanks to Theorem 3 and the following lemma.

Lemma 11. *For all relations R,* $c'(R) = c(R \cup \overline{\preceq})$.

In other words to check whether $(X, Y) \in c'(R \cup \mathit{todo})$, it suffices to compute the normal forms of X and Y w.r.t. the rules from $R \cup \mathit{todo}$ plus the rules $x + y \leftarrow y$ for all $x \preceq y$.

Theorem 6. *Any bisimulation up to* c' *is contained in a bisimulation.*

Proof. Consider the constant function $r''\colon \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S)) \to \mathcal{P}(\mathcal{P}(S) {\times} \mathcal{P}(S))$ mapping all relations to $\sim$. Since language equivalence $(\sim)$ is a bisimulation, we immediately obtain that this function is compatible. Thus so is the function $c'' = (r'' {\cup} s {\cup} t {\cup} u {\cup} id)^\omega$. We have that $\overline{\preceq}$ is contained in $\sim$, so that any bisimulation up to c' is a bisimulation up to c''. Since c'' is compatible, such a relation is contained in a bisimulation, by Proposition 3. □

Note that in the above proof, we can replace $\overline{\preceq}$ by any other relation contained in $\sim$. Intuitively, bisimulations up to c'' correspond to classical *bisimulations up to bisimilarity* [24] from concurrency.

Corollary 5. *For all sets* X, Y, *we have* $X \sim Y$ *iff* HKC'(X, Y).

5.3 Relationship between HKC' and AC'

Like in Section 4.2.1, we can show that for any simulation up to $\curlywedgeuparrow$ there exists a corresponding bisimulation up to c', of the same size.

Lemma 12. *For all relations* $T \subseteq S \times \mathcal{P}(S)$, $\widehat{\curlywedgeuparrow T} \subseteq c'(\widehat{T})$.

Proposition 7. *If* T *is a simulation up to* $\curlywedgeuparrow$, *then* $\widehat{T}$ *is a bisimulation up to* c'.

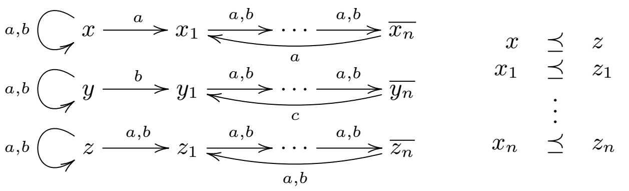


Figure 6. Family of examples where HKC' exponentially improves over AC', for inclusion of disjoint automata: we have $[\![z]\!] \subseteq [\![x+y]\!]$.

However, even for checking inclusion of disjoint automata, AC' cannot mimic HKC', because now the similarity relation allows one to exploit transitivity. To see this, consider the example given in Figure 6, where we want to check that $[\![z]\!] \subseteq [\![x+y]\!]$, and for which the similarity relation is shown on the right-hand side.

Since this is an inclusion of disjoint automata, HKC and AC, which do not exploit similarity, behave the same (cf. Sections 4.2.1 and 4.2.2). Actually, they also behave like HK and they require $2^{n+1}-1$ pairs. On the contrary, the use of similarity allows HKC' to prove the inclusion with only $2n+1$ pairs, by computing the following bisimulation up to c' (Lemma 19 in Appendix E):

$$\begin{aligned} R'' = \{&(z{+}x{+}y,\ x{+}y)\} \\ \cup\, \{&(Z_{i+1}{+}X_i{+}y{+}y_{i+1},\ X_i{+}y{+}y_{i+1}) \mid i<n\} \\ \cup\, \{&(Z_{i+1}{+}X_{i+1}{+}y,\ X_{i+1}{+}y) \mid i<n\}\ , \end{aligned}$$

where $X_i = x{+}x_1{+}\ldots{+}x_i$ and $Z_i = z{+}z_1{+}\ldots{+}z_i$.

Like in Section 4.2.4, to see that this is a bisimulation up to c' (where we do exploit similarity), consider the pair obtained after reading the word ab: $(Z_2{+}x{+}y{+}x_2{+}y_1,\ x{+}y{+}x_2{+}y_1)$. This pair does not belong to R'' or $c(R'')$, but it does belong to $c'(R'')$. Indeed, by Lemmas 7 and 11, this pair belong to $c'(R'')$ iff $Z_2 \subseteq (x{+}y{+}x_2{+}y_1){\downarrow_{R''\cup\preceq}}$, and we have

$$\begin{aligned} & x{+}y{+}x_2{+}y_1 \\ \rightsquigarrow_{R''\cup\preceq}\ & Z_1{+}x{+}y{+}y_1{+}x_2 && (Z_1{+}x{+}y{+}y_1\ R''\ x{+}y{+}y_1) \\ \rightsquigarrow_{R''\cup\preceq}\ & Z_1{+}X_1{+}y{+}y_1{+}x_2 = Z_1{+}X_2{+}y{+}y_1 && (x_1 \preceq z_1) \\ \rightsquigarrow_{R''\cup\preceq}\ & Z_2{+}X_2{+}y{+}y_1{+}x_2 && (Z_2{+}X_2{+}y\ R''\ X_2{+}y) \end{aligned}$$

On the contrary, AC' is not able to exploit similarity in this case, and it behaves like AC: both of them compute the same antichain T_z as in the example from Section 4.2.4, which has $2^{n+1}-1$ elements.

In fact, even when considering inclusion of disjoint automata, the use of similarity tends to virtually merge states, so that HKC' can use the up to transitivity technique which AC and AC' lack.

5.4 A short recap

Figure 7 summarises the relationship amongst the presented algorithms, in the general case and in the special case of language inclusion of disjoint automata. In this diagram, an arrow X→Y (from an algorithm X to Y) means that (a) Y can explore less states than X, and (b) Y can mimic X, i.e., the proof technique of Y is at least as powerful as the one of X. (The labels on the arrows point to the sections showing these relations; unlabelled arrows are not illustrated in this paper, they are easily inferred from what we have shown.)

6. Experimental assessment

To get an intuition of the average behaviour of HKC on various NFA, and to compare it with HK and AC, we provide some benchmarks on random automata and on automata obtained from model-checking problems. In both cases, we conduct the experiments on a MacBook pro 2.4GHz Intel Core i7, with 4GB of memory, running OS X Lion (10.7.4). We use our OCaml implementation for HK, HKC, and HKC' [6], and the libvata C++ library for AC and AC' [20]. (To our knowledge, libvata is the most efficient implementation currently available for the antichain algorithms.)

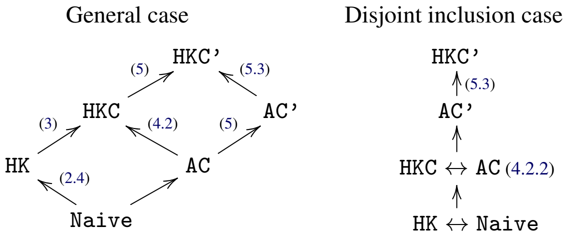


Figure 7. Relationship between the various algorithms.

6.1 Random automata

For a given size n, we generate a thousand random NFA with n states and two letters. According to [31], we use a linear transition density of 1.25 (which means that the expected out-degree of each state and with respect to each letter is 1.25): Tabakov and Vardi empirically showed that one statistically gets more challenging NFA with this particular value. We generate NFA without accepting states: by doing so, we make sure that the algorithms never encounter a counter-example, so that they always continue until they find a (bi)simulation up to: these runs correspond to their worst cases for all possible choices of accepting states for the given NFA.[2]

We run all algorithms on these NFA, starting from two distinct singleton sets, to measure the required time and the number of processed pairs: for HK, HKC, and HKC', this is the number of pairs put into the bisimulation up to (R); for AC and AC', this is the number of pairs inserted into the antichain. The timings for HKC' and AC' do not include the time required to compute similarity.

We report the median values (50%), the last deciles (90%), the last percentiles (99%), and the maximum values (100%) in Table 1. For instance, for $n = 70$, 90% of the examples require less than 155ms with HK; equivalently, 10% of the examples require more than 155ms. (For a few tests, libvata ran out of memory, whence the ∞ symbols in the table.) We also plotted on Figure 8 the distribution of the number of processed pairs when $n = 100$.

HKC and AC are several orders of magnitude better than HK, and HKC is usually two to ten times faster than AC. Moreover, for the first four lines, HKC is much more predictable than AC, i.e., the last percentiles and maximal values are of the same order as the median value. (AC seems to become more predictable for larger values of n.) The same relative behaviour can be observed between HKC' and AC'; moreover, HKC alone is apparently faster than AC'.

Also recall that the size of the relations generated by HK is a lower bound for the number of accessible states of the determinised NFA (Lemma 6 (2)); one can thus see in Table 1 that HKC usually explores an extremely small portion of these DFA (e.g., less than one per thousand for $n = 100$). The last column reports the median size of the minimal DFA for the corresponding parameters, as given in [31]. HK usually explores much many states than what would be necessary with a minimal DFA, while HKC and AC need much less.

6.2 Automata from model-checking

Checking language inclusion of NFA can be useful for model-checking, where one sometimes has to compute a sequence of NFA

[2] To get this behaviour for AC and AC', we actually had to trick libvata, which otherwise starts by removing non-coaccessible states, and thus reduces any of these NFA to the empty one.

		required time (seconds)				number of processed pairs				mDFA size
$n = \|S\|$	algo.	50%	90%	99%	100%	50%	90%	99%	100%	50%
50	HK	0.007	0.022	0.050	0.119	2511	6299	12506	25272	∼1000
	AC	0.002	0.003	0.142	1.083	112	245	2130	5208	
	HKC	0.000	0.000	0.000	0.000	21	26	32	63	
	AC’	0.002	0.002	0.038	0.211	79	131	1098	1926	
	HKC’	0.000	0.000	0.000	0.000	18	23	28	58	
70	HK	0.047	0.155	0.413	0.740	10479	28186	58782	87055	∼6000
	AC	0.002	0.003	1.492	4.163	150	285	8383	15575	
	HKC	0.000	0.000	0.000	0.000	27	34	40	49	
	AC’	0.002	0.003	0.320	0.884	110	172	3017	6096	
	HKC’	0.000	0.000	0.000	0.000	23	29	36	44	
100	HK	0.373	1.207	3.435	5.660	58454	164857	361227	471727	∼30000
	AC	0.003	0.004	3.214	36.990	204	298	13801	48059	
	HKC	0.000	0.000	0.000	0.001	36	44	54	70	
	AC’	0.003	0.004	0.738	6.966	152	211	4087	18455	
	HKC’	0.000	0.000	0.000	0.001	31	39	46	64	
300	AC	0.009	0.010	0.028	0.750	562	622	2232	14655	–
	HKC	0.001	0.002	0.003	0.009	86	104	118	132	
	AC’	0.012	0.013	0.022	0.970	433	484	920	14160	
	HKC’	0.001	0.001	0.002	0.006	76	91	104	116	
500	AC	0.014	0.015	0.039	∞	918	986	2571	∞	–
	HKC	0.002	0.005	0.008	0.018	130	154	176	193	
	AC’	0.025	0.028	0.042	∞	710	772	1182	∞	
	HKC’	0.002	0.004	0.007	0.013	115	136	154	169	
1000	AC	0.029	0.031	0.038	∞	1808	1878	2282	∞	–
	HKC	0.007	0.022	0.055	0.093	228	271	304	337	
	AC’	0.074	0.080	0.092	∞	1409	1488	1647	∞	
	HKC’	0.008	0.019	0.041	0.077	202	238	265	299	

Table 1. Running the five presented algorithms to check language equivalence on random NFA with two letters.

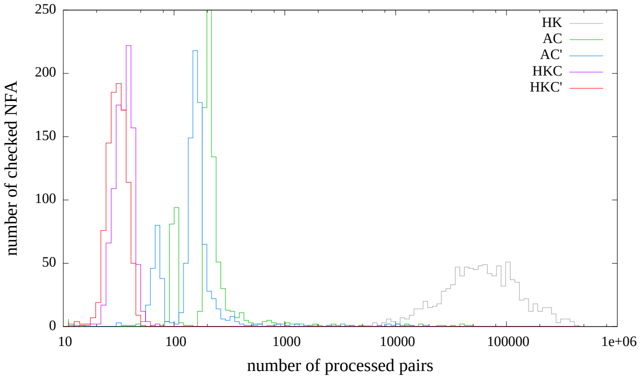


Figure 8. Distributions of the number of processed pairs, for the 1000 NFA with 100 states and 2 letters from Table 1.

by iteratively applying a transducer, until a fixpoint is reached [7]. To know that the fixpoint is reached, one typically has to check whether an NFA is contained in another one.

Abdulla et al. [1] use such benchmarks to test their algorithm (AC’) against the plain antichain algorithm (AC [33]). We reuse them to test HKC’ against AC’ in a concrete scenario. We take the sequences of automata kindly provided by L. Holik, which roughly corresponds to those used in [1] and which come from the model checking of various programs (the bakery algorithm, bubble sort, and a producer-consumer system). For all these sequences, we check the inclusions of consecutive pairs, in both directions. We separate the results into those for which a counter-example is found, and those for which the inclusion holds. We skip the trivial inclusions which hold by similarity ($\preceq^{\forall\exists}$), and for which both HKC’ and AC’ stop immediately.

The results are given in Table 2. Even though these are inclusions of disjoint automata, HKC’ is faster than AC’ on these examples: up to transitivity can be exploited thanks to the similarity pairs, and larger parts of the determinised NFA can be skipped.

7. Related work

A similar notion of bisimulation up to congruence has already been used to obtain decidability and complexity results about context-free processes, under the name of *self-bisimulations*. Caucal [10] introduced this concept to give a shorter and nicer proof of the result by Baeten et al. [4]: bisimilarity is decidable for normed context-free processes. Christensen et al [11] then generalised the result to all context-free processes, also by using self-bisimulations. Hirshfeld et al. [14] used a refinement of this notion to get a polynomial algorithm for bisimilarity in the normed case.

There are two main differences with the ideas we presented here. First, the above papers focus on bisimilarity rather than language equivalence (recall that although we use bisimulation relations, we check language equivalence since we work on the determinised NFA—Remark 3). Second, we consider a notion of bisimulation up to congruence where the congruence is taken with respect to non-determinism (union of sets of states). Self-bisimulations are also bisimulations up to congruence, but the congruence is taken with respect to word concatenation. We cannot consider this operation in our setting since we do not have the corresponding monoid structure in plain NFA.

Other approaches, that are independent from the algebraic structure (e.g., monoids or semi-lattices) and the behavioural equivalence (e.g., bisimilarity or language equivalence) are shown in [5, 21, 22, 26]. These propose very general frameworks into which our up to congruence technique fits as a very special case. To our knowledge, bisimulation up to congruence has never been proposed as a technique for proving language equivalence of NFA.

		required time (seconds)				number of processed pairs				number of tests
result	algo.	50%	90%	99%	100%	50%	90%	99%	100%	
counter-example	AC'	0.012	0.107	1.047	1.134	23	247	598	1352	518
	HKC'	0.001	0.005	0.025	0.383	11	24	112	290	
inclusion holds	AC'	0.079	0.795	1.457	1.480	149	733	1854	3087	178
	HKC'	0.015	0.165	0.340	0.345	61	695	1076	1076	

Table 2. Running HKC' and AC' to test language inclusion of disjoint NFA generated from model-checking.

8. Conclusions and future work

We showed that the standard algorithm by Hopcroft and Karp for checking language equivalence of DFA relies on a bisimulation up to equivalence proof technique; this allowed us to design a new algorithm (HKC) for the non-deterministic case, where we exploit a novel technique called up to congruence.

We then compared HKC to the recently introduced antichain algorithms [33] (AC): when checking the inclusion of disjoint automata, the two algorithms are equivalent, in all the other cases HKC is more efficient since it can use transitivity to prune a larger portion of the state-space.

The difference between these two approaches becomes even more striking when considering some optimisation exploiting similarity. Indeed, as nicely shown with AC' [1], the antichains approach can widely benefit from the knowledge one gets by first computing similarity. Inspired by this work, we showed that both our proof technique (bisimulation up to congruence) and our algorithm (HKC) can be easily modified to exploit similarity. The resulting algorithm (HKC') is now more efficient than AC' even for checking language inclusion of disjoint automata.

We provided concrete examples where HKC and HKC' are exponentially faster than AC and AC' (Sections 4.2.4 and 5.3) and we proved that the coinductive techniques underlying the formers are at least as powerful as those exploited by the latters (Propositions 5 and 7). We finally compared the algorithms experimentally, by running them on both randomly generated automata, and automata resulting from model checking problems. It appears that for these examples, HKC and HKC' perform better than AC and AC'.

Finally note that our implementation of the presented algorithms is available online [6], together with an applet making it possible to test them on user-provided examples.

As future work, we plan to extend our approach to tree automata. In particular, it seems promising to investigate if further up-to techniques can be defined for regular tree expressions. For instance, the algorithms proposed in [3, 18] exploit some optimisation which suggest us coinductive up-to techniques.

References

[1] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *Proc. TACAS*, vol. 6015 of *LNCS*, pages 158–174. Springer, 2010.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In *FPCA*, vol. 523 of *LNCS*, pages 427–447. Springer, 1991.

[4] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. In *Proc. PARLE (II)*, vol. 259 of *LNCS*, pages 94–111. Springer, 1987.

[5] F. Bartels. *On generalized coinduction and probabilistic specification formats*. PhD thesis, Vrije Universiteit Amsterdam, 2004.

[6] F. Bonchi and D. Pous. Web appendix for this paper. http://perso.ens-lyon.fr/damien.pous/hknt, 2012.

[7] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. CAV*, vol. 3114 of *LNCS*. Springer, 2004.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[9] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical Theory of Automata*, vol. 12(6), pages 529–561. Polytechnic Press, NY, 1962.

[10] D. Caucal. Graphes canoniques de graphes algébriques. *ITA*, 24:339–352, 1990.

[11] S. Christensen, H. Hüttel, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. *Information and Computation*, 121(2):143–148, 1995.

[12] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1 (2/3):251–273, 1992.

[13] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. FOCS*, pages 453–462. IEEE Computer Society, 1995.

[14] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theoretical Computer Science*, 158(1&2):143–159, 1996.

[15] J. E. Hopcroft. An n log n algorithm for minimizing in a finite automaton. In *Proc. International Symposium of Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[16] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. TR 114, Cornell Univ., December 1971.

[17] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[18] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[19] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *Proc. STOC*, pages 264–274. ACM, 1992.

[20] O. Lengál, J. Simácek, and T. Vojnar. Vata: A library for efficient manipulation of non-deterministic tree automata. In *TACAS*, vol. 7214 of *LNCS*, pages 79–94. Springer, 2012.

[21] M. Lenisa. From set-theoretic coinduction to coalgebraic coinduction: some results, some problems. *ENTCS*, 19:2–22, 1999.

[22] D. Lucanu and G. Rosu. Circular coinduction with special contexts. In *Proc. ICFEM*, vol. 5885 of *LNCS*, pages 639–659. Springer, 2009.

[23] A. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *Proc. STOC*, pages 1–9. ACM, 1973.

[24] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

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

[26] D. Pous. Complete lattices and up-to techniques. In *Proc. APLAS*, vol. 4807 of *LNCS*, pages 351–366. Springer, 2007.

[27] J. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proc. CONCUR*, vol. 1466 of *LNCS*, pages 194–218. Springer, 1998.

[28] D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8:447–479, 1998.

[29] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[30] A. Silva, F. Bonchi, M. Bonsangue, and J. Rutten. Generalizing the powerset construction, coalgebraically. In *Proc. FSTTCS*, vol. 8 of *LIPIcs*, pages 272–283. Leibniz-Zentrum fuer Informatik, 2010.

[31] D. Tabakov and M. Vardi. Experimental evaluation of classical automata constructions. In *Proc. LPAR*, vol. 3835 of *LNCS*, pages 396–411. Springer, 2005.

[32] D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *LICS*, pages 280–291, 1997.

[33] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. CAV*, vol. 4144 of *LNCS*, pages 17–30. Springer, 2006.

A. Smallest bisimulation and compositionality

In this appendix, we show some (unrelated) properties that have been discussed through the paper, but never formally stated.

The first property concerns the relation computed by $\texttt{Naive}(x,y)$. The following proposition shows that it is the *smallest bisimulation* relating x and y.

Proposition 8. *Let x and y be two states of a DFA. Let $R_{\texttt{Naive}}$ be the relation built by $\texttt{Naive}(x,y)$. If $\texttt{Naive}(x,y) = true$, then $R_{\texttt{Naive}}$ is the smallest bisimulation relating x and y, i.e., $R_{\texttt{Naive}} \subseteq R$, for all bisimulations R such that $(x,y) \in R$.*

Proof. We have already shown in Proposition 2 that $R_{\texttt{Naive}}$ is a bisimulation. We need to prove that it is the smallest. Let R be a bisimulation such that $(x,y) \in R$. For all words $w \in A^*$ and pair of states (x',y') such that $x \overset{w}{\to} x'$ and $y \overset{w}{\to} y'$, it must hold that $(x',y') \in R$ (by definition of bisimulation).

By construction, for all $(x',y') \in R_{\texttt{Naive}}$ there exists a word $w \in A^*$, such that $x \overset{w}{\to} x'$ and $y \overset{w}{\to} y'$. Therefore all the pairs in $R_{\texttt{Naive}}$ must be also in R, that is $R_{\texttt{Naive}} \subseteq R$. □

The second property is

$$[\![X+Y]\!] = [\![X]\!] + [\![Y]\!] \ ,$$

which we have used in the Introduction to give an intuition of bisimulation up to context and to show that the problem of language inclusion can be reduced to language equivalence. We believe that this property is interesting, since it follows from the categorical observation made in [30] that determinised NFA are bialgebras [32], like CCS processes. For this reason, we prove here that $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ is a semi-lattice homomorphism.

Theorem 7. *Let (S,o,t) be a non-deterministic automaton and $(\mathcal{P}(S), o^\sharp, t^\sharp)$ be the corresponding deterministic automaton obtained through the powerset construction. The function $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ is a semi-lattice homomorphism, that is, for all $X_1, X_2 \in \mathcal{P}(S)$,*

$$[\![X_1 + X_2]\!] = [\![X_1]\!] + [\![X_2]\!] \quad \textit{and} \quad [\![0]\!] = 0\,.$$

Proof. We prove that for all words $w \in A^*$, $[\![X_1 + X_2]\!](w) = [\![X_1]\!](w) + [\![X_2]\!](w)$, by induction on w.

- for ϵ, we have:
$$\begin{aligned}[\![X_1+X_2]\!](\epsilon) &= o^\sharp(X_1+X_2)\\ &= o^\sharp(X_1) + o^\sharp(X_2) = [\![X_1]\!](\epsilon) + [\![X_2]\!](\epsilon)\,.\end{aligned}$$
- for $a \cdot w$, we have:
$$\begin{aligned}&[\![X_1+X_2]\!](a\cdot w)\\ &= [\![t^\sharp_a(X_1+X_2)]\!](w) && \text{(by definition)}\\ &= [\![t^\sharp_a(X_1) + t^\sharp_a(X_2)]\!](w) && \text{(by definition)}\\ &= [\![t^\sharp_a(X_1)]\!](w) + [\![t^\sharp_a(X_2)]\!](w) && \text{(by induction hypothesis)}\\ &= [\![X_1]\!](a\cdot w) + [\![X_2]\!](a\cdot w)\,. && \text{(by definition)}\end{aligned}$$

For the second part, we prove that for all words $w \in A^*$, $[\![0]\!](w) = 0$, again by induction on w. *Base case:* $[\![0]\!](\epsilon) = o^\sharp(0) = 0$. *Inductive case:* $[\![0]\!](a\cdot w) = [\![t^\sharp_a(0)]\!](w) = [\![0]\!](w)$ that by induction hypothesis is 0. □

B. Proofs of Section 2

Proposition 1. Two states are language equivalent iff there exists a bisimulation that relates them.

Proof. Let $R_{[\![-]\!]}$ be the relation $\{(x,y) \mid [\![x]\!] = [\![y]\!]\}$. We prove that $R_{[\![-]\!]}$ is a bisimulation. If $x \; R_{[\![-]\!]} \; y$, then $o(x) = [\![x]\!](\epsilon) = [\![y]\!](\epsilon) = o(y)$. Moreover, for all $a \in A$ and $w \in A^*$, $[\![t_a(x)]\!](w) = [\![x]\!](a\cdot w) = [\![y]\!](a\cdot w) = [\![t_a(y)]\!](w)$ that means $[\![t_a(x)]\!] = [\![t_a(y)]\!]$, that is $t_a(x) \; R_{[\![-]\!]} \; t_a(y)$.

We now prove the other direction. Let R be a bisimulation. We want to prove that $x \; R \; y$ entails $[\![x]\!] = [\![y]\!]$, i.e., for all $w \in A^*$, $[\![x]\!](w) = [\![y]\!](w)$. We proceed by induction on w. For $w = \epsilon$, we have $[\![x]\!](\epsilon) = o(x) = o(y) = [\![y]\!](\epsilon)$. For $w = a \cdot w'$, since R is a bisimulation, we have $t_a(x) \; R \; t_a(y)$ and thus $[\![t_a(x)]\!](w') = [\![t_a(y)]\!](w')$ by induction. This allows us to conclude since $[\![x]\!](a\cdot w') = [\![t_a(x)]\!](w')$ and $[\![y]\!](a\cdot w') = [\![t_a(y)]\!](w')$. □

Lemma 1. The following functions are compatible:

id: the identity function;

$f \circ g$: the composition of compatible functions f and g;

$\bigcup F$: the pointwise union of an arbitrary family F of compatible functions: $\bigcup F(R) = \bigcup_{f\in F} f(R)$;

f^ω: the (omega) iteration of a compatible function f.

Proof. The first two points are straightforward;

For the third one, assume that F is a family of compatible functions. Suppose that $R \rightarrowtail R'$; for all $f \in F$, we have $f(R) \rightarrowtail f(R')$ so that $\bigcup_{f\in F} f(R) \rightarrowtail \bigcup_{f\in F} f(R')$.

For the last one, assume that f is compatible; for all n, f^n is compatible because (a) $f^0 = id$ is compatible (by the first point) and (b) $f^{n+1} = f \circ f^n$ is compatible (by the second point and induction hypothesis). By definition $f^\omega = \bigcup_n f^n$ and thus, by the third point, f^ω is compatible. □

Lemma 2. The following functions are compatible:

- the constant reflexive function: $r(R) = \{(x,x) \mid \forall x \in S\}$;
- the converse function: $s(R) = \{(y,x) \mid x \; R \; y\}$;
- the squaring function: $t(R) = \{(x,z) \mid \exists y, x \; R \; y \; R \; z\}$.

Proof. r: observe that the identity relation $Id = \{(x,x) \mid \forall x \in S\}$ is always a bisimulation, i.e., $Id \rightarrowtail Id$. Thus for all R, R' $r(R) = Id \rightarrowtail Id = r(R')$.

s: observe that the definition of progression is completely symmetric. Therefore, if $R \rightarrowtail R'$, then $s(R) \rightarrowtail s(R')$.

t: assume that $R \rightarrowtail R'$. For each $(x,z) \in t(R)$, there exists y such that $(x,y) \in R$ and $(y,z) \in R$. By assumption, (1) $o'(x) = o'(y) = o'(z)$ and (2) for all $a \in A$, $t'_a(x) \; R' \; t'_a(y) \; R' \; t'_a(z)$, that is $t'_a(x) \; t(R') \; t'_a(z)$.

□

C. Proofs of Section 3

Lemma 4. For all relations R, the relation $\rightsquigarrow_R$ is convergent.

Proof. We have that $Z \rightsquigarrow_R Z'$ implies $|Z'| > |Z|$, where $|X|$ denotes the cardinality of the set X (note that $\rightsquigarrow_R$ is irreflexive). Since $|Z'|$ is bounded by $|S|$, the number of states of the NFA, the relation $\rightsquigarrow_R$ is strongly normalising. We can also check that whenever $Z \rightsquigarrow_R Z_1$ and $Z \rightsquigarrow_R Z_2$, either $Z_1 = Z_2$ or there is some Z' such that $Z_1 \rightsquigarrow_R Z'$ and $Z_2 \rightsquigarrow_R Z'$. Therefore, $\rightsquigarrow_R$ is convergent. □

Lemma 13. *The relation $\rightsquigarrow_R$ is contained in $c(R)$.*

Proof. If $Z \rightsquigarrow_R Z'$ then there exists $(X, Y) \in (s \cup \mathrm{id})(R)$ such that $Z = Z + X$ and $Z' = Z + Y$. Therefore $Z\ c(R)\ Z'$ and, thus, $\rightsquigarrow_R$ is contained in $c(R)$. □

Lemma 14. *Let $X, Y \in \mathcal{P}(S)$, we have $(X + Y){\downarrow_R} = (X{\downarrow_R} + Y{\downarrow_R}){\downarrow_R}$.*

Proof. Follows from confluence (Lemma 4) and from the fact that for all Z, Z', U, $Z \rightsquigarrow_R Z'$ entails $U + Z \rightsquigarrow_R^{=} U + Z'$. □

Theorem 3. For all relations R, and for all $X, Y \in \mathcal{P}(S)$, we have $X{\downarrow_R} = Y{\downarrow_R}$ iff $(X, Y) \in c(R)$.

Proof. From right to left. We proceed by induction on the derivation of $(X, Y) \in c(R)$. The cases for rules r, s, and t are straightforward. For rule id, suppose that $X\ R\ Y$, we have to show $X{\downarrow_R} = Y{\downarrow_R}$:

- if $X = Y$, we are done;
- if $X \subsetneq Y$, then $X \rightsquigarrow_R X + Y = Y$;
- if $Y \subsetneq X$, then $Y \rightsquigarrow_R X + Y = X$;
- if neither $Y \subseteq X$ nor $X \subseteq Y$, then $X, Y \rightsquigarrow_R X + Y$.

(In the last three cases, we conclude by confluence—Lemma 4.) For rule u, suppose by induction that $X_i{\downarrow_R} = Y_i{\downarrow_R}$ for $i \in 1, 2$; we have to show that $(X_1 + Y_1){\downarrow_R} = (X_2 + Y_2){\downarrow_R}$. This follows by Lemma 14.

From left to right. By Lemma 13, we have $X\ c(R)\ X{\downarrow_R}$ for any set X, so that $X\ c(R)\ X{\downarrow_R} = Y{\downarrow_R}\ c(R)\ Y$. □

Lemma 5. The three algorithms require at most $1+v{\cdot}|R|$ iterations, where $|R|$ is the size of the produced relation; moreover, this bound is reached whenever they return true.

Proof. At each iteration, one pair is extracted from *todo*. The latter contains one pair before entering the loop and v pairs are added to it every time that a pair is added to R. □

Lemma 15. *Let x and y be two states of a DFA. Let $R_{\mathtt{Naive}}$ and $R_{\mathtt{HK}}$ be relations computed by $\mathtt{Naive}(x, y)$ and $\mathtt{HK}(x, y)$, respectively. If $\mathtt{Naive}(x, y) = \mathtt{HK}(x, y) = true$, then $e(R_{\mathtt{Naive}}) = e(R_{\mathtt{HK}})$.*

Proof. By the proof of Proposition 3, $e^\omega(R_{\mathtt{HK}})$ is a bisimulation. Since e is idempotent, we have $e^\omega = e$ and thus $e(R_{\mathtt{HK}})$ is a bisimulation; we can thus deduce by Proposition 8 that $R_{\mathtt{Naive}} \subseteq e(R_{\mathtt{HK}})$. Moreover, by definition of the algorithms, we have $R_{\mathtt{HK}} \subseteq R_{\mathtt{Naive}}$. Summarising,

$$R_{\mathtt{HK}} \subseteq R_{\mathtt{Naive}} \subseteq e(R_{\mathtt{HK}})$$

It follows that $e(R_{\mathtt{HK}}) = e(R_{\mathtt{Naive}})$, e being monotonic and idempotent. □

Lemma 6. Let $R_{\mathtt{Naive}}$, $R_{\mathtt{HK}}$, and $R_{\mathtt{HKC}}$ denote the relations produced by the three algorithms. We have

$$|R_{\mathtt{HKC}}|, |R_{\mathtt{HK}}| \leq m \qquad\qquad |R_{\mathtt{Naive}}| \leq m^2 \ , \tag{2}$$

where $m \leq 2^n$ is the number of accessible states in the determinised NFA and n is the number of states of the NFA. If the algorithms returned true, we moreover have

$$|R_{\mathtt{HKC}}| \leq |R_{\mathtt{HK}}| \leq |R_{\mathtt{Naive}}| \ . \tag{3}$$

Proof. For the first point, let PS denote the set of (determinised NFA) states accessible from the two starting states, so that $m = |PS| \leq 2^n$. Since $R_{\mathtt{Naive}} \subseteq PS \times PS$, we deduce $|R_{\mathtt{Naive}}| \leq m^2$. Since each pair added to $R_{\mathtt{HK}}$ merges two distinct equivalence classes in $e(R_{\mathtt{HK}})$, we necessarily have $|R_{\mathtt{HK}}| \leq m$ (the largest partition of PS has exactly m singletons). Similarly, each pair added to $R_{\mathtt{HKC}}$ merges *at least* two distinct equivalence classes in $c(R_{\mathtt{HK}})$, so that we also have $|R_{\mathtt{HKC}}| \leq m$.

For the second point, $|R_{\mathtt{HK}}| \leq |R_{\mathtt{Naive}}|$ follows from the fact that $R_{\mathtt{HK}} \subseteq R_{\mathtt{Naive}}$, by definition of the algorithms. The other inequality is less obvious.

By construction, $R_{\mathtt{HKC}} \subseteq R_{\mathtt{Naive}}$ and, since e is monotonic, $e(R_{\mathtt{HKC}}) \subseteq e(R_{\mathtt{Naive}}) = e(R_{\mathtt{HK}})$ (the latter equality is given by Proposition 15). In particular, there are more equivalence classes in $e(R_{\mathtt{HKC}})$ than in $e(R_{\mathtt{HK}})$; using the same argument as above, we deduce that $|R_{\mathtt{HKC}}| \leq |R_{\mathtt{HK}}|$. □

Lemma 8. Let X, Y be two sets of states; let $R_\subseteq$ and $R_\supseteq$ be the relations computed by $\mathtt{HKC}(X{+}Y, Y)$ and $\mathtt{HKC}(X{+}Y, X)$, respectively. If $R_\subseteq$ and $R_\supseteq$ are bisimulations up to congruence, then the following relation is a bisimulation up to congruence:

$$R_= = \{(X', Y') \mid (X'{+}Y', Y') \in R_\subseteq \text{ or } (X'{+}Y', X') \in R_\supseteq\}.$$

Proof. Let $(X', Y') \in R_=$ and suppose that $(X'{+}Y', Y') \in R_\subseteq$ (the other case is symmetric).

First notice that all pairs in $R_\supseteq$ necessarily have the shape $(t^\sharp_w(X{+}Y),\ t^\sharp_w(X))$, for some word w. Since $R_\supseteq$ is a bisimulation up to congruence, $c(R_\supseteq)$ is a bisimulation. Since $(X{+}Y, X) \in c(R_\supseteq)$ then, for all words w, $(t^\sharp_w(X{+}Y), t^\sharp_w(X)) \in c(R_\supseteq)$ and thus $(X'{+}Y', X') \in c(R_\supseteq)$ (we have $X' = t^\sharp_w(X)$ and $Y' = t^\sharp_w(Y)$ for some word w).

Since $c(R_\subseteq)$ and $c(R_\supseteq)$ are bisimulations containing $(X'{+}Y', Y')$ and $(X'{+}Y', X')$, it holds that:

1. $o^\sharp(X') = o^\sharp(X' + Y') = o^\sharp(Y')$;
2. for all a, $t^\sharp_a(X' + Y')\ c(R_\supseteq)\ t^\sharp_a(X')$ and $t^\sharp_a(X' + Y')\ c(R_\subseteq)\ t^\sharp_a(Y')$.

By Lemma 7, $t^\sharp_a(Y') \subseteq t^\sharp_a(X'){\downarrow_{R_\supseteq}}$ and $X' \subseteq t^\sharp_a(Y'){\downarrow_{R_\subseteq}}$ and since all the rewriting rules for $R_\subseteq$ and $R_\supseteq$ are also rewriting rules for $R_=$, then $t^\sharp_a(Y') \subseteq t^\sharp_a(X'){\downarrow_{R_=}}$ and $t^\sharp_a(X') \subseteq t^\sharp_a(Y'){\downarrow_{R_=}}$. By the first observation in the proof of Lemma 7, this means that $t^\sharp_a(X')\ c(R_=)\ t^\sharp_a(Y')$. □

D. Proofs of Section 4

Proposition 4. For all sets X, Y, we have $[\![X]\!] \subseteq [\![Y]\!]$ iff there exists a simulation T such that for all $x \in X$, $x\ T\ Y$.

Proof. Let $T_{[\![-]\!]}$ be the relation $\{(x,Y) \mid [\![x]\!] \subseteq [\![Y]\!]\}$. We prove that $T_{[\![-]\!]}$ is a simulation. If $x \; T_{[\![-]\!]} \; Y$, then $o(x) = [\![x]\!](\epsilon) \leq [\![Y]\!](\epsilon) = o^\sharp(Y)$. Moreover, for all $a \in A$ $x' \in t_a(x)$ and $w \in A^*$, $[\![x']\!](w) \subseteq [\![x]\!](a \cdot w) \subseteq [\![Y]\!](a \cdot w) = [\![t_a^\sharp(Y)]\!](w)$ that means $[\![x']\!] \subseteq [\![t_a^\sharp(Y)]\!]$, that is $t_a(x) \; T_{[\![-]\!]} \; t_a^\sharp(Y)$.

We now prove the other direction. Let T be a simulation. We want to prove that $x \; T \; Y$ entails $[\![x]\!] \subseteq [\![Y]\!]$, i.e., for all $w \in A^*$, $[\![x]\!](w) \leq [\![Y]\!](w)$. We proceed by induction on w. For $w = \epsilon$, we have $[\![x]\!](\epsilon) = o(x) \leq o^\sharp(Y) = [\![Y]\!](\epsilon)$. For $w = a \cdot w'$, since T is a simulation, we have $t_a(x) \; T \; t_a^\sharp(Y)$ and thus $[\![t_a(x)]\!](w') \leq [\![t_a^\sharp(Y)]\!](w')$ by induction. This allows us to conclude since $[\![x]\!](a \cdot w') = [\![t_a(x)]\!](w')$ and $[\![y]\!](a \cdot w') = [\![t_a^\sharp(y)]\!](w')$. □

Definition 10. *A function* $f : \mathcal{P}(S \times \mathcal{P}(S)) \to \mathcal{P}(S \times \mathcal{P}(S))$ *is* s-compatible *if is monotone and for all relations* $T, T' \subseteq S \times \mathcal{P}(S)$, $T \rightarrowtail_s T'$ *entails* $f(T) \rightarrowtail_s f(T')$.

Lemma 16. *Any simulation* T *up to an s-compatible function* f ($T \rightarrowtail_s f(T)$) *is contained in a simulation, namely* $f^\omega(T)$.

Proof. Same proof as for Proposition 3. □

Lemma 17. *The upward closure function* $\uparrow$ *is s-compatible.*

Proof. We assume that $T \rightarrowtail_s T'$ and we prove that $\uparrow T \rightarrowtail_s \uparrow T'$. If $x \uparrow T \; Y$, then $\exists Y' \subseteq Y$ such that $x \; T \; Y'$. Since $Y' \subseteq Y$, $o^\sharp(Y') \leq o^\sharp(Y)$ and $t_a^\sharp(Y') \subseteq t_a^\sharp(Y)$ for all $a \in A$. Since $T \rightarrowtail_s T'$ and $x \; T \; Y'$, then $o(x) \leq o^\sharp(Y') \leq o^\sharp(Y)$ and $t_a(x) \uparrow T' \; t_a^\sharp(Y)$ for all $a \in A$. □

Theorem 4. Any simulation up to $\uparrow$ is contained in a simulation.

Proof. By Lemmas 16 and 17. □

Lemma 18. *The relation*

$$\begin{aligned} R' = &\{(x+y,\ z)\} \\ &+ \{(x+Y_i+y_{i+1},\ Z_{i+1}) \mid i < n\} \\ &+ \{(x+Y_i+x_{i+1},\ Z_{i+1}) \mid i < n\} \end{aligned}$$

is a bisimulation up to congruence for the NFA in Fig. 5.

Proof. First notice that

$$X_1 + y \quad c(R') \quad x + Y_1 \quad c(R') \quad Z_1$$

We then consider each kind of pair of R' separately:

- (x, y): we have $o^\sharp(x+y) = 0 = o^\sharp(z)$ and $t_a^\sharp(x+y) = X_1 + y \; R' \; Z_1 = t_a^\sharp(z)$ and, similarly, $t_b^\sharp(x+y) = x + Y_1 \; R' \; Z_1 = t_b^\sharp(z)$.
- $(x+Y_i+y_{i+1},\ Z_{i+1})$: both members are accepting iff $i+1 = n$; setting $j = \min(i+2, n)$, we have

$$\begin{aligned} t_a^\sharp(x+Y_i+y_{i+1}) &= X_1 + y + y_2 + \cdots + y_j \\ &\quad c(R') \; x + Y_1 + y_2 + \cdots + y_j \\ &= x + Y_j \; R' \; Z_j = t_a^\sharp(Z_{i+1}) \end{aligned}$$

and

$$t_b^\sharp(x+Y_i+y_{i+1}) = x + Y_j \; R' \; Z_j = t_b^\sharp(Z_{i+1})$$

- $(x+Y_i+x_{i+1},\ Z_{i+1})$: both members are accepting iff $i+1 = n$; if $i+1 < n$ then we have:

$$\begin{aligned} t_a^\sharp(x+Y_i+x_{i+1}) &= X_1 + y + y_2 + \cdots + y_{i+1} + x_{i+2} \\ &\quad c(R') \; x + Y_1 + y_2 + \cdots + y_{i+1} + x_{i+2} \\ &= x + Y_{i+1} + x_{i+2} \\ &\quad R' \; Z_{i+2} = t_a^\sharp(Z_{i+1}) \end{aligned}$$

and

$$t_b^\sharp(x+Y_i+x_{i+1}) = x + Y_{i+1} + x_{i+2} \; R' \; Z_{i+2} = t_b^\sharp(Z_{i+1})$$

otherwise, i.e., $i+1 = n$, notice that:

$$\begin{aligned} x + Y_n + x_n \; &c(R') \; Z_n + y_n \\ &c(R') \; x + Y_n + y_n = x + Y_n \\ &c(R') \; Z_n = t_a^\sharp(Z_n)\ , \end{aligned}$$

from which we deduce:

$$\begin{aligned} t_a^\sharp(x+Y_i+x_n) &= X_1 + y + y_2 + \cdots + y_n + x_n \\ &\quad c(R') \; x + Y_1 + y_2 + \cdots + y_n + x_n \\ &= x + Y_n + x_n \; c(R') \; t_a^\sharp(Z_n) \end{aligned}$$

and

$$t_b^\sharp(x+Y_i+x_n) = x + Y_n + x_n \; c(R') \; t_a^\sharp(Z_n)$$

□

E. Proofs of Section 5

Theorem 5. Any simulation up to $\curlywedge$ is contained in a simulation.

Proof. By Lemma 16, it suffices to show that $\curlywedge$ is s-compatible. Suppose that $T \rightarrowtail_s T'$, we have to show that $\curlywedge T \rightarrowtail_s \curlywedge T'$. Assume that $x \curlywedge T \; Y$.

- if $x \preceq^{\forall\exists} Y$ then $x \preceq y$ for some $y \in Y$. Therefore, we have $o(x) \leq o(y) \leq o^\sharp(Y)$ and for all $a \in A$, $x' \in t_a(x)$, we have some $y' \in t_a(y)$ with $x' \preceq y'$. Since $t_a(y) \subseteq t_a^\sharp(Y)$, we deduce $x' \preceq^{\forall\exists} t_a^\sharp(Y)$, and hence $x' \curlywedge T' \; t_a^\sharp(Y)$, as required.
- otherwise, we have some $(x', Y') \in T$ such that $(x', Y') \sqsubseteq^{\preceq} (x, Y)$, i.e., $x \preceq x'$ and $Y' \preceq^{\forall\exists} Y$. Since $T \rightarrowtail_s T'$, we have $o(x) \leq o(x') \leq o^\sharp(Y') \leq o^\sharp(Y)$. Now take some $x'' \in t_a(x)$, we have some $x''' \in t_a(x')$ with $x'' \preceq x'''$, and since $T \rightarrowtail_s T'$, we know $x''' \; T' \; t_a^\sharp(Y')$. It suffices to show that $t_a^\sharp(Y') \preceq^{\forall\exists} t_a^\sharp(Y)$ to conclude; this follows easily from $Y' \preceq^{\forall\exists} Y$ and from the definition of similarity. □

Lemma 11. For all relations R, $c'(R) = c(R \cup \overline{\preceq})$.

Proof. The inclusion $c(R \cup \overline{\preceq}) \subseteq c'(R)$ is trivial. For the other inclusion we take $d = r' \cup s \cup t \cup u \cup id$ and we prove by induction that for all natural numbers n, $d^n(R) \subseteq c(R \cup \overline{\preceq})$. For $n = 0$, $d^0(R) = R \subseteq c(R \cup \overline{\preceq})$. For $n+1$, $d^{n+1}(R) = d(d^n(R))$. By induction hypothesis, $d^n(R) \subseteq c(R \cup \overline{\preceq})$ and, by monotonicity of d, $d(d^n(R)) \subseteq d(c(R \cup \overline{\preceq}))$. By definition of d, the latter is equal to $c(R \cup \overline{\preceq})$. □

Lemma 12. For all relations $T \subseteq S \times \mathcal{P}(S)$, $\widehat{\curlywedge T} \subseteq c'(\widehat{T})$.

Proof. If $(x+Y,Y) \in \widehat{\Yup T}$, then either (a) $x \preceq^{\forall\exists} Y$ or (b) there exists $x \preceq x'$ and $Y' \preceq^{\forall\exists} Y$ such that $(x',Y') \in T$. We have to show $(x{+}Y,Y) \in c'(\widehat{T})$, i.e., $(x{+}Y,Y) \in c(\widehat{T}+\preceq)$ by Lemma 11, that is $x \in Y{\downarrow}_{\widehat{T}+\preceq}$ by Lemma 7. For (b), we have:

$$
\begin{aligned}
Y &\rightsquigarrow^{\star}_{\widehat{T}+\preceq} Y+Y' && (Y' \preceq^{\forall\exists} Y)\\
&\rightsquigarrow_{\widehat{T}+\preceq} Y+Y'+x' && ((x'{+}Y',Y') \in \widehat{T})\\
&\rightsquigarrow_{\widehat{T}+\preceq} Y+Y'+x'+x && (x \preceq x')
\end{aligned}
$$

$x \in Y{\downarrow}_{\widehat{T}+\preceq}$ follows by confluence (Lemma 4). For (a), we immediately have that $Y \rightsquigarrow_{\widehat{T}+\preceq} Y+x$. □

Proposition 7. If T is a simulation up to $\Yup$, then $\widehat{T}$ is a bisimulation up to c'.

Proof. First observe that if $T \rightarrowtail_s T'$, then $\widehat{T} \rightarrowtail u^{\omega}(\widehat{T'})$. Therefore, if $T \rightarrowtail_s {\uparrow}T$, then $\widehat{T} \rightarrowtail u^{\omega}(\widehat{{\uparrow}T})$. By Lemma 12, $\widehat{T} \rightarrowtail u^{\omega}(c'(\widehat{T})) = c'(\widehat{T})$. □

Lemma 19. *The relation*

$$
\begin{aligned}
R'' = &\{(z{+}x{+}y,\ x{+}y)\}\\
&\cup \{(Z_{i+1}{+}X_i{+}y{+}y_{i+1},\ X_i{+}y{+}y_{i+1}) \mid i<n\}\\
&\cup \{(Z_{i+1}{+}X_{i+1}{+}y,\ X_{i+1}{+}y) \mid i<n\}\ ,
\end{aligned}
$$

is a bisimulation up to c' for the NFA in Figure 6.

Proof. Let X_i' be the set X_i without x_1 and note that $X_i \xrightarrow{a} X_{i+1}$ and $X_i \xrightarrow{b} X'_{i+1}$. First we observe that for all i,

$$X_i' + Y_1 \rightsquigarrow_{R''\cup\preceq} X_i' + Y_1 + Z_1 \rightsquigarrow_{R''\cup\preceq} X_i' + Y_1 + Z_1 + x_1$$

where the first reduction is given by $(Z_1+X_0+y+y_1, X_0+y+y_1) \in R''$ and the second by $x_1 \preceq z_1$. Since $X_i'+x_1 = X_i$, then one can apply the third kind of pairs in R'', so that

$$X_i' + Y_1 \rightsquigarrow^{*}_{R''\cup\preceq} X_i + Y_1 + Z_i$$

that is $Z_i \subseteq (X_i'+Y_1){\downarrow}_{R''\cup\preceq}$. By Lemmas 7 and 11, this means that

$$Z_i + X_i' + Y_1\ c'(R'')\ X_i' + Y_1 \tag{2}$$

If we moreover have y_{i+1}, we can apply the second kind of pair in R'' and obtain

$$X_i' + Y_1 + y_{i+1} \rightsquigarrow^{*}_{R''\cup\preceq} X_i + Y_1 + Z_{i+1} + y_{i+1}$$

that is

$$Z_{i+1} + X_i' + Y_1 + y_{i+1}\ c'(R'')\ X_i' + Y_1 + y_{i+1} \tag{3}$$

With (2) and (3), it is easy to prove that R'' is a bisimulation up to c', by simply proceeding by cases:

- $(z{+}x{+}y,\ x{+}y)$: we have $o^\sharp(x{+}y{+}z) = 0 = o^\sharp(x{+}y)$ and $t_a^\sharp(x{+}y{+}z) = Z_1{+}X_1{+}y\ R''\ X_1+y = t_a^\sharp(x{+}y)$ and, similarly, $t_b^\sharp(x{+}y{+}z) = Z_1{+}x{+}Y_1\ R''\ x{+}Y_1 = t_b^\sharp(z)$.
- $(Z_{i+1}{+}X_i{+}y{+}y_{i+1},\ X_i{+}y{+}y_{i+1})$ and $i < n-1$: both members are not accepting;

$$
\begin{aligned}
t_a^\sharp(Z_{i+1}{+}X_i{+}y{+}y_{i+1}) &= Z_{i+2}{+}X_{i+1}{+}y{+}y_{i+2}\\
&R''\ X_{i+1}{+}y{+}y_{i+2}\\
&= t_a^\sharp(X_i{+}y{+}y_{i+1})
\end{aligned}
$$

and

$$
\begin{aligned}
t_b^\sharp(Z_{i+1}{+}X_i{+}y{+}y_{i+1}) &= Z_{i+2}{+}X'_{i+1}{+}Y_1{+}y_{i+2}\\
&c'(R'')\ X'_{i+1}{+}Y_1{+}y_{i+2}\\
&= t_b^\sharp(X_i{+}y{+}y_{i+1})
\end{aligned}
$$

- $(Z_n{+}X_{n-1}{+}y{+}y_n,\ X_{n-1}{+}y{+}y_n)$ and $i = n-1$: both members are accepting;

$$
\begin{aligned}
t_a^\sharp(Z_n{+}X_{n-1}{+}y{+}y_n) &= Z_n{+}X_n{+}y\\
&R''\ X_n{+}y\\
&= t_a^\sharp(X_{n-1}{+}y{+}y_n)
\end{aligned}
$$

and

$$
\begin{aligned}
t_b^\sharp(Z_n{+}X_{n-1}{+}y{+}y_n) &= Z_n{+}X'_n{+}Y_1\\
&c'(R'')\ X'_n{+}Y_1\\
&= t_b^\sharp(X_{n-1}{+}y{+}y_n)
\end{aligned}
$$

- $(Z_{i+1}{+}X_{i+1}{+}y,\ X_{i+1}{+}y)$ and $i < n-1$: both members are not accepting;

$$
\begin{aligned}
t_a^\sharp(Z_{i+1}{+}X_{i+1}{+}y) &= Z_{i+2}{+}X_{i+2}{+}y\\
&R''\ X_{i+2}{+}y\\
&= t_a^\sharp(X_{i+1}{+}y)
\end{aligned}
$$

and

$$
\begin{aligned}
t_b^\sharp(Z_{i+1}{+}X_{i+1}{+}y) &= Z_{i+2}{+}X'_{i+2}{+}Y_1\\
&c(R'')\ X'_{i+2}{+}Y_1\\
&= t_b^\sharp(X_{i+1}{+}y)
\end{aligned}
$$

- $(Z_n{+}X_n{+}y,\ X_n{+}y)$: both members are accepting; Moreover,

$$
\begin{aligned}
t_a^\sharp(Z_n{+}X_n{+}y) &= Z_n{+}X_n{+}y\\
&R''\ X_n{+}y = t_a^\sharp(X_n{+}y)
\end{aligned}
$$

and

$$
\begin{aligned}
t_b^\sharp(Z_n{+}X_n{+}y) &= Z_n{+}X'_n{+}Y_1\\
&c(R'')\ X'_n{+}Y_1\\
&= t_b^\sharp(X_n{+}y)
\end{aligned}
$$

The cases for the letter c are always trivial since $Z_i \xrightarrow{c} 0$. □

Concurrent Flexible Reversibility*

Ivan Lanese[1], Michael Lienhardt[2], Claudio Antares Mezzina[3], Alan Schmitt[4], and Jean-Bernard Stefani[4]

[1] Focus Team, University of Bologna/Inria, Italy lanese@cs.unibo.it
[2] PPS Laboratory, Paris Diderot University, France lienhard@cs.unibo.it
[3] SOA Unit, FBK, Trento, Italy mezzina@fbk.eu
[4] Inria, France alan.schmitt@inria.fr, jean-bernard.stefani@inria.fr

Abstract. Concurrent reversibility has been studied in different areas, such as biological or dependable distributed systems. However, only "rigid" reversibility has been considered, allowing to go back to a past state and restart the exact same computation, possibly leading to divergence. In this paper, we present croll-π, a concurrent calculus featuring *flexible reversibility*, allowing the specification of alternatives to a computation to be used upon rollback. Alternatives in croll-π are attached to messages. We show the robustness of this mechanism by encoding more complex idioms for specifying flexible reversibility, and we illustrate the benefits of our approach by encoding a calculus of communicating transactions.

1 Introduction

Reversible programs can be executed both in the standard, forward direction as well as in the backward direction, to go back to past states. Reversible programming is attracting much interest for its potential in several areas. For instance, chemical and biological reactions are typically bidirectional, and the direction of execution is fixed by environmental conditions such as temperature. Similarly, quantum computations are reversible as long as they are not observed. Reversibility is also used for backtracking in the exploration of a program state-space toward a solution, either as part of the design of the programming language as in Prolog, or to implement transactions. We are particularly interested in the use of reversibility for modeling and programming concurrent reliable systems. In this setting, the main idea is that in case of an error the program backtracks to a past state where the decisions leading to the error have not been taken yet, so that a new forward execution may avoid repeating the (same) error.

Reversibility has a non trivial interplay with concurrency. Understanding this interplay is fundamental in many of the areas above, e.g., for biological or reliable distributed systems, which are naturally concurrent. In the spirit of concurrency, independent threads of execution should be rolled-back independently, but causal dependencies between related threads should be taken into account.

* This work has been partially supported by the French National Research Agency (ANR), projects REVER ANR 11 INSE 007 and PiCoq ANR 10 BLAN 0305.

This form of reversibility, termed *causal consistent*, was first introduced by RCCS [12], a reversible variant of CCS. RCCS paved the way to the definition of reversible variants of more expressive concurrent calculi [9, 20, 22, 24]. This line of research considered rigid, uncontrolled, step-by-step reversibility. *Step-by-step* means that each single step can be undone, as opposed, e.g., to checkpointing where many steps are undone at once. *Uncontrolled* means that there is no hint as to when to go forward and when to go backward, and up to where. *Rigid* means that the execution of a forward step followed by the corresponding backward step leads back to the starting state, where an identical computation can restart.

While these works have been useful to understand the basics of concurrent reversibility in different settings, some means to *control* reversibility are required in practice. In the literature four different forms of control have been proposed: relating the direction of execution to some energy parameter [3], introducing irreversible actions [13], using an explicit rollback primitive [19], and using a superposition operator to control forward and backward execution [26].

With the exception of [26], these works have causally consistent reversibility but exhibit rigid reversibility. However, rigid reversibility may not always be the best choice. In the setting of reliable systems, for instance, rigid reversibility means that to recover from an error a past state is reached. From this past state the computation that lead to the error is still possible. If the error was due to a transient fault, retrying the same computation may be enough to succeed. If the failure was permanent, the program may redo the same error again and again.

Our goal is to overcome this limitation by providing the programmer with suitable linguistic constructs to specify what to do after a causally consistent backward computation. Such constructs can be used to ensure that new forward computations explore new possibilities. To this end, we build on our previous work on roll-π [19], a calculus where concurrent reversibility is controlled by the roll γ operator. Executing it reverses the action referred by γ together with all the dependent actions. Here, we propose a new calculus called croll-π, for compensating roll-π, as a framework for *flexible reversibility*. We attempt to keep croll-π as close as possible to roll-π while enabling many new possible applications. We thus simply replace roll-π communication messages $a\langle P\rangle$ by *messages with alternative* $a\langle P\rangle \div c\langle Q\rangle$. In forward computation, a message $a\langle P\rangle \div c\langle Q\rangle$ behaves exactly as $a\langle P\rangle$. However, if the interaction consuming it is reversed, the original message is not recreated—as would be the case with rigid reversibility—but the alternative $c\langle Q\rangle$ is released instead. Our rollback and alternative message primitives provide a simple form of reversibility control, which always respects the causal consistency of reverse computation. It contrasts with the fine-grained control provided by the superposition constructs in [26], where the execution of a CCS process can be constrained by a controller, possibly reversing identified past actions in a way that is non-causally consistent.

Our contributions are as follows. We show that the simple addition of alternatives to roll-π greatly extends its expressive power. We show that messages with alternative allow for programming different patterns for flexible reversibility. We show that croll-π can be used to model the communicating transactions of [14]. Notably, the tracking of causality of croll-π is more precise than the one in [14], thus allowing to improve on the original proposal by avoiding some spurious undo of actions. Additionally, we study some aspects of the behavioral theory of croll-π, including a context lemma for barbed congruence. This allows us to reason about croll-π programs, in particular to prove the correctness of the encodings of primitives for flexible reversibility and of the transactional calculus of [14]. Finally, we present an interpreter, written in Maude [11], for a small language based on croll-π.

Outline. Section 2 gives an informal introduction to croll-π. Section 3 defines the croll-π calculus, its reduction semantics, and it introduces the basics of its behavioral theory. Section 4 presents various croll-π idioms for flexible reversibility. Section 5 outlines the croll-π interpreter in Maude and the concurrent solution for the Eight Queens problem. Section 6 presents an encoding and an analysis of the TransCCS constructs from [14]. Section 7 concludes the paper with related work and a mention of future studies. The paper includes short proof sketches for the main results. We refer to the online technical report [18] for full proofs and an additional example, an encoding of the transactional constructs from [2].

2 Informal Presentation

Rigid reversibility in roll-π. The croll-π calculus is a conservative extension of the roll-π calculus introduced in [19].[5] We briefly review the roll-π constructs before presenting the extension added by croll-π. Processes in roll-π are essentially processes of the asynchronous higher-order π-calculus [27], extended with a rollback primitive. Processes in roll-π cannot directly execute, only *configurations* can. A configuration is essentially a parallel composition of *tagged processes* along with *memories* tracking past interactions and *connectors* tracing causality information. In a tagged process of the form $k : P$, the tag k uniquely identifies the process P in a given configuration. We often use the term *key* instead of tag.

The uniqueness of tags in configurations is achieved thanks to the following reduction rule that defines how parallel processes are split.

$$k : P \mid Q \longrightarrow \nu k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

In the above reduction, $\mid$ is the parallel composition operator and ν is the restriction operator, both standard from the π-calculus. As usual, the scope of restriction extends as far to the right as possible. Connector $k \prec (k_1, k_2)$ is used to remember that the process tagged by k has been split into two sub-processes identified by the new keys k_1 and k_2. Thus complex processes can be split into *threads*, where a thread is either a *message*, of the form $a\langle P\rangle$ (where a is a channel name), a receiver process (also called a *trigger*), of the form $a(X) \triangleright_\gamma P$, or a *rollback* instruction of the form roll k, where k is a key.

[5] The version of roll-π presented here is slightly refined w.r.t. the one in [19].

A *forward* communication step occurs when a message on a channel can be received by a trigger on the same channel. It takes the following form (roll-π is an asynchronous higher-order calculus).

$$(k_1 : a\langle P\rangle) \mid (k_2 : a(X) \triangleright_\gamma Q) \longrightarrow \nu k.\, k : Q\{^{P,k}/_{X,\gamma}\} \mid [\mu; k]$$

In this forward step, keys k_1 and k_2 identify threads consisting respectively of a message $a\langle P\rangle$ on channel a and a trigger $a(X) \triangleright_\gamma Q$ expecting a message on channel a. The result of the message input yields, as in higher-order π, the body of the trigger Q with the formal parameter X instantiated by the received value, i.e., process P. Message input also has three side effects: (i) the tagging of the newly created process $Q\{^{P,k}/_{X,\gamma}\}$ by a fresh key k; (ii) the creation of a memory $[\mu; k]$, which records the original two threads,[6] $\mu = (k_1 : a\langle P\rangle) \mid (k_2 : a(X) \triangleright_\gamma Q)$, together with key k; and (iii) the instantiation of variable γ with the newly created key k (the trigger construct is a binder both for its process parameter and its key parameter).

In roll-π, a forward computation, i.e., a series of forward reduction steps as above, can be perfectly undone by backward reductions triggered by the occurrence of an instruction of the form roll k, where k refers to a previously instantiated memory. In roll-π, we have for instance the following forward and backward steps, where $M = (k_1 : a\langle Q\rangle) \mid (k_2 : a(X) \triangleright_\gamma X \mid \mathsf{roll}\ \gamma)$:

$$\begin{array}{l} M \longrightarrow \nu k.\, (k : Q \mid \mathsf{roll}\ k) \mid [M; k] \longrightarrow \\ \nu k\, k_3\, k_4.\, k \prec (k_3, k_4) \mid k_3 : Q \mid k_4 : \mathsf{roll}\ k \mid [M; k] \longrightarrow M \end{array}$$

The communication between threads k_1 and k_2 in the first step and the split of process k into k_3 and k_4 are perfectly undone by the third (backward) step.

More generally, the set of memories and connectors of a configuration M provides us with an ordering $<:$ between the keys of M that reflects their causal dependency: $k' <: k$ means that key k' has key k as *causal descendant*. Thus, the effects of a rollback can be characterized as follows. When a rollback takes place in a configuration M, triggered by an instruction k_r : roll k, it suppresses all threads and processes whose tag is a causal descendant of k, as well as all connectors $k' \prec (k_1, k_2)$ and memories $m = [k_1 : \tau_1 \mid k_2 : \tau_2; k']$ whose key k' is a causal descendant of k. When suppressing such a memory m, the rollback operation may release a thread $k_i : \tau_i$ if k_i is not a causal descendant of k (at least one of the threads of m must have k as causal antecedent if k' has k as causal antecedent). This is due to the fact that a thread that is not a causal descendant of k may be involved in a communication (and then captured into a memory) by a descendant of k. This thread can be seen as a resource that is taken from the environment through interaction, and it should be restored in case of rollback. Finally, rolling-back also releases the content μ of the memory $[\mu; k]$ targeted by the roll, reversing the corresponding communication step.

[6] Work can be done to store memories in a more efficient way. We will not consider this issue in the current paper; an approach can be found in [22].

Flexible reversibility in croll-π. In roll-π, a rollback perfectly undoes a computation originated by a specific message receipt. However, nothing prevents the same computation from taking place again and again (although not necessarily in the same context, as independent computations may have proceeded on their own in parallel). To allow for flexible reversibility, we extend roll-π with a single new construct, called a *message with alternative*. In croll-π, a message may now take the form $a\langle P\rangle \div C$, where alternative C may either be a message $c\langle Q\rangle \div \mathbf{0}$ with null alternative or the null process $\mathbf{0}$. When the message receipt of $k : a\langle P\rangle \div C$ is rolled-back, configuration $k : C$ is released instead of the original $k : a\langle P\rangle$, as would be the case in roll-π. (Only the alternative associated to the message in the memory $[\mu; k]$ targeted by the roll is released: other processes may be restored, but not modified.) For example, if $M = (k_1 : a\langle Q\rangle \div \mathbf{0}) \mid (k_2 : a(X) \triangleright_\gamma X \mid \mathsf{roll}\ \gamma)$ then we have the following computation, where the communication leading to the rollback becomes disabled.

$$\begin{aligned}
&M \longrightarrow \nu k.\,(k : Q \mid \mathsf{roll}\ k) \mid [M; k] \longrightarrow\\
&\nu k\, k_3\, k_4.\, k \prec (k_3, k_4) \mid k_3 : Q \mid k_4 : \mathsf{roll}\ k \mid [M; k] \longrightarrow\\
&k_1 : \mathbf{0} \mid (k_2 : a(X) \triangleright_\gamma X \mid \mathsf{roll}\ \gamma)
\end{aligned}$$

We will show that croll-π is powerful enough to devise various kinds of alternatives (see Section 4), whose implementation is not possible in roll-π (cf. Theorem 2). Also, thanks to the higher-order aspect of the calculus, the behavior of roll-π can still be programmed: rigid reversibility can be seen as a particular case of flexible reversibility. Thus, the introduction of messages with alternatives has limited impact on the definition of the syntax and of the operational semantics, but it has a strong impact on what can actually be modeled in the calculus and on its theory.

3 The croll-π Calculus: Syntax and 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 set $\mathcal{V}_\mathcal{K}$ of *key variables*, and the set $\mathcal{V}_\mathcal{P}$ of *process variables*. $\mathbb{N}$ denotes the set of natural numbers. 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 $\mathcal{N} \cup \mathcal{K}$; γ range over $\mathcal{V}_\mathcal{K}$; X, Y, Z range over $\mathcal{V}_\mathcal{P}$. We denote by $\tilde{u}$ a finite set $u_1 \ldots u_n$.

Syntax. The syntax of the croll-π calculus is given in Figure 1. *Processes*, given by the P, Q productions, are the standard processes of the asynchronous higher-order π-calculus [27], except for the presence of the roll primitive, the extra bound tag variable in triggers, and messages with alternative that replace roll-π messages $a\langle P\rangle$. The alternative operator $\div$ binds more strongly than any other operator. *Configurations* in croll-π are given by the M, N productions. A configuration is built up from *tagged processes* $k : P$, *memories* $[\mu; k]$, and *connectors*

$$
\begin{array}{rl}
P, Q ::= & \mathbf{0} \;\mid\; X \;\mid\; \nu a.\, P \;\mid\; (P \mid Q) \;\mid\; a(X) \triangleright_\gamma P \;\mid\; a\langle P\rangle \div C \;\mid\; \mathsf{roll}\ k \;\mid\; \mathsf{roll}\ \gamma \\
M, N ::= & \mathbf{0} \;\mid\; \nu u.\, M \;\mid\; (M \mid N) \;\mid\; k : P \;\mid\; [\mu; k] \;\mid\; k \prec (k_1, k_2) \qquad C ::= a\langle P\rangle \div \mathbf{0} \;\mid\; \mathbf{0} \\
\mu ::= & (k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q) \\
& a, b, c \in \mathcal{N} \quad X, Y, Z \in \mathcal{V}_{\mathcal{P}} \quad \gamma \in \mathcal{V}_{\mathcal{K}} \quad u, v, w \in \mathcal{N} \cup \mathcal{K} \quad h, k, l \in \mathcal{K}
\end{array}
$$

Fig. 1. Syntax of $\mathsf{croll}\text{-}\pi$

$k \prec (k_1, k_2)$. In a memory $[\mu; k]$, we call μ the *configuration part* of the memory and k its *key*. $\mathcal{P}$ denotes the set of $\mathsf{croll}\text{-}\pi$ processes and $\mathcal{C}$ the set of $\mathsf{croll}\text{-}\pi$ configurations. We let (together with their decorated variants) P, Q, R range over $\mathcal{P}$ and L, M, N range over $\mathcal{C}$. We call *thread* a process that is either a message with alternative $a\langle P\rangle \div C$, a trigger $a(X) \triangleright_\gamma P$, or a rollback instruction $\mathsf{roll}\ k$. We let τ and its decorated variants range over threads. We write $\prod_{i \in I} M_i$ for the parallel composition of configurations M_i for each $i \in I$ (by convention $\prod_{i \in I} M_i = \mathbf{0}$ if $I = \emptyset$), and we abbreviate $a\langle \mathbf{0}\rangle$ to $\overline{a}$.

Free identifiers and free variables. Notions of free identifiers and free variables in $\mathsf{croll}\text{-}\pi$ are standard. 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 key variable γ with scope P. We denote by $\mathtt{fn}(P)$ and $\mathtt{fn}(M)$ the set of free names and keys of process P and configuration M, respectively. Note in particular that $\mathtt{fn}(k : P) = \{k\} \cup \mathtt{fn}(P)$, $\mathtt{fn}(\mathsf{roll}\ k) = \{k\}$. We say that a process P or a configuration M is *closed* if it has no free (process or key) variable. We denote by $\mathcal{P}_{cl}$ and $\mathcal{C}_{cl}$ the sets of closed processes and configurations, respectively. We abbreviate $a(X) \triangleright_\gamma P$, where X is not free in P, to $a \triangleright_\gamma P$; and $a(X) \triangleright_\gamma P$, where γ is not free in P, to $a(X) \triangleright P$.

Remark 1. We have no construct for replicated processes or internal choice in $\mathsf{croll}\text{-}\pi$: as in the higher-order π-calculus, 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 Reduction Semantics

The reduction semantics of $\mathsf{croll}\text{-}\pi$ is defined via a reduction relation $\longrightarrow$, which is a binary relation over closed configurations ($\longrightarrow \subset \mathcal{C}_{cl} \times \mathcal{C}_{cl}$), and a structural congruence relation $\equiv$, which is a binary relation over configurations ($\equiv \subset \mathcal{C} \times \mathcal{C}$). We define *configuration contexts* as "configurations with a hole $\bullet$", given by the grammar: $\mathbb{C} ::= \bullet \mid (M \mid \mathbb{C}) \mid \nu u.\, \mathbb{C}$. *General contexts* $\mathbb{G}$ are just configurations with a hole $\bullet$ in a place where an arbitrary process P can occur. A *congruence* on processes or configurations is an equivalence relation $\mathcal{R}$ that

$$(\text{E.P}\textsc{ar}\text{C})\ M \mid N \equiv N \mid M \qquad (\text{E.P}\textsc{ar}\text{A})\ M_1 \mid (M_2 \mid M_3) \equiv (M_1 \mid M_2) \mid M_3$$

$$(\text{E.N}\textsc{il}\text{M})\ M \mid \mathbf{0} \equiv M \qquad (\text{E.N}\textsc{ew}\text{N})\ \nu u.\, \mathbf{0} \equiv \mathbf{0}$$

$$(\text{E.N}\textsc{ew}\text{C})\ \nu u.\, \nu v.\, M \equiv \nu v.\, \nu u.\, M \qquad (\text{E.N}\textsc{ew}\text{P})\ (\nu u.\, M) \mid N \equiv \nu u.\, (M \mid N)$$

$$(\text{E.}\alpha)\ M =_\alpha N \implies M \equiv N \qquad (\text{E.T}\textsc{ag}\text{C})\ k \prec (k_1, k_2) \equiv k \prec (k_2, k_1)$$

$$(\text{E.T}\textsc{ag}\text{A})\ \nu h.\, k \prec (h, k_3) \mid h \prec (k_1, k_2) \equiv \nu h.\, k \prec (k_1, h) \mid h \prec (k_2, k_3)$$

Fig. 2. Structural congruence for croll-π.

$$(\text{S.C}\textsc{om})\quad \frac{\mu = (k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q_2)}{(k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q_2) \longrightarrow \nu k.\, (k : Q_2\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$(\text{S.T}\textsc{ag}\text{N})\ k : \nu a.\, P \longrightarrow \nu a.\, k : P$$

$$(\text{S.T}\textsc{ag}\text{P})\ k : P \mid Q \longrightarrow \nu k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

$$(\text{S.R}\textsc{oll})\quad \frac{k <: N \qquad \texttt{complete}(N \mid [\mu; k] \mid (k_r : \mathsf{roll}\ k)) \qquad \mu' = \texttt{xtr}(\mu)}{N \mid [\mu; k] \mid (k_r : \mathsf{roll}\ k) \longrightarrow \mu' \mid N \lightning_k}$$

$$(\text{S.C}\textsc{tx})\ \frac{M \longrightarrow N}{\mathbb{C}[M] \longrightarrow \mathbb{C}[N]} \qquad (\text{S.E}\textsc{qv})\ \frac{M \equiv M' \qquad M' \longrightarrow N' \qquad N' \equiv N}{M \longrightarrow N}$$

Fig. 3. Reduction rules for croll-π

is closed for general or configuration contexts: $P \mathcal{R} Q \implies \mathbb{G}[P] \mathcal{R} \mathbb{G}[Q]$ and $M \mathcal{R} N \implies \mathbb{C}[M] \mathcal{R} \mathbb{C}[N]$.

Structural congruence $\equiv$ is defined as the smallest congruence on configurations that satisfies the axioms in Figure 2, where $t =_\alpha t'$ denotes equality of t and t' modulo α-conversion. Axioms E.PARC to E.α are standard from the π-calculus. Axioms E.TAGC and E.TAGA model commutativity and associativity of connectors, in order not to have a rigid tree structure. Thanks to axiom E.NEWC, $\nu\tilde{u}.\, A$ stands for $\nu u_1 \ldots u_n.\, A$ if $\tilde{u} = u_1\, \ldots\, u_n$.

Configurations can be written in normal form using structural congruence.

Lemma 1 (Normal form). *Given a configuration M, we have:*

$$M \equiv \nu\tilde{n}.\, \prod_i (k_i : P_i) \mid \prod_j [\mu_i; k_j] \mid \prod_l k_l \prec (k'_l, k''_l)$$

The reduction relation $\longrightarrow$ is defined as the smallest binary relation on closed configurations satisfying the rules of Figure 3. This extends the naïve semantics of roll-π introduced in [19],[7] and outlined here in Section 2, to manage alternatives. We denote by $\Longrightarrow$ the reflexive and transitive closures of $\longrightarrow$.

Reductions are either forward, given by rules S.Com, S.TagN, and S.TagP, or backward, defined by rule S.Roll. They are closed under configuration contexts (rule S.Ctx) and under structural congruence (rule S.Eqv). The rule for communication S.Com is the standard communication rule of the higher-order π-calculus with the side effects discussed in Section 2. Rule S.TagN allows restrictions in processes to be lifted at the configuration level. Rule S.TagP allows to split parallel processes. Rule S.Roll enacts rollback, canceling all the effects of the interaction identified by the unique key k, and releasing the initial configuration that gave rise to the interaction, where the alternative replaces the original message. This is the only difference between croll-π and roll-π: in the latter, the memory μ was directly released. However, this small modification yields significant changes to the expressive power of the calculus, as we will see later.

The rollback impacts only the causal descendants of k, defined as follows.

Definition 1 (Causal dependence). *Let M be a configuration and let $\mathcal{T}_M$ be the set of keys occurring in M. Causal dependence $<:_M$ is the reflexive and transitive closure of $<_M$, which is defined as the smallest binary relation on $\mathcal{T}_M$ satisfying the following clauses:*

- *$k <_M k'$ if $k \prec (k_1, k_2)$ occurs in M with $k' = k_1$ or $k' = k_2$;*
- *$k <_M k'$ if a thread $k : P$ occurs (inside μ) in a memory $[\mu; k']$ of M.*

If the configuration M is clear from the context, we write $k <: k'$ for $k <:_M k'$.

A backward reduction triggered by roll k involves *all* and *only* the descendants of key k. We ensure they are all selected by requiring that the configuration is *complete*, and that no other term is selected by requiring *k-dependence*.

Definition 2 (Complete configuration). *A configuration M is* complete*, denoted as $\texttt{complete}(M)$, if, for each memory $[\mu; k]$ and each connector $k' \prec (k, k_1)$ or $k' \prec (k_1, k)$ that occurs in M there exists in M either a connector $k \prec (h_1, h_2)$ or a tagged process $k : P$ (possibly inside a memory).*

A configuration M is k-dependent if all its components depend on k.

Definition 3 (k-dependence). *Let M be a configuration such that:*
$M \equiv \nu\tilde{u}.\ \prod_{i \in I}(k_i : P_i) \mid \prod_{j \in J}[\mu_j; k_j] \mid \prod_{l \in L} k_l \prec (k'_l, k''_l)$ with $k \notin \tilde{u}$.
Configuration M is k-dependent, written $k <: M$ by overloading the notation for causal dependence among keys, if for every i in $I \cup J \cup L$, we have $k <:_M k_i$.

Rollback should release all the resources consumed by the computation to be rolled-back which were provided by other threads. They are computed as follows.

[7] We extend the naïve semantics instead of the high-level or the low-level semantics (also defined in [19]) for the sake of simplicity. However, reduction semantics corresponding to the high-level and low-level semantics of roll-π can similarly be specified.

Definition 4 (Projection). *Let M be a configuration such that:* $M \equiv \nu\tilde{u}.\prod_{i\in I}(k_i : P_i) \mid \prod_{j\in J}[k'_j : R_j \mid k''_j : T_j; k_j] \mid \prod_{l\in L} k_l \prec (k'_l, k''_l)$ *with* $k \notin \tilde{u}$. *Then:*

$$M\lightning_k = \nu\tilde{u}.\,(\prod_{j'\in J'} k'_{j'} : R_{j'}) \mid (\prod_{j''\in J''} k''_{j''} : T_{j''})$$

where $J' = \{j \in J \mid k \not<: k'_j\}$ *and* $J'' = \{j \in J \mid k \not<: k''_j\}$.

Intuitively, $M\lightning_k$ consists of the threads inside memories in M which are not dependent on k.

Finally, and this is the main novelty of croll-π, function `xtr` defined below replaces messages from the memory targeted by the roll by their alternatives.

Definition 5 (Extraction function).

$$\mathtt{xtr}(M \mid N) = \mathtt{xtr}(M) \mid \mathtt{xtr}(N) \qquad \mathtt{xtr}(k : a\langle P\rangle \div C) = k : C$$
$$\mathtt{xtr}(k : a(X) \rhd_\gamma Q) = k : a(X) \rhd_\gamma Q$$

No other case needs to be taken into account as `xtr` is only called on the contents of memories.

Remark 3. Not all syntactically licit configurations make sense. In particular, we expect configurations to respect the causal information required for executing croll-π programs. We therefore work only with *coherent* configurations. A configuration is coherent if it is obtained by reduction starting from a configuration of the form $\nu k.\, k : P$ where P is closed and contains no roll h primitive (all the roll primitives should be of the form roll γ).

3.3 Barbed Congruence

We define notions of strong and weak barbed congruence to reason about croll-π processes and configurations. Name a is *observable* in configuration M, denoted as $M{\downarrow_a}$, if $M \equiv \nu\tilde{u}.\,(k : a\langle P\rangle \div C) \mid N$, with $a \notin \tilde{u}$. We write $M\mathcal{R}{\downarrow_a}$, where $\mathcal{R}$ is a binary relation on configurations, if there exists N such that $M\mathcal{R}N$ and $N{\downarrow_a}$. The following definitions are classical.

Definition 6 (Barbed congruences for configurations). *A relation* $\mathcal{R} \subseteq \mathcal{C}_{cl} \times \mathcal{C}_{cl}$ *on closed configurations is a* strong *(respectively* weak*)* barbed simulation *if whenever* $M\,\mathcal{R}\,N$,

- $M{\downarrow_a}$ *implies* $N{\downarrow_a}$ *(respectively* $N \Longrightarrow{\downarrow_a}$*);*
- $M \longrightarrow M'$ *implies* $N \longrightarrow N'$ *(respectively* $N \Longrightarrow N'$*) with* $M'\mathcal{R}N'$.

A relation $\mathcal{R} \subseteq \mathcal{C}_{cl} \times \mathcal{C}_{cl}$ *is a* strong (weak) barbed bisimulation *if* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are strong (weak) barbed simulations. We call* strong (weak) barbed bisimilarity *and denote by* $\sim$ $(\approx)$ *the largest strong (weak) barbed bisimulation. The largest congruence for configuration contexts included in* $\sim$ $(\approx)$ *is called* strong (weak) barbed congruence, *denoted by* $\sim_c$ $(\approx_c)$.

The notion of strong and weak barbed congruence extends to closed and open processes, by considering general contexts that form closed configurations.

Definition 7 (Barbed congruences for processes). *A relation $\mathcal{R} \subseteq \mathcal{P}_{cl} \times \mathcal{P}_{cl}$ on closed processes is a* strong *(resp.* weak*) barbed congruence if whenever $P\mathcal{R}Q$, for all general contexts $\mathbb{G}$ such that $\mathbb{G}[P]$ and $\mathbb{G}[Q]$ are closed configurations, we have $\mathbb{G}[P] \sim_c \mathbb{G}[Q]$ (resp. $\mathbb{G}[P] \approx_c \mathbb{G}[Q]$).*

Two open processes P and Q are said to be strong (resp. weak) barbed congruent, denoted by $P \sim_c^o Q$ (resp. $P \approx_c^o Q$) if for all substitutions σ such that $P\sigma$ and $Q\sigma$ are closed, we have $P\sigma \sim_c Q\sigma$ (resp. $P\sigma \approx_c Q\sigma$).

Working with arbitrary contexts can quickly become unwieldy. We offer the following Context Lemma to simplify the proofs of congruence.

Theorem 1 (Context lemma). *Two processes P and Q are weak barbed congruent, $P \approx_c^o Q$, if and only if for all substitutions σ such that $P\sigma$ and $Q\sigma$ are closed, all closed configurations M, and all keys k, we have: $M \mid (k : P\sigma) \approx M \mid (k : Q\sigma)$.*

The proof of this Context Lemma is much more involved than the corresponding one in the π-calculus, notably because of the bookkeeping required in dealing with process and thread tags. It is obtained by composing the lemmas below.

The first lemma shows that the only relevant configuration contexts are parallel contexts.

Lemma 2 (Context lemma for closed configurations). *For any closed configurations M, N, $M \sim_c N$ if and only if, for all closed configurations L, $M \mid L \sim N \mid L$. Likewise, $M \approx_c N$ if and only if, for all L, $M \mid L \approx N \mid L$.*

Proof. The left to right implication is immediate, by definition of $\sim_c$. For the other direction, the proof consists in showing that $\mathcal{R} = \{\langle \mathbb{C}[M], \mathbb{C}[N]\rangle \mid \forall L, M \mid L \sim N \mid L\}$ is included in $\sim$. The weak case is identical to the strong one. □

We can then prove the thesis on closed processes.

Lemma 3 (Context lemma for closed processes). *Let P and Q be closed processes. We have $P \approx_c Q$ if and only if, for all closed configuration contexts $\mathbb{C}$ and $k \notin \mathtt{fn}(P, Q)$, we have $\mathbb{C}[k : P] \approx \mathbb{C}[k : Q]$.*

Proof. The left to right implication is clear. One can prove the right to left direction by induction on the form of general contexts for processes, using the factoring lemma below for message contexts. □

Lemma 4 (Factoring). *For all closed processes P, all closed configurations M such that $M\{^P/_X\}$ is closed, and all $c, t, k, k' \notin \mathtt{fn}(M, P)$, we have*

$$M\{^P/_X\} \approx_c \nu c, t, k_0, k_0'.\, M\{^{\overline{c}}/_X\} \mid k_0 : t\langle Y_P\rangle \mid k_0' : Y_P$$

where $Y_P = t(Y) \triangleright (c \triangleright P) \mid t\langle Y\rangle \mid Y$.

We then deal with open processes.

Lemma 5 (Context lemma for open processes). *Let P and Q be (possibly open) processes. We have $P \approx_c^o Q$ if and only if for all closed configuration contexts $\mathbb{C}$, all substitutions σ such that $P\sigma$ and $Q\sigma$ are closed, and all $k \notin \mathtt{fn}(P, Q)$, we have $\mathbb{C}[k : P\sigma] \approx \mathbb{C}[k : Q\sigma]$.*

Proof. For the only if part, one proceeds by induction on the number of bindings in σ. The case for zero bindings follows from Lemma 3. For the inductive case, we write $\mathbb{P}[\bullet]$ for a process where an occurrence of $\mathbf{0}$ has been replaced by $\bullet$, and we show that contexts of the form $\mathbb{P} = a\langle R\rangle \mid a(X) \triangleright \mathbb{P}'[\bullet]$ where a is fresh and $\mathbb{P} = a\langle R\rangle \mid a(X) \triangleright_\gamma \mathbb{P}'[\bullet]$ where a is fresh and X never occurs in the continuation actually enforce the desired binding.

For the if part, the proof is by induction on the number of triggers. If the number of triggers is 0 then the thesis follows from Lemma 3. The inductive case consists in showing that equivalence under substitutions ensures equivalence under a trigger context. □

Proof (of Theorem 1). A direct consequence of Lemma 5 and Lemma 2. □

4 croll-π Expressiveness

4.1 Alternative Idioms

The message with alternative $a\langle P\rangle \div C$ triggers alternative C upon rollback. We choose to restrict C to be either a message with $\mathbf{0}$ alternative or $\mathbf{0}$ itself in order to have a minimal extension of roll-π. However, this simple form of alternative is enough to encode far more complex alternative policies and constructs, as shown below. We define the semantics of the alternative idioms below by only changing function `xtr` in Definition 5. We then encode them in croll-π and prove the encoding correct w.r.t. weak barbed congruence. More precisely, for every extension below the notion of barbs is unchanged. The notion of barbed bisimulation thus relates processes with slightly different semantics (only `xtr` differs) but sharing the same notion of barbs. Since we consider extensions of croll-π, in weak barbed congruence we consider just closure under croll-π contexts. By showing that the extensions have the same expressive power of croll-π, we ensure that allowing them in contexts would not change the result. Every encoding maps unmentioned constructs homomorphically to themselves. After having defined each alternative idiom, we freely use it as an abbreviation.

Arbitrary alternatives. Messages with arbitrary alternative can be defined by allowing C to be any process Q. No changes are required to the definition of function `xtr`. We can encode arbitrary alternatives as follows, where c is not free in P, Q.

$$(\!|a\langle P\rangle \div Q|\!)_{aa} = \nu c.\, a\langle (\!|P|\!)_{aa}\rangle \div c\langle (\!|Q|\!)_{aa}\rangle \div \mathbf{0} \mid c(X) \triangleright X$$

Proposition 1. *$P \approx_c (\!|P|\!)_{aa}$ for any closed process with arbitrary alternatives.*

$$
\begin{aligned}
\mathcal{R} &= \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \cup \mathcal{R}_4 \cup \mathcal{R}_5 \cup Id \\
\mathcal{R}_1 &= \{\langle k : a\langle P\rangle \div Q \mid L \;,\; k : (\nu c.\, a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid c(X) \triangleright X) \mid L\rangle\} \\
\mathcal{R}_2 &= \{\langle k : a\langle P\rangle \div Q \mid L \;,\; \nu c\, k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid k_2 : c(X) \triangleright X \mid L\rangle\} \\
\mathcal{R}_3 &= \{\langle \nu h.\, [k : a\langle P\rangle \div Q \mid k' : a(X) \triangleright_\gamma R; h] \mid L'' \;, \\
&\qquad \nu c\, k_1\, k_2\, h.\, k \prec (k_1, k_2) \mid [k_1 : a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid k' : a(X) \triangleright_\gamma R; h] \mid k_2 : c(X) \triangleright X \mid L''\rangle\} \\
\mathcal{R}_4 &= \{\langle k : Q \mid L''' \;,\; \nu c\, k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : c\langle Q\rangle \div \mathbf{0} \mid k_2 : c(X) \triangleright X \mid L'''\rangle\} \\
\mathcal{R}_5 &= \{\langle k : Q \mid L''' \;,\; \nu c\, k_1\, k_2\, h.\, k \prec (k_1, k_2) \mid [k_1 : c\langle Q\rangle \div \mathbf{0} \mid k_2 : c(X) \triangleright X; h] \mid h : Q \mid L'''\rangle\}
\end{aligned}
$$

Fig. 4. Bisimulation relation for arbitrary alternatives.

Proof. We consider just one instance of arbitrary alternative, the thesis will follow by transitivity.

Thanks to Lemma 5 and Lemma 2, we only need to prove that for all closed configurations L and $k \notin \texttt{fn}(P)$, we have $k : a\langle P\rangle \div Q \mid L \approx k : (\nu c.\, a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid c(X) \triangleright X) \mid L$. We consider the relation $\mathcal{R}$ in Figure 4 and prove that it is a weak barbed bisimulation. In every relation, L is closed and $k \notin \texttt{fn}(P)$.

In $\mathcal{R}_1$, the right configuration can reduce via rule S.TagN followed by S.TagP. These lead to $\mathcal{R}_2$. Performing these reductions is needed to match the barb and the relevant reductions of the left configuration, thus we consider directly $\mathcal{R}_2$. In $\mathcal{R}_2$ the barbs coincide. Rollbacks lead to the identity. The only possible communication is on a, and requires $L \equiv L' \mid k' : a(X) \triangleright_\gamma R$. It leads to $\mathcal{R}_3$, where $L'' = L' \mid R\{^{P,h}/_{X,\gamma}\}$. In $\mathcal{R}_3$ the barbs coincide too. All the reductions can be matched by staying in $\mathcal{R}_3$ or going to the identity, but for executing a roll with key h. This leads to $\mathcal{R}_4$. From $\mathcal{R}_4$ we can always execute the internal communication at c leading to $\mathcal{R}_5$. The thesis follows from the result below, whose proof requires again to find a suitable bisimulation relation.

Lemma 6. *For each configuration M k-dependent and complete such that k', t, $k_1, k_2 \notin \texttt{fn}(M)$ we have $M \approx_c \nu k'\, t\, k_1\, k_2.\, k \prec (k_1, k_2) \mid [k_1 : t\langle Q\rangle \div C \mid k_2 : t(X) \triangleright R; k'] \mid M\{^{k'}/_k\}$.*

□

Proofs concerning other idioms follow similar lines, and can be found in the online technical report [18].

A particular case of arbitrary alternative $a\langle P\rangle \div Q$ is when Q is a message whose alternative is not $\mathbf{0}$. By applying this pattern recursively we can write $a_1\langle P_1\rangle \div \ldots \div a_n\langle P_n\rangle \div Q$. In particular, by choosing $a_1 = \cdots = a_n$ and $P_1 = \cdots = P_n$ we can try n times the alternative P before giving up by executing Q.

Endless retry. We can also retry the same alternative infinitely many times, thus obtaining the behavior of roll-π messages. These messages can be integrated into croll-π semantics by defining function $\texttt{xtr}$ as the identity on them.

$$(\!|a\langle P\rangle|\!)_{er} = \nu t.\, Y \mid a\langle (\!|P|\!)_{er}\rangle \div t\langle Y\rangle \qquad Y = t(Z) \triangleright Z \mid a\langle (\!|P|\!)_{er}\rangle \div t\langle Z\rangle$$

Proposition 2. *$P \approx_c \llparenthesis P \rrparenthesis_{er}$ for any closed process with* roll-π *messages.*

As corollary of Proposition 2 we thus have the following.

Corollary 1. croll-π *is a conservative extension of* roll-π*.*

Triggers with alternative. Until now we attached alternatives to messages. Symmetrically, one may attach alternatives to triggers. Thus, upon rollback, the message is released and the trigger is replaced by a new process.

The syntax for triggers with alternative is $(a(X) \triangleright_\gamma Q) \div b\langle Q' \rangle \div \mathbf{0}$. As for messages, we use a single message as alternative, but one can use general processes as described earlier. Triggers with alternative are defined by the extract clause below.

$$\mathtt{xtr}(k : (a(X) \triangleright_\gamma Q) \div b\langle Q' \rangle \div \mathbf{0}) = k : b\langle Q' \rangle \div \mathbf{0}$$

Interestingly, messages with alternative and triggers with alternative may coexist. The encoding of triggers with alternative is as follows.

$$\llparenthesis (a(X) \triangleright_\gamma Q) \div b\langle Q' \rangle \div \mathbf{0} \rrparenthesis_{at} = \nu c\, d.\, \overline{c} \div \overline{d} \div \mathbf{0} \mid (c \triangleright_\gamma a(X) \triangleright \llparenthesis Q \rrparenthesis_{at}) \mid (d \triangleright b\langle \llparenthesis Q' \rrparenthesis_{at} \rangle \div \mathbf{0})$$

Proposition 3. *$P \approx_c \llparenthesis P \rrparenthesis_{at}$ for any closed process with triggers with alternative.*

4.2 Comparing croll-π and roll-π

While Corollary 1 shows that croll-π is at least as expressive as roll-π, a natural question is whether croll-π is actually strictly more expressive than roll-π or not. The theorem below gives a positive answer to this question.

Theorem 2. *There is no encoding $\llparenthesis \bullet \rrparenthesis$ from* croll-π *to* roll-π *such that for each* croll-π *configuration M:*

1. *if M has a computation including at least a backward step, then $\llparenthesis M \rrparenthesis$ has a computation including at least a backward step;*
2. *if M has only finite computations, then $\llparenthesis M \rrparenthesis$ has only finite computations.*

Proof. Consider configuration $M = \nu k.\, k : \overline{a} \div \overline{b} \div \mathbf{0} \mid a \triangleright_\gamma \mathsf{roll}\ \gamma$. This configuration has a unique possible computation, composed by one forward step followed by one backward step. Assume towards a contradiction that an encoding exists and consider $\llparenthesis M \rrparenthesis$. $\llparenthesis M \rrparenthesis$ should have at least a computation including a backward step. From roll-π loop lemma [19, Theorem 1], if we have a backward step, we are able to go forward again, and then there is a looping computation. This is in contrast with the second condition of the encoding. The thesis follows. □

The main point behind this result is that the Loop Lemma, a cornerstone of roll-π theory [19] capturing the essence of rigid rollback (and similar results in [9, 20, 22, 24]), does not hold in croll-π. Naturally, the result above does not imply that croll-π cannot be encoded in HOπ or in π-calculus. However, these calculi are too low level for us, as hinted at by the fact that the encoding of a simple reversible higher order calculus into HOπ is quite complex, as shown in [20].

$$\begin{aligned}
Q_i \;\triangleq\;& (act_i(z_i) \triangleright p_i\langle i,1\rangle \div \ldots \div p_i\langle i,8\rangle \div f_i\langle \mathbf{0}\rangle \div \mathbf{0} \mid \\
& \quad (p_i(\mathbf{x_i}) \triangleright_{\gamma_i} act_{i+1}\langle \mathbf{0}\rangle \mid f_{i+1}(z) \triangleright \mathsf{roll}\ \gamma_i \mid ok_i(w_1) \triangleright \ldots ok_i(w_{i-1}) \triangleright ! c_i\langle \mathbf{x_i}\rangle \div \mathbf{0} \mid \\
& \quad\quad \textstyle\prod_{j=1}^{i-1} c_j(\mathbf{y_j}) \triangleright \texttt{if}\ err(\mathbf{x_i},\mathbf{y_j})\ \texttt{then}\ \mathsf{roll}\ \gamma_i\ \texttt{else}\ ok_i\langle \mathbf{0}\rangle \div \mathbf{0})) \\
& err((x_1,x_2),(y_1,y_2)) \;\triangleq\; (x_1 = y_1 \vee x_2 = y_2 \vee |x_1 - y_1| = |x_2 - y_2|)
\end{aligned}$$

Fig. 5. The i-th queen

5 Programming in croll-π

A main goal of croll-π is to make reversibility techniques exploitable for application development. Even if croll-π is not yet a full-fledged language, we have developed a proof-of-concept interpreter for it. To the best of our knowledge, this is the first interpreter for a causal-consistent reversible language. We then put the implementation at work on a few simple, yet interesting, programming problems. We detail below the algorithm we devised to solve the Eight Queens problem [4, p. 165]. The interpreter and the code for solving the Eight Queens problem are available at `http://proton.inrialpes.fr/~mlienhar/croll-pi/implem`, together with examples of encodings of primitives for error handling, and an implementation of the car repair scenario of the EU project Sensoria.

The interpreter for croll-π is written in Maude [11], a language based on both equational and rewriting logic that allows the programmer to define terms and reduction rules, e.g., to execute reduction semantics of process calculi. Most of croll-π's rules are straightforwardly interpreted, with the exception of rule S.Roll. This rule is quite complex as it involves checks on an unbounded number of interacting components. Such an issue is already present in roll-π [19], where it is addressed by providing an easier to implement, yet equivalent, low-level semantics. This semantics replaces rule S.Roll with a protocol that sends notifications to all the involved components to roll-back, then waits for them to do so. Extending the low-level semantics from roll-π to croll-π simply requires the application of function `xtr` to the memory targeted by the rollback. We do not detail the low-level semantics of croll-π here, and refer the reader to [19] for a detailed description in the setting of roll-π. Our Maude interpreter is based on this low-level semantics, extended with values (integers and pairs) and with the `if-then-else` construct. It is fairly concise (less than 350 lines of code).

The Eight Queens problem is a well-known constraint-programming probem which can be formulated as follows: how to place 8 queens on an 8 × 8 chess board so that no queen can directly capture another? We defined an algorithm in croll-π where queens are concurrent entities, numbered from 1 to 8, all executing the code schema shown in Figure 5. We use $\mathbf{x}$ to indicate a pair of integer variables (x_1, x_2), and replicated messages $!c_i\langle\mathbf{x}\rangle \div \mathbf{0}$ to denote the encoding of a parallel composition of an infinite number of messages $c_i\langle\mathbf{x}\rangle \div \mathbf{0}$ (cf. Remark 1).

The queens are activated in numeric order. The i-th queen is activated by messages on channels act_i from its predecessor. When a queen is activated it looks for its position by trying sequentially all the positions in the i-th row of the chess board. To try a position, it sends it over channel p_i and checks whether the position conflicts with the choices of the other queens. This is done by computing (in parallel) $err(\mathbf{x_j}, \mathbf{x_i})$ for each $j < i$. If a check fails, $\mathsf{roll}\ \gamma_i$ rolls-back the choice of the position. The alternatives mechanism allows to try the next position. If no suitable position is available, the choice of position of the previous queen is rolled-back (possibly recursively) by the communication over f_i. If instead the check succeeds, it generates a message on channel ok_i. When there are exactly $i - 1$ messages on the channel ok_i, the queen commits its position on c_i.

6 Asynchronous Interacting Transactions

This section shows how croll-π can model in a precise way interacting transactions with compensations as formalized in TransCCS [14]. Actually, the natural croll-π encoding improves on the semantics in [14], since croll-π causality tracking is more precise than the one in TransCCS, which is based on dynamic embedding of processes into transactions. Thus croll-π avoids some spurious undo of actions, as described below. Before entering the details of TransCCS, let us describe the general idea of transaction encoding.

We consider a very general notion of atomic (but not necessarily isolated) transaction, i.e., a process that executes completely or not at all. Informally, a transaction $[P, Q]_\gamma$ with name γ executing process P with compensation Q can be modeled by a process of the form:

$$[P, Q]_\gamma = \nu a\, c.\, \overline{a} \div \overline{c} \div \mathbf{0} \mid (a \triangleright_\gamma P) \mid (c \triangleright Q)$$

Intuitively, when $[P, Q]_\gamma$ is executed, it first starts process P under the rollback scope γ. Abortion of the transaction can be triggered in P by executing a $\mathsf{roll}\ \gamma$. Whenever P is rolled-back, the rollback does not restart P (since the message on a is substituted by the alternative on c), but instead starts the compensation process Q. In this approach commit is implicit: when there is no reachable $\mathsf{roll}\ \gamma$, the transaction is committed. From the explanation above, it should be clear that in the execution of $[P, Q]_\gamma$, either P executes completely, i.e., until it reaches a commit, or not at all, in the sense that it is perfectly rolled-back. If P is ever rolled-back, its failed execution can be compensated by that of process Q. Interestingly, and in contrast with irreversible actions used in [13], our rollback scopes can be nested without compromising this all-or-nothing semantics.

Let us now consider an asynchronous fragment of TransCCS [14], removing choice and recursion. Dealing with the whole calculus would not add new difficulties related to rollback, but only related to the encoding of such operators in higher-order π. The syntax of the fragment of TransCCS we consider is:

$$P ::= \mathbf{0} \;\mid\; \nu a.\, P \;\mid\; (P \mid Q) \;\mid\; \overline{a} \;\mid\; a.P \;\mid\; \mathsf{co}\ k \;\mid\; \llbracket P \triangleright_k Q \rrbracket$$

Essentially, it extends CCS with a transactional construct $\llbracket P \triangleright_k Q \rrbracket$, executing a transaction with body P, name k and compensation Q, and a commit operator $\mathsf{co}\ k$.

$$\text{(R-Comm)}\quad \overline{a} \mid a.P \longrightarrow P \qquad\qquad \text{(R-Emb)}\quad \frac{k \notin \mathtt{fn}(R)}{[\![P \rhd_k Q]\!] \mid R \longrightarrow [\![P \mid R \rhd_k Q \mid R]\!]}$$

$$\text{(R-Co)}\quad [\![P \mid \mathsf{co}\ k \rhd_k Q]\!] \longrightarrow P \qquad\qquad \text{(R-Ab)}\quad [\![P \rhd_k Q]\!] \longrightarrow Q$$

and is closed under active contexts $\nu a.\,\bullet$, $\bullet \mid Q$ and $[\![\bullet \rhd_k Q]\!]$, and structural congruence.

Fig. 6. Reduction rules for TransCCS

The rules defining the semantics of TransCCS are given in Figure 6. Structural congruence contains the usual rules for parallel composition and restriction. Keep in mind that transaction scope is a binder for its name k, thus k does not occur outside the transaction, and there is no name capture in rules R-Co and R-Emb.

A croll-π transaction $[P, Q]_\gamma$ as above has explicit abort, specified by roll γ, where γ is used as the transaction name, and implicit commit. TransCCS takes different design choices, using non-deterministic abort and programmable commit. Thus we have to instantiate the encoding above.

Definition 8 (TransCCS encoding). *Let P be a TransCCS process. Its encoding $(\!|\bullet|\!)_t$ in croll-π is defined as:*

$$(\!|\nu a.\,P|\!)_t = \nu a.\,(\!|P|\!)_t \qquad (\!|P \mid Q|\!)_t = (\!|P|\!)_t \mid (\!|Q|\!)_t \qquad (\!|\overline{a}|\!)_t = \overline{a}$$
$$(\!|a.P|\!)_t = a \rhd (\!|P|\!)_t \qquad (\!|\mathsf{co}\ l|\!)_t = l(X) \rhd \mathbf{0} \qquad (\!|\mathbf{0}|\!)_t = \mathbf{0}$$

$$(\!|[\![P \rhd_l Q]\!]|\!)_t = [\nu l.\,(\!|P|\!)_t \mid l\langle \mathsf{roll}\ \gamma\rangle \mid l(X) \rhd X, (\!|Q|\!)_t]_\gamma$$

Since in croll-π only configurations can execute, the behavior of P should be compared with $\nu k.\,k : (\!|P|\!)_t$.

In the encoding, abort is always possible since at any time the only occurrence of the roll in the transaction can be activated by a communication on l. On the other hand, executing the encoding of a TransCCS commit disables the roll related to the transaction. This allows to garbage collect the compensation, and thus corresponds to an actual commit. Note, however, that in croll-π the abort operation is not atomic as in TransCCS since the roll related to a transaction first has to be enabled through a communication on l, disabling in this way any possibility to commit, and then it can be executed. Clearly, until the roll is executed, the body of the transaction can continue its execution. To make abort atomic one would need the ability to disable an active roll, as could be done using a (mixed) choice such as $(\mathsf{roll}\ k) + (l \rhd \mathbf{0})$. In this setting an output on l would commit the transaction. Adding choice would not make the reduction semantics more difficult, but its impact on behavioral equivalence has not been studied yet.

The relation between the behavior of a TransCCS process P and of its translation $(\!|P|\!)_t$ is not immediate, not only because of the comment above on atomicity, but also because of the approximate tracking of causality provided by TransCCS.

TransCCS tracks interacting processes using rule (R-Emb): only processes inside the same transaction may interact, and when a process enters the transaction it is saved in the compensation, so that it can be restored in case of abort. However, no check is performed to ensure that the process actually interacts with the transaction code. For instance, a process $\overline{a} \mid a.P$ may enter a transaction $[\![Q \triangleright_k R]\!]$ and then perform the communication at a. Such a communication would be undone in case of abort. This is a spurious undo, since the communication at a is not related to the transaction code. Actually, the same communication could have been performed outside the transaction, and in this case it would not have been undone.

In croll-π encoding, a process is "inside" the transaction with key k if and only if its tag is causally dependent on k. Thus a process enters a transaction only by interacting with a process inside it. For this reason, there is no reduction in croll-π corresponding to rule (R-Emb), and since no process inside the transaction is involved in the reduction at a above, the reduction would not be undone in case of abort, since it actually happens "outside" the transaction. Thus our encoding avoids spurious undo, and computations in croll-π correspond to computations in TransCCS with minimal applications of rule (R-Emb). These computations are however very difficult to characterize because of syntactic constraints. In fact, for two processes inside two parallel transactions k_1 and k_2 to interact, either k_1 should move inside k_2 or vice versa, but in both the cases not only the interacting processes move, as minimality would require, but also all the other processes inside the same transactions have to move. Intuitively, TransCCS approximates the causality relation, which is a dag, using the tree defined by containment. The spurious reductions undone in TransCCS can always be redone so to reach a state corresponding to the croll-π one. In this sense croll-π minimizes the set of interactions undone.

We define a notion of weak barbed bisimilarity ${}_t{\approx}_{c\pi}$ relating a TransCCS process P and a croll-π configuration M. First, we define barbs in TransCCS by the predicate $P{\downarrow_a}$, which is true in the cases below, false otherwise.

$$\begin{array}{ll} \overline{a}{\downarrow_a} & \nu b.\, P{\downarrow_a} \ \text{ if } P{\downarrow_a} \wedge a \neq b \\ P \mid P'{\downarrow_a} \ \text{ if } P{\downarrow_a} \vee P'{\downarrow_a} & [\![P \triangleright_k Q]\!]{\downarrow_a} \ \text{ if } P{\downarrow_a} \wedge a \neq k \end{array}$$

Here, differently from [14], we observe barbs inside the transaction body, to have a natural correspondence with croll-π barbs.

Definition 9. *A relation $\mathcal{R}$ relating TransCCS processes P and croll-π configurations M is a weak barbed bisimulation if and only if for each $(P, M) \in \mathcal{R}$:*

1. *if $P{\downarrow_a}$ then $M \Longrightarrow{\downarrow_a}$;*
2. *if $M{\downarrow_a}$ then $P \Longrightarrow{\downarrow_a}$;*
3. *if $P \longrightarrow P_1$ is derived using rule* (R-Ab) *then $M \Longrightarrow M'$, $P_1 \Longrightarrow P_2$ and $P_2 \mathcal{R} M'$;*
4. *if $P \longrightarrow P_1$ is derived without using rule* (R-Ab) *then $M \Longrightarrow M'$ and $P_1 \mathcal{R} M'$;*
5. *if $M \longrightarrow M'$ then either: (i) $P \mathcal{R} M'$ or (ii) $P \longrightarrow P_1$ and $P_1 \mathcal{R} M'$ or (iii) $M' \longrightarrow M''$, $P \longrightarrow P_1$ and $P_1 \mathcal{R} M''$.*

Weak barbed bisimilarity ${}_t{\approx}_{c\pi}$ *is the largest weak barbed bisimulation.*

The main peculiarities of the definition above are in condition 3, which captures the need of redoing some reductions that are unduly rolled-back in TransCCS, and in case (iii) of condition 5, which forces atomic abort.

Theorem 3. *For each TransCCS process* P*,* $P \; {}_t{\approx}_{c\pi} \; \nu k.\, k : (\!|P|\!)_t$*.*

Proof. The proof has to take into account the fact that different croll-π configurations may correspond to the same TransCCS process. In particular, a TransCCS transaction $[\![P {\triangleright_k} Q]\!]$ is matched in different ways if Q is the original compensation or if part of it is the result of an application of rule (R-Emb).

Thus, in the proof, we give a syntactic characterization of the set of croll-π configurations $(\!|P|\!)^p$ matching a TransCCS process P. Then we show that $\nu k.\, k : (\!|P|\!)_t \in (\!|P|\!)^p$, and that there is a match between reductions of P and the weak reductions of each configuration in $(\!|P|\!)^p$. The proof, in the two directions, is by induction on the rule applied to derive a single step. □

7 Related Work and Conclusion

We have presented a concurrent process calculus with explicit rollback and minimal facilities for alternatives built on a reversible substrate analogous to a Lévy labeling [5] for concurrent computations. We have shown by way of examples how to build more complex alternative idioms and how to use rollback and alternatives in conjunction to encode transactional constructs. In particular, we have developed an analysis of communicating transactions proposed in TransCCS [14]. We also developed a proof-of-concept interpreter of our language and used it to give a concurrent solution of the Eight Queens problem.

Undo or rollback capabilities in sequential languages have a long history (see [21] for an early survey). In a concurrent setting, interest has developed more recently. Works such as [10] introduce logging and process group primitives as a basis for defining fault-tolerant abstractions, including transactions. Ziarek et al. [28] introduce a checkpoint abstraction for concurrent ML programs. Field et al. [16] extend the actor model with checkpointing constructs. Most of the approaches relying instead on a fully reversible concurrent language have already been discussed in the introduction. Here we just recall that models of reversible computation have also been studied in the context of computational biology, e.g., [9]. Also, the effect of reversibility on Hennessy-Milner logic has been studied in [25]. Several recent works have proposed a formal analysis of transactions, including [14] studied in this paper, as well as several other works such as [23, 6, 8] (see [1] for numerous references to the line of work concentrating on software transactional memories). Note that although reversible calculi can be used to implement transactions, they offer more flexibility. For instance, transactional events [15] only allow an all-or-nothing execution of transactions. Moreover, no visible side-effect is allowed during the transaction, as there is no way to specify how to compensate the side-effects of a failed transaction. A reversible calculus with alternatives allows the encoding of such compensations.

With the exception of the seminal work by Danos and Krivine [13] on RCCS, we are not aware of other work exploiting precise causal information as provided by our reversible machinery to analyze recovery-oriented constructs. Yet this precision seems important: as we have seen in Section 6, it allows us to weed out spurious undo of actions that appear in an approach that relies on a cruder transaction "embedding" mechanism. Although we have not developed a formal analysis yet, it seems this precision would be equally important, e.g., to avoid uncontrolled cascading rollbacks (domino effect) in [28] or to ensure that, in contrast to [16], rollback is always possible in failure-free computations. Although [10] introduces primitives able to track down causality information among groups of processes, called conclaves, it does not provide automatic support for undoing the effects of aborted conclaves, while our calculus directly provides a primitive to undo all the effects of a communication.

While encouraging, our results in Section 6 are only preliminary. Our concurrent rollback and minimal facilities for alternatives provide a good basis for understanding the "all-or-nothing" property of transactions. To this end it would be interesting to understand whether we are able to support both strong and weak atomicity of [23]. How to support isolation properties found, e.g., in software transactional memory models, in a way that combines well with these facilities remains to be seen. Further, we would like to study the exact relationships that exist between these facilities and the different notions of compensation that have appeared in formal models of computation for service-oriented computing, such as [6, 8]. It is also interesting to compare with zero-safe Petri nets [7], since tokens in zero places dynamically define transaction scopes as done by communications in croll-π.

From a practical point of view, we want both to refine the interpreter, and to test it against a wider range of more complex case studies. Concerning the interpreter, a main point is to allow for garbage collection of memories which cannot be restored any more, so to improve space efficiency.

References

[1] M. Abadi and T. Harris. Perspectives on transactional memory. In *CONCUR'09*, volume 5710 of *LNCS*. Springer, 2009.

[2] L. Acciai, M. Boreale, and S. Dal-Zilio. A concurrent calculus with atomic transactions. In *ESOP'07*, volume 4421 of *LNCS*. Springer, 2007.

[3] G. Bacci, V. Danos, and O. Kammar. On the statistical thermodynamics of reversible communicating processes. In *CALCO 2011*, volume 6859 of *LNCS*, 2011.

[4] W. W. Rouse Ball. *Mathematical Recreations and Essays (12th ed.)*. Macmillan, New York, 1947.

[5] G. Berry and J.-J. Lévy. Minimal and optimal computations of recursive programs. *J. ACM*, 26(1), 1979.

[6] R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL'05*. ACM, 2005.

[7] R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation*, 156(1-2), 2000.

[8] M. J. Butler, C.A.R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years CSP*, number 3525 in LNCS. Springer, 2004.

[9] L. Cardelli and C. Laneve. Reversible structures. In *CMSB 2011*. ACM, 2011.

[10] T. Chothia and D. Duggan. Abstractions for fault-tolerant global computing. *Theor. Comput. Sci.*, 322(3), 2004.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comp. Sci.*, 285(2), 2002.

[12] V. Danos and J. Krivine. Reversible communicating systems. In *CONCUR'04*, volume 3170 of *LNCS*. Springer, 2004.

[13] V. Danos and J. Krivine. Transactions in RCCS. In *CONCUR'05*, volume 3653 of *LNCS*. Springer, 2005.

[14] E. de Vries, V. Koutavas, and M. Hennessy. Communicating transactions. In *CONCUR 2010*, volume 6269 of *LNCS*. Springer, 2010.

[15] K. Donnelly and M. Fluet. Transactional events. *Journal of Functional Programming*, 18(5–6), 2008.

[16] J. Field and C.A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *POPL'05*. ACM, 2005.

[17] T. Harris, S. Marlow, S. L. Peyton Jones, and M. Herlihy. Composable memory transactions. *Commun. ACM*, 51(8), 2008.

[18] I. Lanese, M. Lienhardt, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Concurrent flexible reversibility (TR). `http://www.cs.unibo.it/~lanese/publications/fulltext/TR-crollpi.pdf.gz`, 2012.

[19] I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order pi. In *CONCUR 2011*, volume 6901 of *LNCS*. Springer, 2011.

[20] I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversing higher-order pi. In *CONCUR 2010*, volume 6269 of *LNCS*. Springer, 2010.

[21] G.B. Leeman. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1), 1986.

[22] M. Lienhardt, I. Lanese, C. A. Mezzina, and J.-B. Stefani. A reversible abstract machine and its space overhead. In *FMOODS/FORTE 2012*, volume 7273 of *LNCS*, 2012.

[23] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL'08*. ACM, 2008.

[24] I. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebr. Program.*, 73(1-2), 2007.

[25] I. Phillips and I. Ulidowski. A logic with reverse modalities for history-preserving bisimulations. In *EXPRESS 2011*, volume 64 of *EPTCS*, 2011.

[26] I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *Reversible Computation 2012*, volume 7581 of *LNCS*, 2012.

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

[28] L. Ziarek and S. Jagannathan. Lightweight checkpointing for concurrent ML. *J. Funct. Program.*, 20(2), 2010.

Name-passing calculi: from fusions to preorders and types

Daniel Hirschkoff, Jean-Marie Madiot
ENS Lyon, U. de Lyon, CNRS, INRIA, UCBL
{daniel.hirschkoff, jeanmarie.madiot}@ens-lyon.fr

Davide Sangiorgi
University of Bologna and INRIA
davide.sangiorgi@cs.unibo.it

***Abstract*—The fusion calculi are a simplification of the pi-calculus in which input and output are symmetric and restriction is the only binder. We highlight a major difference between these calculi and the pi-calculus from the point of view of types, proving some impossibility results for subtyping in fusion calculi. We propose a modification of fusion calculi in which the name equivalences produced by fusions are replaced by name preorders, and with a distinction between positive and negative occurrences of names. The resulting calculus allows us to import subtype systems, and related results, from the pi-calculus. We examine the consequences of the modification on behavioural equivalence (e.g., context-free characterisations of barbed congruence) and expressiveness (e.g., full abstraction of the embedding of the asynchronous pi-calculus).**

***Index Terms*—process calculus; fusions; types; subtyping;**

I. Introduction

The π-calculus is the paradigmatical name-passing calculus, that is, a calculus where names (a synonym for "channels") may be passed around. Key aspects for the success of the π-calculus are the minimality of its syntax and its expressiveness. Expressiveness comes at a price: often, desirable behavioural properties, or algebraic laws, fail. The reason is that, when employing π-calculus to describe a system, one normally follows a discipline that governs how names can be used. The discipline can be made explicit by means of *types*. Types bring in other benefits, notably the possibility of statically detecting many programming errors. Types are indeed a fundamental aspect of the π-calculus theory, and one of the most important differences between name-passing calculi and process calculi such as CCS in which names may not be passed.

One of the basic elements in type systems for name-passing calculi is the possibility of separating the capabilities for actions associated to a name, e.g., the capability of using a name in input or in output. The control of capabilities has behavioural consequences because it allows one to express constraints on the use of names. For a simple example, consider a process P that implements two distinct services $\mathtt{A}$ and $\mathtt{B}$, accessible using channels a and b that must be communicated to clients of the services. We assume here only two clients, that receive the channels via c_1 and c_2:

$$P \overset{\text{def}}{=} (\boldsymbol{\nu} a, b)\,\big(\overline{c_1}\langle a, b\rangle.\,\overline{c_2}\langle a, b\rangle.\,(\mathtt{A} \mid \mathtt{B})\big) \qquad (1)$$

We expect that outputs at a or b from the clients are eventually received and processed by the appropriate service. But this is not necessarily the case: a malign client can disrupt the expected protocol by simply offering an input at a or b and then throwing away the values received, or forwarding the values to the wrong service. These misbehaviours are ruled out by a capability type system imposing that the clients only obtain the output capability on the names a and b when receiving them from c_1 and c_2. The typing rules are straightforward, and mimic those for the typing of references in imperative languages with subtyping.

Capabilities [1] are at the basis of more complex type systems, with a finer control on names. For instance, type systems imposing constraints on successive usages of the names like usage-based type systems and deadlock-detection systems, session types, and so on [2], [3], [4].

Capabilities are closely related to subtyping. In the example (1), P creates names a and b, and possesses both the input and the output capabilities on them; it however transmits to the clients only a subset of the capabilities (namely the output capability alone). The subset relation on capabilities gives rise to a subtype relation on types. All forms of subtyping for π-calculus or related calculi in the literature require a discipline on capabilities. Subtyping can also be used to recover well-known forms of subtyping in other computational paradigms, e.g., functional languages or object-oriented languages, when an encoding of terms into processes is enhanced with an encoding of types [5].

An interesting family of variants of the π-calculus are — what we call here — the *fusion calculi*: Fusion [6], Update [7], Explicit Fusions [8], Chi [9], Solos [10]. Their beauty is the simplification achieved, with binding removed from the input construct. Thus input prefixing becomes symmetric to output prefixing, and restriction remains as the only binder. The effect of a synchronisation between an output $\overline{a}b.\,P$ and an input $ac.\,Q$ is to fuse the two object names b and c, which are now interchangeable. Thus communications produce, step-by-step, an equivalence relation on names. Different fusion-like calculi differ in the way the name equivalence is handled. The operational theories of these calculi have been widely studied, e.g. [6], [11], [12], [13], [14].

As for the π-calculus (sometimes abbreviated as π in the sequel), however, the expressiveness of fusion calculi makes desirable behavioural properties fail. The same examples for the π-calculus can be used. For instance, the problems of misbehaving clients of the services of (1) remain. Actually, in fusion calculi additional problems arise; for example a client receiving the two channels a and b along c_i could fuse them using an input $c_i\langle n, n\rangle.\, R$. Now a and b are indistinguishable, and an emission on one of them can reach any of the two services (and if a definition of a service is recursive, a recursive call could be redirected towards the other service).

In the paper we study the addition of types to fusion calculi; more generally, to single-binder calculi, where input is not binding (in fusion calculi, in addition, reductions fuse names). We begin by highlighting a striking difference between π-calculus and fusion calculi, proving some impossibility results for subtyping (and hence for general capability-based type systems, implicitly or explicitly involving subtyping). In the statement of the results, we assume a few basic properties of type systems for name-passing calculi, such as strengthening, weakening and type soundness, and the validity of the ordinary typing rules for the base operators of parallel composition and restriction. These results do not rule out completely the possibility of having subtyping or capabilities in fusion calculi, because of the few basic assumptions we make. They do show, however, that such type systems would have to be more complex than those for ordinary name-passing calculi such as the π-calculus, or require modifications or constraints in the syntax of the calculi.

Intuitively, the impossibility results arise because at the heart of the operational semantics for fusion calculi is an equivalence relation on names, generated through name fusions. In contrast, subtyping and capability systems are built on a preorder relation (be it subtyping, or set inclusion among subsets of capabilities). The equivalence on names forces one to have an equivalence also on types, instead of a preorder.

We propose a solution whose crux is the replacement of the equivalence on names by a preorder, and a distinction on occurrences of names, between 'positive' and 'negative'. In the resulting single-binder calculus, $\pi\mathtt{P}$ ('π with Preorder'), reductions generate a preorder. The basic reduction rule is

$$\overline{c}a.\, P \mid cb.\, Q \longrightarrow P \mid Q \mid a/b \ .$$

The particle a/b, called an *arc*, sets a to be above b in the name preorder. Such a process may redirect a prefix at b (which represents a 'positive' occurrence of b) to become a prefix at a. We show that the I/O (input/output) capability systems of the π-calculus can be reused in $\pi\mathtt{P}$, following a generalisation of the typing rules of the π-calculus that takes into account the negative and positive occurrences of names. A better understanding of type systems with subtyping in name-passing calculi is a by-product of this study. For instance, the study suggests that it is essential for subtyping that substitutions produced by communications (in $\pi\mathtt{P}$, the substitutions produced by arcs) only affect the positive occurrences of names.

The modification also brings in behavioural differences. For instance, both in the π-calculus and in $\pi\mathtt{P}$, a process that creates a new name a has the guarantee that a will remain different from all other known names, even if a is communicated to other processes (only the creator of a can break this, by using a in negative position). This is not true in fusion calculi, where the emission of a may produce fusions between a and other names. To demonstrate the proximity with the π-calculus we show that the embedding of the asynchronous π-calculus into $\pi\mathtt{P}$ is fully abstract (full abstraction of the encoding of the π-calculus into fusion calculi fails). We also exhibit an encoding of Explicit Fusions into $\pi\mathtt{P}$, where fusions become bi-directional arcs.

We present two possible semantics for $\pi\mathtt{P}$ that differ on the moment arcs enable substitutions. In the *eager* semantics, arcs may freely act on prefixes; in the *by-need* semantics, arcs act on prefixes only when interactions occur. We provide a characterisation of the reference contextual behavioural equivalence (barbed congruence) as a context-free labelled bisimilarity for the by-need semantics. We also compare and contrast the semantics, both between them and with semantics based on name fusion.

A property of certain fusion calculi (Fusion, Explicit Fusion) is a semantic duality induced by the symmetry between input and output prefixes. In $\pi\mathtt{P}$, the syntax still allows us to swap inputs and outputs, but in general the original and final processes have incomparable behaviours.

We conclude by examining the following syntactic constraint in single-binder calculi: each name, say b, may occur at most once in negative position (this corresponds to input object, as in $ab.\, P$, or to the source of an arc, as in a/b). Under this constraint, the two semantics for $\pi\mathtt{P}$, eager and by-need, coincide. In fusion calculi, the constraint allows us to import the π-calculus type systems. The constraint is however rather strong, and, in fusion calculi, breaks the semantic duality between inputs and outputs.

In summary, $\pi\mathtt{P}$, while being syntactically similar to fusion calculi, remains fairly close to the π-calculus (type systems, management of names).

Further related work: Central to $\pi\mathtt{P}$ is the preorder on names, that breaks the symmetry of name equivalence in fusion-like calculi. Another important ingredient for the theory of $\pi\mathtt{P}$ is the distinction between negative and positive occurrences of a name. In Update [7] and (asymmetric versions of) Chi [9], reductions produce ordinary substitutions on names. In practice, however, substitutions are not much different from fusions: a substitution $\{a/b\}$ fuses a with b *and* makes a the representative of the equivalence class. Still, substitutions are directed, and in this sense Update and Chi look closer to $\pi\mathtt{P}$ than the other fusion calculi. For instance Update and Chi, like $\pi\mathtt{P}$, lack the duality property on computations. Update was refined to the Fusion calculus [6] because of difficulties in the extension with polyadicity. Another major difference for Update and Chi with respect to $\pi\mathtt{P}$ is that in the former calculi substitutions replace all occurrences of names, whereas $\pi\mathtt{P}$ takes into account the distinction between positive and negative occurrences.

The question of controlling the fusion of private names has been addressed in [15], in the U-calculus. This calculus makes no distinction between input and output, and relies on two forms of binding to achieve a better control of scope extrusion, thus leading to a sensible behavioural theory that encompasses fusions and π. Thus the calculus is not single-binder. It is unclear how capability types could be defined in it, as it does not have primitive constructs for input and output.

Paper outline: Section II gives some background. In Section III, we present some impossibility results on type systems for fusion-like calculi. Section IV introduces $\pi\mathtt{P}$ and its type system. The behavioural theory of $\pi\mathtt{P}$ is explored in Section V, and we give some expressiveness results in Section VI. Section VII studies a syntactical restriction that can be applied to $\pi\mathtt{P}$ and fusions, and we discuss future work in Section VIII.

II. Background on name-passing calculi

In this section we group terminology and notation that are common to all the calculi discussed in the paper. For simplicity of presentation, all calculi in the paper are finite. The addition of operators like replication for writing infinite behaviours goes as expected. The results in the paper would not be affected.

We informally call *name-passing* the calculi in the π-calculus tradition, which have the usual constructs of parallel composition and restriction, and in which computation is interaction between input and output constructs. *Names* identify the pairs of matching inputs/outputs, and the values transmitted may themselves be names. Restriction is a binder for the names; in some cases the input may be a binder too. Examples of these calculi are the π-calculus, the asynchronous π-calculus, the Join calculus, the Distributed π-calculus, the Fusion calculus, and so on. Binders support the usual alpha-conversion mechanism, and give rise to the usual definitions of free and bound names.

Convention 1. To simplify the presentation, throughout the paper, in all statements (including rules), we assume that the bound names of the entities in the statements are *different from each other and different from the free names* (*Barendregt convention on names*). Similarly, we say that a name is *fresh* or *fresh for a process*, if the name does not appear in the entities of the statements or in the process. □

We use $a, b, \ldots$ to range over names. In a free input $ab.\,P$, bound input $a(b).\,P$, output $\overline{a}b.\,P$, we call a the *subject* of the prefix, and b the *object*. We sometimes abbreviate prefixes as $a.\,P$ and $\overline{a}.\,P$ when the object carried is not important. We omit trailing $\mathbf{0}$, for instance writing $\overline{a}b$ in place of $\overline{a}b.\,\mathbf{0}$. We write $P\{a/b\}$ for the result of applying the substitution of b with a in P.

When restriction is the only binder (hence the input construct is not binding), we say that the calculus *has a single binder*. If in addition interaction involves fusion between names, so that we have ($\Longrightarrow$ stands for an arbitrary number of reduction steps, and in the right-hand side P and Q can be omitted if they are $\mathbf{0}$)

$$(\boldsymbol{\nu}c)\,(\overline{a}b.\,P \mid ac.\,Q \mid R) \Longrightarrow (P \mid Q \mid R)\{b/c\}\ , \qquad (2)$$

we say that the calculus *has name-fusions*, or, more briefly, has *fusions*. (We are not requiring that (2) is among the rules of the operational semantics of the calculus, just that (2) holds. The shape of (2) has been chosen so to capture the existing calculi; the presence of R allows us to capture also the Solos calculus.) All single-binder calculi in the literature (Update [7], Chi [9], Fusion [6], Explicit Fusion calculus [11], Solos [10]) have fusions. In Section IV we will introduce a single-binder calculus without fusions.

In all calculi in the paper, (reduction-closed) barbed congruence will be our reference behavioural equivalence. Its definition only requires a reduction relation, $\longrightarrow$, and a notion of barb on names, $\downarrow_a$. Intuitively, a barb at a holds for a process if that process can accept an offer of interaction at a from its environment. We omit the definition, which is standard. We write $\simeq_{\mathcal{L}}$ for (strong) reduction-closed barbed congruence in a calculus $\mathcal{L}$. Informally, $\simeq_{\mathcal{L}}$ is the largest relation that is context-closed, barb-preserving, and reduction-closed. Its weak version, written $\approxeq_{\mathcal{L}}$, replaces the relation $\longrightarrow_{\mathcal{L}}$ with its reflexive and transitive closure $\Longrightarrow_{\mathcal{L}}$, and the barbs $\downarrow_a^{\mathcal{L}}$ with the weak barbs $\Downarrow_a^{\mathcal{L}}$, where $\Downarrow_a^{\mathcal{L}}$ is the composition of the relations $\Longrightarrow_{\mathcal{L}}$ and $\downarrow_a^{\mathcal{L}}$ (i.e., the barb is visible after some internal actions). See Appendix A for more details.

III. Typing and subtyping with fusions

We consider typed versions of languages with fusions. We show that in such languages it is impossible to have a non-trivial subtyping, assuming a few simple and standard typing properties of name-passing calculi.

We use T, U to range over types, and Γ to range over type environments, i.e., partial functions from names to types. We write $\mathrm{dom}(\Gamma)$ for the set of names on which Γ is defined. In name-passing calculi, a type system assigns a type to each name. Typing judgements are of the form $\Gamma \vdash P$ (process P respects the type assignments in Γ), and $\Gamma \vdash a : T$ (name a can be assigned type T in Γ).[1] The following are the standard typing rules for parallel composition and restriction:

$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \mid P_2} \qquad \frac{\Gamma, x : T \vdash P}{\Gamma \vdash (\boldsymbol{\nu}x : T)\, P} \qquad (3)$$

The first rule says that any two processes typed in the same type environment can be composed in parallel. The second rule handles name restriction.[2]

In name-passing calculi, the basic type construct is the channel (or connection) type $\sharp\, T$. This is the type of a name that may carry, in an input or an output, values of type T. Consequently, we also assume that the following rule for prefixes $ab.\,P$ and $\overline{a}b.\,P$ is *admissible*.

$$\frac{\Gamma(a) = \sharp\, T \qquad \Gamma(b) = T \qquad \Gamma \vdash P}{\Gamma \vdash \alpha.\,P}\ \alpha \in \{ab, \overline{a}b\} \qquad (4)$$

(Prefixes may not have a continuation, in which case P would be missing from the rule.) In the rule, the type of the subject

[1] We consider in this paper basic type systems and basic properties for them; more sophisticated type systems exist where processes have a type too, e.g., behavioural type systems.

[2] In resource-sensitive type systems, i.e., those for linearity [16] and receptiveness [5], where one counts certain occurrences of the names, the rule for parallel composition has to be modified. As mentioned earlier, in this paper we stick to basic type systems, ignoring resource consumption.

and of the object of the prefix are compatible. Again, these need not be the typing rules for prefixes; we are just assuming that the rules are valid in the type system. The standard rule for prefix would have, as hypotheses,

$$\Gamma \vdash a : \sharp T \qquad \Gamma \vdash b : T \ .$$

These imply, but are not equivalent to, the hypotheses in (4), for instance in presence of subtyping.

Fundamental properties of type systems are:

- Subject Reduction (or Type Soundness): if $\Gamma \vdash P$ and $P \to P'$, then $\Gamma \vdash P'$;
- Weakening: if $\Gamma \vdash P$ and a is fresh, then $\Gamma, a : T \vdash P$;
- Strengthening: whenever $\Gamma, a : T \vdash P$ and a is fresh for P, then $\Gamma \vdash P$;
- Closure under injective substitutions: if $\Gamma, a : T \vdash P$ and b is fresh, then $\Gamma, b : T \vdash P\{b/a\}$.

Definition 2. *A typed calculus with single binder is* plain *if it satisfies Subject Reduction, Weakening, Strengthening, Closure under injective substitutions, and the typing rules (3) and (4) are admissible.*

If the type system admits subtyping, then another fundamental property is narrowing, which authorises, in a typing environment, the specialisation of types:

- (Narrowing): if $\Gamma, a : T \vdash P$ and $U \leq T$ then also $\Gamma, a : U \vdash P$.

When narrowing holds, we say that the calculus *supports narrowing*.

A typed calculus *has trivial subtyping* if, whenever $T \leq U$, we have $\Gamma, a : T \vdash P$ iff $\Gamma, a : U \vdash P$. When this is not the case (i.e., there are T, U with $T \leq U$, and T, U are not interchangeable in all typing judgements) we say that the calculus has *meaningful* subtyping.

Under the assumptions of Definition 2, a calculus with fusions may only have trivial subtyping.

Theorem 3. *A typed calculus with fusions that is plain and supports narrowing has trivial subtyping.*

In the proof, given in Appendix B, we assume a meaningful subtyping and use it to derive a contradiction from type soundness and the other hypotheses.

One may wonder whether, in Theorem 3, more limited forms of narrowing, or a narrowing in the opposite direction, would permit some meaningful subtyping. Narrowing is interesting when it allows us to modify the type of the values exchanged along a name, that is, the type of the object of a prefix. (In process calculi, communication is the analogous of application for functional languages, and changing the type of an object is similar to changing the type of a function or of its argument.) In other words, disallowing narrowing on objects would make subtyping useless. We show that *any* form of narrowing, on one prefix object, would force subtyping to be trivial.

Theorem 4. *Suppose a typed calculus with fusions is plain and there is at least one prefix α with object b, different from the subject, and there are two types S and T such that $S \leq T$ and one of the following forms of narrowing holds for all Γ:*

1) *whenever $\Gamma, b : T \vdash \alpha.\,\mathbf{0}$, we also have $\Gamma, b : S \vdash \alpha.\,\mathbf{0}$;*
2) *whenever $\Gamma, b : S \vdash \alpha.\,\mathbf{0}$, we also have $\Gamma, b : T \vdash \alpha.\,\mathbf{0}$.*

Then S and T are interchangeable in all typing judgements.

As a consequence, authorising one of the above forms of narrowing for all S and T such that $S \leq T$ implies that the calculus has trivial subtyping. The proof of Theorem 4 is similar to that of Theorem 3. (Appendix B).

Remark 5. Theorems 3 and 4 both apply to all fusion calculi: Fusion, Explicit Fusions, Update, Chi, Solos (where the continuation P is $\mathbf{0}$). □

Another consequence of Theorems 3 and 4 is that it is impossible, in plain calculi with fusions, to have an I/O type system; more generally, it is impossible to have any capability-based type system that supports meaningful subtyping.

Actually, to apply the theorems, it is not even necessary for the capability type system to have an explicit notion of subtyping. For Theorem 3, it is sufficient to have sets of capabilities with a non-trivial ordering under inclusion, meaning that we can find two capability types T and U such that whenever $\Gamma, a : U \vdash P$ holds then also $\Gamma, a : T \vdash P$ holds, but not the converse (e.g., T provides more capabilities than U). We could then impose a subtype relation $\leq$ on types, as the least preorder satisfying $T \leq U$. Theorem 3 then tells us that type soundness and the other properties of Definition 2 would require also $U \leq T$ to hold, i.e., T and U are interchangeable in all typing judgements. In other words, the difference between the capabilities in T and U has no consequence on typing. Similarly, to apply Theorem 4 it is sufficient to find two capability types T and U and a single prefix in whose typing U can replace T.

IV. A calculus with name preorders

A. Preorders, positive and negative occurrences

We now refine the fusion calculi by replacing the equivalence relation on names generated through communication by a preorder, yielding $\pi\mathtt{P}$ ('π with Preorder'). As the preorder on types given by subtyping allows promotions between related types, so the preorder on names of $\pi\mathtt{P}$ allows promotions between related names. Precisely, if a is below a name b in the preorder, then a prefix at a may be promoted to a prefix at b and then interact with another prefix at b. Thus an input $av.\,P$ may interact with an output $\overline{b}w.\,Q$; and, if also c is below b, then $av.\,P$ may as well interact with an output $\overline{c}z.\,R$.

The ordering on names is introduced by means of the *arc* construct, $a/\!\!/b$, that declares the *source* b to be below the *target* a. The remaining operators are as for fusion calculi (i.e., those of the π-calculus with bound input replaced by free input).

$$P ::= \mathbf{0} \;\Big|\; P \mid P \;\Big|\; \overline{a}b.\,P \;\Big|\; ab.\,P \;\Big|\; \boldsymbol{\nu} aP \;\Big|\; a/\!\!/b \ .$$

The semantics of the calculus is given in the reduction style. Structural congruence, $\equiv$, is defined as the usual congruence produced by the monoidal rules for parallel composition and the rules for commuting and extruding restriction (see Appendix C for a complete definition). We explain the effect of reduction by means of contexts, rather than separate rules for each operator. Contexts allow us a more succinct presentation, and a simpler comparison with an alternative semantics (Section V). An *active context* is one in which the hole may reduce. Thus the only difference with respect to ordinary contexts is that the hole may not occur underneath a prefix. We use C to range over (ordinary) contexts, and E for active contexts. The rules for reduction are as follows (the subscript in $\longrightarrow_{\mathrm{ea}}$, for "eager", will distinguish this from the alternative semantics in Section V-A):

$$\text{R-SCon}: \quad \frac{P \equiv E[Q] \qquad Q \longrightarrow_{\mathrm{ea}} Q' \qquad E[Q'] \equiv P'}{P \longrightarrow_{\mathrm{ea}} P'}$$

$$\text{R-Inter}: \quad \overline{a}b.\, P \mid ac.\, Q \longrightarrow_{\mathrm{ea}} P \mid Q \mid b/c$$

$$\text{R-SubOut}: \quad a/b \mid \overline{b}c.\, Q \longrightarrow_{\mathrm{ea}} a/b \mid \overline{a}c.\, Q$$

$$\text{R-SubInp}: \quad a/b \mid bc.\, Q \longrightarrow_{\mathrm{ea}} a/b \mid ac.\, Q$$

Rule R-Inter shows that communication generates an arc. Rules R-SubOut and R-SubInp show that arcs only act on the subject of prefixes; moreover, they only act on *unguarded* prefixes (i.e., prefixes that are not underneath another prefix). The rules also show that arcs are persistent processes. Acting only on prefix subjects, arcs can be thought of as particles that "redirect prefixes": an arc a/b redirects a prefix at b towards a higher name a. (Arcs remind us of special π-calculus processes, called forwarders or wires [17], which under certain hypotheses allow one to model substitutions; as for arcs, so the effect of forwarders is to replace the subject of prefixes.)

We write $\Longrightarrow_{\mathrm{ea}}$ for the reflexive and transitive closure of $\longrightarrow_{\mathrm{ea}}$. Here are some examples of reduction.

$$\begin{array}{rlrl}
 & & \overline{a}c.\overline{c}a.e.P \mid & ad.de.\overline{a}.Q \\
\text{R-Inter} & \longrightarrow_{\mathrm{ea}} & \overline{c}a.e.P \mid & de.\overline{a}.Q \mid c/d \\
\text{R-SubInp} & \longrightarrow_{\mathrm{ea}} & \overline{c}a.e.P \mid & ce.\overline{a}.Q \mid c/d \\
\text{R-Inter} & \longrightarrow_{\mathrm{ea}} & e.P \mid & \overline{a}.Q \mid c/d \mid a/e \\
\text{R-SubInp} & \longrightarrow_{\mathrm{ea}} & a.P \mid & \overline{a}.Q \mid c/d \mid a/e \\
\text{R-Inter} & \longrightarrow_{\mathrm{ea}} & P \mid & Q \mid c/d \mid a/e
\end{array}$$

Reductions can produce multiple arcs that act on the same name. This may be used to represent certain forms of choice, as in the following processes:

$$\begin{array}{l}
(\boldsymbol{\nu} h, k)\, (bu.\, cu.\, \overline{u} \mid \overline{b}h.\, h.\, P \mid \overline{c}k.\, k.\, Q) \\
\Longrightarrow_{\mathrm{ea}} \ (\boldsymbol{\nu} h, k)\, (\overline{u} \mid h/u \mid k/u \mid h.\, P \mid k.\, Q) \ .
\end{array}$$

Both arcs may act on $\overline{u}$, and are therefore in competition with each other. The outcome of the competition determines which process between P and Q is activated. For instance, reduction may continue as follows:

$$\begin{array}{rll}
\text{R-SubOut} & \longrightarrow_{\mathrm{ea}} & (\boldsymbol{\nu} h, k)\, (\overline{k} \mid h/u \mid k/u \mid h.\, P \mid k.\, Q) \\
\text{R-Inter} & \longrightarrow_{\mathrm{ea}} & (\boldsymbol{\nu} h, k)\, (h/u \mid k/u \mid h.\, P \mid Q) \ .
\end{array}$$

Definition 6 (Positive and negative occurrences)**.** *In an input $ab.\, P$ and an arc a/b, the name b has a* negative occurrence*. All other occurrences of names in input, output and arcs are* positive occurrences*.*

An occurrence in a restriction ($\boldsymbol{\nu}$a) is neither negative nor positive, intuitively because restriction acts only as a binder, and does not stand for an usage of the name (in particular, it does not take part in a substitution).

Negative occurrences are particularly important, as by properly tuning them, different usages of names may be obtained. For instance, a name with zero negative occurrence is a constant (i.e., it is a channel, and may not be substituted); and a name that has a single negative occurrence is like a π-calculus name bound by an input (see Section VI-B).

The number of negative occurrences of a name is invariant under reduction.

Lemma 7. *If $P \longrightarrow_{\mathrm{ea}} P'$ then for each b, the number of negative occurrences of b in P and P' is the same.*

B. Types

We now show that the I/O capability type system and its subtyping can be transplanted from π to $\pi\mathsf{P}$. In all typed calculi in the paper, binding occurrences of names are annotated with their type — we are not concerned with type inference.

In the typing rules for I/O-types in the (monadic) π-calculus [1], two additional types are introduced: $\mathtt{o}\, T$, the type of a name that can be used only in output and that carries values of type T; and $\mathtt{i}\, T$, the type of a name that can be used only in input and that carries values of type T. The subtyping rules stipulate that $\mathtt{i}$ is covariant, $\mathtt{o}$ is contravariant, and $\sharp$ is invariant. Subtyping is brought up into the typing rules through the subsumption rule. The most important typing rules are those for input and output prefixes; for input we have:

$$\text{T-InpBound}: \quad \frac{\Gamma \vdash a : \mathtt{i}\, T \qquad \Gamma, b : T \vdash P}{\Gamma \vdash a(b : T).\, P}$$

The π-calculus supports narrowing, and this is essential in the proof of subject reduction.

The type system for $\pi\mathsf{P}$ is presented in Table I. With respect to the π-calculus, only the rule for input needs an adjustment, as $\pi\mathsf{P}$ uses free, rather than bound, input. The idea in rule T-InpFree of $\pi\mathsf{P}$ is however the same as in rule T-InpBound of π: we look up the type of the object of the prefix, say T, and we require $\mathtt{i}\, T$ as the type for the subject of the prefix. To understand the typing of an arc a/b, recall that such an arc allows one to replace b with a. Rule T-Arc essentially checks that a has at least as many capabilities as b, in line with the intuition for subtyping in capability type systems.

Common to all premises of T-InpBound, T-InpFree and T-Arc is the look-up of the type of names that occur negatively (the source of an arc and the object of an input prefix): the type that appears for b in the hypothesis is precisely the type found in the conclusion (within the process or in Γ). In contrast, the types for positive occurrences may be different (e.g., because of subsumption $\Gamma \vdash a : \mathtt{i}\, T$ may hold even if $\Gamma(a) \neq \mathtt{i}\, T$). We cannot type inputs like outputs: consider

$$\text{T-InpFree2-Wrong}: \quad \frac{\Gamma \vdash a : \mathtt{i}\, T \qquad \Gamma \vdash b : T}{\Gamma \vdash ab}$$

Rule T-InpFree2-Wrong would accept, for instance, an input ab in an environment Γ where $a : \mathtt{i}\, \mathtt{i}\, \mathbf{1}$ and $b : \sharp\, \mathbf{1}$. By

Types ($\mathbf{1}$ is the unit type): $T ::= \mathtt{i}\,T \mid \mathtt{o}\,T \mid \sharp\,T \mid \mathbf{1}$

Subtyping rules:

$$\frac{}{\sharp\,T \le \mathtt{i}\,T} \qquad \frac{}{\sharp\,T \le \mathtt{o}\,T} \qquad \frac{S \le T}{\mathtt{i}\,S \le \mathtt{i}\,T} \qquad \frac{S \le T}{\mathtt{o}\,T \le \mathtt{o}\,S} \qquad \frac{}{T \le T} \qquad \frac{S \le T \qquad T \le U}{S \le U}$$

Typing rules:

$$\textsc{Tv-Name}\ \frac{}{\Gamma, a : T \vdash a : T} \qquad \textsc{Subsumption}\ \frac{\Gamma \vdash a : S \qquad S \le T}{\Gamma \vdash a : T} \qquad \text{T-}\textsc{Res}\ \frac{\Gamma, a : T \vdash P}{\Gamma \vdash \boldsymbol{\nu} a P} \qquad \text{T-}\textsc{Par}\ \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \text{T-}\textsc{Nil}\ \frac{}{\Gamma \vdash \mathbf{0}}$$

$$\text{T-}\textsc{Out}\ \frac{\Gamma \vdash a : \mathtt{o}\,T \qquad \Gamma \vdash b : T \qquad \Gamma \vdash P}{\Gamma \vdash \overline{a}b.\, P} \qquad \text{T-}\textsc{InpFree}\ \frac{\Gamma \vdash a : \mathtt{i}\,\Gamma(b) \qquad \Gamma \vdash P}{\Gamma \vdash ab.\, P} \qquad \text{T-}\textsc{Arc}\ \frac{\Gamma \vdash a : \Gamma(b)}{\Gamma \vdash a/b}$$

TABLE I
THE TYPE SYSTEM OF πP

subtyping and subsumption, we could then derive $\Gamma \vdash b : \mathtt{i}\,\mathbf{1}$. In contrast, rule T-INPFREE, following the input rule of the π-calculus, makes sure that the object of the input does not have too many capabilities with respect to what is expected in the type of the subject of the input. This constraint is necessary for subject reduction. As a counterexample, assuming rule T-INPFREE2-WRONG, we would have $a : \sharp\,\mathtt{i}\,\mathbf{1}, b : \sharp\,\mathbf{1}, c : \mathtt{i}\,\mathbf{1} \vdash P$, for $P \stackrel{\text{def}}{=} ab \mid \overline{a}c \mid \overline{b}$. However, $P \longrightarrow_{\text{ea}} c/b \mid \overline{b} \longrightarrow_{\text{ea}} c/b \mid \overline{c}$, and the final derivative is not typable under Γ (as Γ only authorises inputs at c).

In πP, the direction of the narrowing is determined by the negative or positive occurrences of a name.

Theorem 8 (Polarised narrowing)**.** *Let T_1 and T_2 be two types such that $T_1 \le T_2$.*

1) *If a occurs only positively in P, then $\Gamma, a : T_2 \vdash P$ implies $\Gamma, a : T_1 \vdash P$.*
2) *If a occurs only negatively in P, then $\Gamma, a : T_1 \vdash P$ implies $\Gamma, a : T_2 \vdash P$.*
3) *If a occurs both positively and negatively in P, then it is in general unsound to replace, in a typing $\Gamma \vdash P$, the type of a in Γ with a subtype or supertype.*

Theorem 8 (specialised to prefixes) does not contradict Theorem 4, because in πP, reduction does not satisfy (2) (from Section II). Our system enjoys subject reduction:

Theorem 9. *If $\Gamma \vdash P$ and $P \longrightarrow_{\text{ea}} P'$ then also $\Gamma \vdash P'$.*

Remark 10. Theorem 8 may be seen as a refinement of the standard narrowing result for name-passing calculi. In the π-calculus, for instance, a free name only has positive occurrences. Hence the usual narrowing corresponds to Theorem 8(1). And in an input $a(b : T).\, P$, the binder for b represents a negative occurrence, so that if b is free in P then b has both positive and negative occurrences, which means that the type T may not be modified, as by Theorem 8(3). In contrast, Theorem 8(2) is vacuous in π, as a name b with only negative occurrences is found in an input $a(b : T).\, P$ where b is not free in P.

In general, in a name-passing calculus, if a name has only positive occurrences, then its type (be it declared in the typing environment, or in the binding occurrence of that name within the process) may be replaced by a subtype, and conversely for names with only negative occurrences, whereas the type of names with both positive and negative occurrences may not be changed. Defining rules that distinguish between negative and positive occurrences in name-passing calculi is beyond the scope of this paper. A rule of thumb however seems that if the occurrence of a name generates a substitution acting on that name (i.e., a replacement of the name), then the occurrence is negative; if it does not, then it is positive. Thus in a fusion $a = b$ of the Explicit Fusion calculus, the occurrences of a and b are both positive and negative, as a fusion may produce a substitution a/b or a substitution b/a (which, incidentally, gives another explanation of the impossibility of narrowing in presence of an explicit fusion construct). □

Remark 11. For the Subject Reduction theorem for πP it is critical that an arc a/b only acts on positive occurrences of b. Provided this is respected, the theorem remains valid under different behaviours for arcs (e.g., simultaneously replacing all positive occurrences of b, not only at top-level). □

V. BEHAVIOURS

A. An alternative semantics

The operational semantics given to πP in Section IV allows arcs to act locally, at any time. The effect of an arc is irreversible: the application of an arc a/b to a prefix at b commits that prefix to interact along a name that is greater than, or equal to, a in the preorder among names. A commitment may disable certain interactions, even block a prefix for ever. Consider, e.g.,

$$(\boldsymbol{\nu} a, c)\ (bv.\, P \mid \overline{c}w.\, Q \mid a/b \mid c/b) \qquad (5)$$

There is a competition between the two arcs; if the first wins, the process is deadlocked:

$$\longrightarrow_{\mathrm{ea}} (\boldsymbol{\nu} a, c)\,(av.\,P \mid \overline{c}w.\,Q \mid a/b \mid c/b)$$

since a and c are unrelated in the preorder.

We consider here an alternative semantics, in which the action of arcs is not a commitment: arcs come about only when interaction occurs. For this reason we call the new semantics *by-need* (arcs act only when 'needed'), whereas we call *eager* the previous semantics (arcs act regardless of matching prefixes). In this semantics, as in the π-calculus, an interaction involves both a synchronisation and a substitution; however unlike in the π-calculus where the substitution is propagated to the whole term, here substitution only replaces the subject of the interacting prefixes.

The formalisation of the new semantics makes use of the partial order on names induced by arcs. In a process, an arc is *active* if it is unguarded, i.e., it is not underneath a prefix. We write $\mathtt{preor}(P)$ for the preorder on names produced by the active arcs in P (i.e., the least preorder $\leq$ that includes $b \leq a$ for each active arc a/b in P). Similarly, $\mathtt{preor}(C)$ is the preorder produced by the active arcs of the context C. Note that this definition relies on the Barendregt convention on names (Convention 1), as it is purely syntactic, i.e., if P and P' are alpha convertible then $\mathtt{preor}(P)$ and $\mathtt{preor}(P')$ may be different. A definition that does not rely on the convention is given in Appendix D.

We write $P \triangleright a \curlyvee b$ if $\{a, b\}$ has an upper bound in the preorder $\mathtt{preor}(P)$, that is, there is a name that is above both a and b; in this case we also say that a and b are *joinable*. Similarly we write $C \triangleright a \curlyvee b$ for contexts. For instance, we have $\boldsymbol{\nu} u(u/a \mid u/b \mid Q) \triangleright a \curlyvee b$, and $\boldsymbol{\nu} v(\overline{v}t \mid (\boldsymbol{\nu} w)(w/v \mid a/w \mid [\cdot]) \triangleright a \curlyvee v$. We have $P \triangleright a \curlyvee b$ iff $P' \triangleright a \curlyvee b$ if P and P' are alpha convertible and a and b occur free in P.

Example 12. *A process $M_{fg} = (\boldsymbol{\nu} c)(c/f \mid c/g)$ acts like a* mediator*: it joins names f and g (we have $M_{fg} \triangleright f \curlyvee g$). Mediators remind us of equators in the π-calculus, or of fusions in the Explicit Fusion calculus, but lack the transitivity property (e.g., $M_{fg} \mid M_{gh} \triangleright f \curlyvee h$ does not hold).*

Definition 13 (By-need reduction)**.** *The* by-need *reduction relation, $P \longrightarrow_{\mathrm{bn}} P'$, is defined by the following rules, where $\equiv$ is as in the eager semantics:*

$$\text{BN-SCON}:\quad \frac{P \equiv E[Q] \qquad Q \longrightarrow_{\mathrm{bn}} Q' \qquad E[Q'] \equiv P'}{P \longrightarrow_{\mathrm{bn}} P'}$$

$$\text{BN-RED}:\quad \frac{E \triangleright a \curlyvee b}{E[ac.\,P \mid \overline{b}d.\,Q] \longrightarrow_{\mathrm{bn}} E[P \mid d/c \mid Q]}$$

Relation $\Longrightarrow_{\mathrm{bn}}$ is the reflexive transitive closure of $\longrightarrow_{\mathrm{bn}}$.

While the eager semantics has simpler rules, the by-need semantics avoids 'too early commitments' on prefixes. For instance, the only immediate reduction of the process in (5) is

$$\longrightarrow_{\mathrm{bn}} (\boldsymbol{\nu} a, c)\,(P \mid w/v \mid Q \mid a/b \mid c/b)$$

where prefixes $bv.\,P$ and $\overline{c}w.\,Q$ interact because their subjects are joinable in the preorder generated by the two arcs.

Lemma 14 (Eager and by-need)**.** *$P \longrightarrow_{\mathrm{bn}} P'$ (by-need semantics) implies $P \Longrightarrow_{\mathrm{ea}} P'$ (eager semantics).*

Corollary 15. *Theorem 9 holds for the by-need semantics.*

B. Behavioural equivalence

We contrast barbed congruence in $\pi\mathtt{P}$ under the two semantics we have given, eager and by-need. We have already defined reduction relations, we only need to define barbs. This requires some care, as the interaction of a process with its environment may be mediated by arcs. For this, and to have a uniform definition of barbs under the eager and by-need semantics, we follow the definition of success in testing equivalence [18], using a special signal ω that we assume may not appear in processes: thus for any name a, the barb $\downarrow_a$ holds for a process P if there is a prefix α with subject a such that $P \mid \alpha.\,\omega$ reduces in one step to a process in which ω is unguarded (i.e., the offer of the environment of an action at a may be accepted by P). Weak barbs and barbed congruence are then defined in the standard way, as outlined in Section II. We write $\simeq_{\mathrm{ea}}$ and $\approxeq_{\mathrm{ea}}$ (resp. $\simeq_{\mathrm{bn}}$ and $\approxeq_{\mathrm{bn}}$) for the strong and weak versions of eager (resp. by-need) barbed congruence.

The eager and by-need semantics of $\pi\mathtt{P}$ yield incomparable equivalences. The two following laws are valid in the by-need case, and fail in the eager case:

$$(\boldsymbol{\nu} a)a/c = \mathbf{0} \qquad\qquad a \mid a = a.\,a\ \ .$$

To see the failure of the first law in the eager semantics, consider a context $C \stackrel{\text{def}}{=} [\cdot] \mid (\boldsymbol{\nu} b)(b/c) \mid c \mid \overline{c}.\,\overline{w}$; then $C[(\boldsymbol{\nu} a)(a/c)]$ can lose the possibility of emitting at w, by reducing in two steps to $(\boldsymbol{\nu} a)(a/c \mid a) \mid (\boldsymbol{\nu} b)(b/c \mid \overline{b}.\,\overline{w})$, because of a commitment determined by arcs; this cannot happen for $C[\mathbf{0}]$. There are no early commitments in the by-need semantics, for which the two processes are hence equal.

Similarly, in the eager semantics, it is possible to put $a \mid a$ in a context where two arcs rewrite each a prefix differently, while one can only rewrite the topmost prefix in $a.\,a$. This scenario cannot be played in the by-need semantics.

On the other hand, the following law is valid for strong (and weak) eager equivalence, but fails to hold in the by-need case:

$$(\boldsymbol{\nu} abu)(a/u \mid b/u \mid \overline{u} \mid a.\,\overline{w}) = (\boldsymbol{\nu} v)(\overline{v} \mid v.\,\tau.\,\overline{w} \mid v.\,\mathbf{0})\ \ .$$

($\tau.\,\overline{w}$ stands for $\boldsymbol{\nu} c(c \mid \overline{c}.\,\overline{w})$). The intuition is that concurrent substitutions are used on the left-hand side to implement internal choice. As a consequence of the law $(\boldsymbol{\nu} a)a/c = \mathbf{0}$, in the by-need case, process b/u can be disregarded on the left, so that the process on the left *must* do the output on w.

We have introduced $\pi\mathtt{P}$ with the eager semantics for reasons of simplicity, but we find the by-need semantics more compelling. Below, unless otherwise stated, we work under by-need, though we also indicate what we know under eager.

C. Context-free characterisations of barbed congruence

When it comes to proving behavioural equalities, the definition of barbed congruence is troublesome, as it involves a heavy quantification on contexts. One therefore looks for context-free coinductive characterisations, as labelled bisimilarities that take into account not only reductions within a process, but also the potential interactions between the process and its environment (e.g., input and output actions). We present such characterisation for the by-need equivalence; currently we do not have one for the eager.

As actions for the by-need labelled bisimilarity, we use, besides τ-actions, only free input and free output:

$$\mu \quad ::= \quad \tau \mid ab \mid \overline{a}b \ .$$

In by-need, labelled transitions are written $P \xrightarrow{\mu}_{\mathrm{bn}} P'$. Internal transitions have already been defined, in the reduction semantics, thus we can take relation $\xrightarrow{\tau}_{\mathrm{bn}}$ to coincide with the reduction relation $\longrightarrow_{\mathrm{bn}}$. Input and output transitions are defined by these rules:

$$\text{BN-I}{\small\text{NP}} : \quad \frac{E \triangleright a \curlyvee b \qquad E \text{ does not bind } b \text{ and } d}{E[ac.\, P] \xrightarrow{bd}_{\mathrm{bn}} E[d/c \mid P]}$$

$$\text{BN-O}{\small\text{UT}} : \quad \frac{E \triangleright a \curlyvee b \qquad E \text{ does not bind } b \text{ and } d}{E[\overline{a}c.\, P] \xrightarrow{\overline{b}d}_{\mathrm{bn}} E[c/d \mid P]}$$

The purpose of the two rules is to define the input and output transitions, with labels as simple as possible, with which to derive a labelled bisimilarity. The two rules are not supposed to be composed together to derive τ-actions (which are computed from the rules of reduction). We leave the definition of a pure SOS semantics, which avoids the structural manipulations of structural congruence, for future work.

To understand rules BN-INP and BN-OUT, suppose the environment is offering an action at b. Since a and b are joinable, there is a name, say e, that is above both a and b in the preorder; hence the prefix at a in the process and the prefix at b in the environment can be transformed into prefixes at e, and can interact. The need for the preorder explains why we found it convenient to express actions via active contexts. In the action, the use of a free object d allows us to ignore name extrusion and thus simplifies the bisimulation checks. As an example of BN-OUT, we have (similar observations can be made for BN-INP):

$$\begin{array}{l}(\boldsymbol{\nu} u)\big(\, u/b \mid (\boldsymbol{\nu} a, c)(u/a \mid \overline{a}c.\, P)\,\big) \\ \quad \xrightarrow{\overline{b}d}_{\mathrm{bn}} \ (\boldsymbol{\nu} u)\big(\, u/b \mid (\boldsymbol{\nu} a, c)(u/a \mid c/d \mid P)\,\big) \ .\end{array}$$

Here the process can interact with the environment at b (and hence perform a transition where b is the subject), because a and b are joinable. Name c is not extruded; instead the arc c/d redirects interactions on d to c.

The labelled bisimulation requires, besides the invariance for actions, invariance under the addition of arcs; moreover a check is made on the visible effects of arcs. In the clause for actions, no extrusion or binding on names is involved; further, it is sufficient that the objects of the actions are *fresh names*.

Definition 16 (Bisimulation)**.** *A* by-need *bisimulation $\mathcal{R}$ is a set of pairs (P, Q) s.t. $P\mathcal{R}Q$ implies:*

1) *$P \mid a/b \;\mathcal{R}\; Q \mid a/b$, for each name a, b (invariance under arcs);*
2) *if a and b appear free in P, then $P \triangleright a \curlyvee b$ implies $Q \triangleright a \curlyvee b$;*
3) *if $P \xrightarrow{\mu}_{\mathrm{bn}} P'$, then $Q \xrightarrow{\mu}_{\mathrm{bn}} Q'$ and $P'\mathcal{R}Q'$ (where the object part of μ is fresh);*
4) *the converse of clauses (2) and (3).*

Bisimilarity, written $\sim_{\mathrm{bn}}$, is the largest bisimulation.

We now present some examples and laws that are proved using the coinductive proof method of labelled bisimilarity. All equalities and inequalities also hold under the eager semantics, though for some equalities only in the weak case (e.g., Lemma 19).

Any input and output of πP can be transformed into a bound prefix, by introducing a new restricted name:

Lemma 17. *We have $ax.\, P \sim_{\mathrm{bn}} (\boldsymbol{\nu} x')ax'.\, (x'/x \mid P)$ and $\overline{b}y.\, Q \sim_{\mathrm{bn}} (\boldsymbol{\nu} y')\overline{b}y'.\, (y/y' \mid Q)$, for fresh x' and y'.*

If these laws are applied to all inputs and outputs of a process P, then the result is a process P' that is behaviourally the same as P, and in which all names exchanged in an interaction are fresh. Thus P' reminds us of a variant of π that achieves symmetry between input and output constructs, namely πI, the π-calculus with internal mobility [19].

Lemma 18. *We have $(\boldsymbol{\nu} b, c)\overline{a}c.\, \overline{a}b.\, \mathbf{0} \not\sim_{\mathrm{bn}} (\boldsymbol{\nu} c)\overline{a}c.\, \overline{a}c.\, \mathbf{0}$, and $(\boldsymbol{\nu} b, c)ac.\, ab.\, \mathbf{0} \sim_{\mathrm{bn}} (\boldsymbol{\nu} c)ac.\, ac.\, \mathbf{0}$.*

These laws show a difference between input and output in behavioural equalities. The reason for the inequality is that the first process can produce two transitions with objects e, f yielding $P \overset{\text{def}}{=} \boldsymbol{\nu} c\, (c/f \mid c/e)$, and then $P \triangleright e \curlyvee f$.

Lemma 19 (Substitution and polarities)**.**

1) *If name a has only positive occurrences in P, then $(\boldsymbol{\nu} a)(P \mid b/a) \sim_{\mathrm{bn}} P\{b/a\}$;*
2) *if name a has only negative occurrences in P, then $(\boldsymbol{\nu} a)(P \mid a/b) \sim_{\mathrm{bn}} P\{b/a\}$;*
3) *$(\boldsymbol{\nu} a)(P \mid b/a \mid a/b) \sim_{\mathrm{bn}} P\{b/a\}$.*

For the comparison between labelled bisimilarity and barbed congruence, the most delicate part is the proof of congruence for bisimilarity. This is due to the shape of visible transitions, where an arc is introduced and the object part is always a fresh name, and to the use of $\equiv$ in the definition of transitions. The proof can be found in Appendices H and I.

Theorem 20. *Bisimilarity is a congruence.*

Theorem 21 (Characterisation of barbed congruence)**.** *In πP, relations $\sim_{\mathrm{bn}}$ and $\simeq_{\mathrm{bn}}$ coincide.*

Hence all the laws stated above for $\sim_{\mathrm{bn}}$ hold for $\simeq_{\mathrm{bn}}$.

VI. EXPRESSIVENESS OF πP

We compare πP with a few other calculi, both as examples of the use of the calculus and as a test for its expressiveness.

When useful, we work in a *polyadic* version of $\pi\mathtt{P}$; the addition of polyadicity goes as for other name-passing calculi in the literature. All results in this section use the by-need semantics; we do not know their status under the eager semantics.

A. Explicit Fusions

Bi-directional arcs, e.g., $a/b \mid b/a$, work as name fusions (cf, Lemma 19(3)). We thus can encode calculi based on name fusion into $\pi\mathtt{P}$. As an example, we consider the Explicit Fusion calculus [8]. Its syntax extends the Fusion calculus with a fusion construct $a = b$. The encoding is defined as follows for prefixes and explicit fusions, the other constructs being encoded homomorphically:

$$\begin{array}{rcl} \llbracket \overline{a}\langle v\rangle .\, P \rrbracket & = & (\boldsymbol{\nu} w)\overline{a}\langle v, w\rangle .\, wv .\, \llbracket P \rrbracket \\ \llbracket ax .\, Q \rrbracket & = & (\boldsymbol{\nu} y)a\langle x, y\rangle .\, \overline{y}\langle x\rangle .\, \llbracket Q \rrbracket \\ \llbracket a = b \rrbracket & = & a/b \mid b/a \end{array}$$

In Explicit Fusions, an interaction introduces a name fusion. In the $\pi\mathtt{P}$ encoding, this is mimicked in two steps so to be able to produce bidirectional arcs. The second step is the reverse of the original interaction, and is realised by means of an extra private name. We have operational correspondence for the encoding (we do not know whether it is is fully abstract).

Theorem 22. *Let P, Q be processes of the Explicit Fusion calculus, and $\longrightarrow_{\mathrm{EF}}$ the reduction relation in the calculus.*

1) *If $P \equiv Q$ then $\llbracket P \rrbracket \simeq_{\mathrm{bn}} \llbracket Q \rrbracket$;*
2) *if $P \longrightarrow_{\mathrm{EF}} P'$ then $\llbracket P \rrbracket \longrightarrow_{\mathrm{bn}} \approxeq_{\mathrm{bn}} \llbracket P' \rrbracket$;*
3) *conversely, if $\llbracket P \rrbracket \longrightarrow_{\mathrm{bn}} Q$, then $Q \approxeq_{\mathrm{bn}} \llbracket P' \rrbracket$ for some P' such that $P \longrightarrow_{\mathrm{EF}} P'$.*

A similar result holds for the Fusion calculus, though for Explicit Fusions the statement is simpler because in the latter calculus a restriction is not necessary for fusions to act.

B. π-calculus

The embedding of the π-calculus into a fusion calculus is defined by translating the bound input construct as follows:

$$\llbracket a(x).\, P \rrbracket = (\boldsymbol{\nu} x)\, ax.\, \llbracket P \rrbracket$$

(the other constructs being translated homomorphically). The same encoding can be used for $\pi\mathtt{P}$.

The encoding of π-calculus into Fusions is not fully abstract for barbed congruence. For instance, in the π-calculus, a new channel is guaranteed to remain different from all other existing channels. Thus in a process $\boldsymbol{\nu} a\, (\overline{b}a.\, (a.\, P \mid \overline{c}.\, Q))$, the two prefixes $a.\, P$ and $\overline{c}.\, Q$ may never interact with each other, in any context, even if a is exported. This property does not hold in the Fusion calculus, as a recipient of the newly created name a could equate it with any other name (e.g., using the context $bc.\, \mathbf{0} \mid [\cdot]$).

We do not know whether the encoding of the full π-calculus into $\pi\mathtt{P}$ is fully abstract. However, at least the encoding is fully abstract on the asynchronous subset (where no continuation is allowed after the output prefix).

Theorem 23. *Suppose P, Q are processes from the asynchronous π-calculus, $\mathrm{A}\pi$. Then $P \simeq_{\mathrm{A}\pi} Q$ iff $\llbracket P \rrbracket \simeq_{\mathrm{bn}} \llbracket Q \rrbracket$.*

In the theorem, $\simeq_{\mathrm{A}\pi}$ could be replaced by $\simeq_{\pi}$ (barbed congruence in the full π-calculus). Note that $\simeq_{\mathrm{A}\pi}$ is the standard barbed congruence, as opposed to *asynchronous* barbed congruence, where output barbs are visible but input barbs are not. We believe the theorem also holds under asynchronous barbed congruence.

For the proof of the theorem, we first establish results of operational correspondence between source and target terms of the encoding. Then the direction from right to left is easy because contexts of the π-calculus are also contexts of $\pi\mathtt{P}$ (under the encoding). The delicate direction is the opposite. Here we use Theorem 21, and the characterisation of π-calculus barbed congruence on the subset of asynchronous processes as ground bisimilarity [5]. We also make use of some up-to techniques, notably 'by-need bisimulation up to $\sim_{\mathrm{bn}}$ and restriction' whose soundness is proved along the lines of soundness proofs of similar techniques for other forms of bisimilarity. We finally consider the relation defined as $\{(\llbracket P \rrbracket \mid \sigma, \llbracket Q \rrbracket \mid \sigma) \;\big|\; P \sim_{\mathrm{g}} Q\}$, where σ is a parallel composition of arcs, and prove that it is a by-need bisimulation up to $\sim_{\mathrm{bn}}$ and up to restriction.

Regarding translations in the opposite direction, both for fusion calculi and for $\pi\mathtt{P}$, the encoding into π is not possible in general. However, for $\pi\mathtt{P}$ some results can be obtained under constraints such as *asynchrony* and *locality*. Something similar has been done by Merro [20] for the Fusion calculus.

VII. Unique negative occurrences of names

In this section we consider a constrained version of the calculi discussed in the paper, where each name may have at most one negative occurrence in a process. In the fusion calculus [6] the constraint means that each name appears at most once as the object of an input. In $\pi\mathtt{P}$, the constraint affects also arcs (as their source is a negative occurrence).

The constraint is rather draconian, bringing the calculi closer to the π-calculus (where the constraint is enforced by having binding input). Still, the constraint is more generous than tying the input to a binder as in π. For instance, we have more complex forms of causality involving input, as in $\boldsymbol{\nu} x(ax.\, \overline{w}t \mid \overline{b}x)$, where the input at a blocks the output at w, and can be triggered before or after the output at b takes place. We call $\pi\mathtt{P1}$ and Fu1 the constrained versions of $\pi\mathtt{P}$ and Fusions; in both languages the constraint is preserved by reduction.

We show that the constraint makes certain differences between calculi or semantics disappear. In $\pi\mathtt{P1}$ the eager and the by-need semantics of $\pi\mathtt{P}$ coincide, at least in a weak semantics.

Theorem 24. *In $\pi\mathtt{P1}$, relations $\approxeq_{\pi\mathtt{P1}\mathrm{ea}}$ and $\approxeq_{\pi\mathtt{P1}\mathrm{bn}}$ coincide.*

The following property is useful in the proof (see Appendix E).

Lemma 25. *For $P \in \pi\mathtt{P1}$, suppose $P \longrightarrow_{\mathrm{ea}} P'$ where the reduction is a rewrite step involving an arc. Then $P \approxeq_{\pi\mathtt{P1}\mathrm{ea}} P'$.*

The calculi $\pi\mathtt{P1}$ and Fu1 resulting from the constraint are behaviourally similar. For instance, in $\pi\mathtt{P1}$ the directionality of arcs is irrelevant, as shown by the following law (where we omit the subscripts 'ea' and 'bn' in the light of Theorem 24).

Lemma 26. $a/b \approxeq_{\pi\mathrm{P}1} b/a$.

Another difference that disappears under the constraint of unique negative occurrences of names is the one concerning capabilities and subtyping in fusion calculi with respect to π and $\pi\mathrm{P}$, exposed in Sections III and IV. Indeed, to equip $\mathrm{F}\textsc{u}1$ with an I/O type system and subtyping, we can use exactly the rules of $\pi\mathrm{P}$ in Section IV-B — with the exception of T-ARC as $\mathrm{F}\textsc{u}1$ does not have arcs. This intuitively because $\mathrm{F}\textsc{u}1$ is, syntactically, a subset of $\pi\mathrm{P}$ (each process of $\mathrm{F}\textsc{u}1$ is also a process of $\pi\mathrm{P}$), and the Subject Reduction theorem for $\pi\mathrm{P}$ in Section IV-B holds regardless of when and how arcs generate substitutions (Remark 11); making an arc a/b act immediately and on all positive occurrences of b is similar to substitution as in $\mathrm{F}\textsc{u}1$. This may however involve changing the type of a name c into a smaller type when c is used in input object; e.g., in $ac \mid (\boldsymbol{\nu} b : T)\overline{a}b.\, P \longrightarrow_{\mathrm{Fu1}} P\{c/b\}$ (where $\longrightarrow_{\mathrm{Fu1}}$ is reduction in $\mathrm{F}\textsc{u}1$), name c is used at type T, which is a smaller type than $\Gamma(c)$.

Theorem 27. *Let P be a* $\mathrm{F}\textsc{u}1$ *process. If $\Gamma \vdash P$ and $P \longrightarrow_{\mathrm{Fu1}} P'$, then $\Gamma' \vdash P'$, where for at most one name c, $\Gamma'(c) \leq \Gamma(c)$; for other names b, $\Gamma'(b) = \Gamma(b)$.*

Note that $\mathrm{F}\textsc{u}1$ does not satisfy the conditions of Definition 2 because well-typed processes may not be freely put in parallel, as this could break the constraint on unique input objects.

We leave for future work a thorough comparison between $\pi\mathrm{P}1$, $\mathrm{F}\textsc{u}1$, and π-calculus.

VIII. Future work

Here we mention some lines for future work, in addition to those already mentioned in the main text.

The coinductive characterisation of behavioural equivalence in $\pi\mathrm{P}$ has been presented in the strong case, and should be extended to the weak case. We have presented and compared two semantics for $\pi\mathrm{P}$, eager and by-need. While we tend to consider the advantages so far uncovered for the by-need superior, more work is needed to draw more definite conclusions. For instance, it would also be interesting to contrast axiomatisations of the semantics, rules for pure SOS presentations of the operational semantics, the expressiveness of the subcalculus in which the two semantics agree, and implementations. We do not expect, in contrast, significant differences to arise from type systems.

Another possible advantage of by-need is a smoother extension with dynamic operators like guarded choice, in which an action may discard a component. (In the eager case it is unclear what should be the effect of an arc that acts on one of the summands of a choice.) Choice would be useful for axiomatisations. In by-need, we would have for instance

$$(\boldsymbol{\nu} b, c)\overline{a}b.\, \overline{a}c.\, (\overline{b}|c) \sim (\boldsymbol{\nu} b, c)\overline{a}b.\, \overline{a}c.\, (\overline{b}.\, c + c.\, \overline{b}).$$

The law, valid in both $\pi\mathrm{P}$ and π, illustrates the possibility of generating fresh names that cannot be identified with other names even if exported. The law fails in fusion calculi as a recipient might decide to equate b and c (cf. Section VI-B).

Solos calculus is the polyadic Fusion calculus without continuations. Solos can encode continuations [10]. We believe the same machinery would work for the 'Solos version' of $\pi\mathrm{P}$.

It could also be interesting to study the representation of $\pi\mathrm{P}$ into Psi calculi [21]. This may not be immediate because the latter make use of on an equivalence relation on channels, while the former uses a preorder. One could then see whether the move from Fusions and π to $\pi\mathrm{P}$ in this paper, and the corresponding results on types, can be lifted at the level of Psi calculi, by comparing them with variants based on preorders. [24] presents type systems for Psi calculi, and for explicit fusions, but does not address subtyping.

Acknowledgement

The authors acknowledge support from the ANR projects 2010-BLAN-0305 PiCoq and 12IS02001 PACE.

References

[1] B. Pierce and D. Sangiorgi, "Typing and subtyping for mobile processes," *Math. Str. in Comp. Sci.*, vol. 6, no. 5, pp. 409–453, 1996.

[2] N. Kobayashi, "Type systems for concurrent programs," in *10th Anniversary Colloquium of UNU/IIST*, ser. LNCS, vol. 2757. Springer, 2003, pp. 439–453.

[3] ——, "A new type system for deadlock-free processes," in *CONCUR*, ser. LNCS, vol. 4137. Springer, 2006, pp. 233–247.

[4] K. Honda, V. T. Vasconcelos, , and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in *ESOP*, ser. LNCS, vol. 1381. Springer, 1998, pp. 122—-138.

[5] D. Sangiorgi and D. Walker, *The Pi-Calculus: a theory of mobile processes*. Cambridge University Press, 2001.

[6] J. Parrow and B. Victor, "The fusion calculus: expressiveness and symmetry in mobile processes," in *LICS*. IEEE, 1998, pp. 176 –185.

[7] ——, "The update calculus (extended abstract)," in *AMAST*, ser. LNCS, vol. 1349. Springer, 1997, pp. 409–423.

[8] L. Wischik and P. Gardner, "Explicit fusions," *Theor. Comput. Sci.*, vol. 340, no. 3, pp. 606–630, 2005.

[9] Y. Fu, "The χ-calculus," in *APDC*. IEEE Comp. Soc., 1997, pp. 74–81.

[10] C. Laneve and B. Victor, "Solos in concert," *Math. Str. in Comp. Sci.*, vol. 13, no. 5, pp. 657–683, 2003.

[11] P. Gardner and L. Wischik, "Explicit fusions," in *MFCS*, ser. LNCS, vol. 1893. Springer, 2000, pp. 373–382.

[12] J. Parrow and B. Victor, "The tau-laws of fusion," in *CONCUR*, ser. LNCS, vol. 1466. Springer, 1998, pp. 99–114.

[13] G. L. Ferrari, U. Montanari, E. Tuosto, B. Victor, and K. Yemane, "Modelling Fusion Calculus using HD-Automata," in *CALCO*, ser. LNCS, vol. 3629. Springer, 2005, pp. 142–156.

[14] F. Bonchi, M. G. Buscemi, V. Ciancia, and F. Gadducci, "A presheaf environment for the explicit fusion calculus," *J. Autom. Reasoning*, vol. 49, no. 2, pp. 161–183, 2012.

[15] M. Boreale, M. G. Buscemi, and U. Montanari, "A general name binding mechanism," in *TGC*, ser. LNCS, vol. 3705. Springer, 2005, pp. 61–74.

[16] N. Kobayashi, B. Pierce, and D. Turner, "Linearity and the pi-calculus," *TOPLAS*, vol. 21, no. 5, pp. 914–947, 1999.

[17] K. Honda and N. Yoshida, "On reduction-based process semantics," *Theor. Comp. Sci.*, vol. 152, no. 2, pp. 437–486, 1995.

[18] R. De Nicola and M. Hennessy, "Testing equivalences for processes," *Theor. Comput. Sci.*, vol. 34, pp. 83–133, 1984.

[19] D. Sangiorgi, "Pi-calculus, internal mobility, and agent-passing calculi," *Theor. Comput. Sci.*, vol. 167, no. 1&2, pp. 235–274, 1996.

[20] M. Merro, "Locality in the pi-calculus and applications to distributed objects," Ph.D. dissertation, École des Mines, France, 2000.

[21] J. Bengtson, M. Johansson, J. Parrow, and B. Victor, "Psi-calculi: Mobile processes, nominal data, and logic," in *LICS*. IEEE, 2009, pp. 39—48.

[22] B. Victor, "The fusion calculus : Expressiveness and symmetry in mobile processes," Ph.D. thesis, Uppsala University, 1998.

[23] Web appendix to this paper, available from http://hal.inria.fr/hal-00818068, 2013.

[24] H. Hüttel, "Typed ψ-calculi," in *CONCUR*, ser. LNCS, vol. 6901. Springer, 2011, pp. 265–279.

Appendix

A. Reduction-closed barbed congruence (Section II)

Definition 28 (Reduction-closed barbed congruence)**.** *Let $\mathcal{L}$ be a process calculus, in which a reduction relation $\longrightarrow_{\mathcal{L}}$ and barb predicates $\downarrow_a^{\mathcal{L}}$, for each a in a given set of names, have been defined.*

A relation $\mathcal{R}$ on the processes of $\mathcal{L}$ is context-closed *if $P \mathcal{R} Q$ implies $C[P]\mathcal{R}C[Q]$, for each context C of $\mathcal{L}$; the relation is* barb-preserving *if for any name a, $P \downarrow_a^{\mathcal{L}}$ implies $Q \downarrow_a^{\mathcal{L}}$; it is* reduction-closed *if whenever $P \longrightarrow_{\mathcal{L}} P'$, there is Q' s.t. $Q \longrightarrow_{\mathcal{L}} Q'$ and $P'\mathcal{R}Q'$.*

Then reduction-closed barbed congruence in *$\mathcal{L}$, written $\simeq_{\mathcal{L}}$, is the largest symmetric relation on the processes of $\mathcal{L}$ that is context-closed, reduction-closed, and barb-preserving.*

B. Proofs of impossibility results (Section III)

Statement of Theorem 3: *A typed calculus with fusions that is plain and supports narrowing has trivial subtyping.*

Proof Sketch: We define the following active context:

$$E \triangleq (\boldsymbol{\nu} cb)(\overline{u}b \mid uc \mid \overline{v}a \mid vc \mid [\cdot]) \ \ .$$

Note that in E we only use b as an output object. The intention is that, given some process P, and u, v, c some fresh names, $E[P]$ should reduce to $P\{a/b\}$. Indeed, by applying hypothesis (2) twice, we have

$$\begin{aligned} E[P] \quad &= \quad (\boldsymbol{\nu} bc)(\overline{u}b \mid uc \mid \overline{v}a \mid vc \mid P) && (6)\\ &\Longrightarrow \quad (\boldsymbol{\nu} b)(\overline{v}a \mid vb \mid P\{b/c\}) && (7)\\ &= \quad (\boldsymbol{\nu} b)(\overline{v}a \mid vb \mid P) && (8)\\ &\Longrightarrow \quad P\{a/b\} \ \ . && (9) \end{aligned}$$

Suppose $U \leq T$, we show $\Gamma, a : T \vdash P$ iff $\Gamma, a : U \vdash P$. The implication from left to right is narrowing. To prove the right to left implication, suppose $\Gamma, a : U \vdash P$, and prove $\Gamma, a : T \vdash P$. By injective name substitution we have $\Gamma, b : U \vdash P\{b/a\}$ for some fresh b.

In the typing environment $\Gamma, b{:}U, u{:}\sharp\, T, v{:}\sharp\, T, c{:}T, a{:}T$ the process $\overline{u}b$ is well-typed thanks to narrowing and weakening, hence so is $(\overline{u}b \mid uc \mid \overline{v}a \mid vc \mid P\{b/a\})$. By the restriction rule we get $\Gamma, a{:}T, u{:}\sharp\, T, v{:}\sharp\, T \ \vdash \ E[P\{b/a\}]$, the latter reducing to $P\{b/a\}\{a/b\}$ by (9). Since b has been taken fresh, $P\{b/a\}\{a/b\} = P$. Hence, by Subject Reduction, $\Gamma, a{:}T, u{:}\sharp\, T, v{:}\sharp\, T \ \vdash \ P$. We finally deduce $\Gamma, a : T \vdash P$ by Strengthening. ■

Statement of Theorem 4: *Suppose a typed calculus with fusions is plain and there is at least one prefix α with object b, different from the subject, and there are two types S and T such that $S \leq T$ and one of the following forms of narrowing holds for all Γ:*

1) *whenever $\Gamma, b : T \vdash \alpha.\, \mathbf{0}$, we also have $\Gamma, b : S \vdash \alpha.\, \mathbf{0}$;*
2) *whenever $\Gamma, b : S \vdash \alpha.\, \mathbf{0}$, we also have $\Gamma, b : T \vdash \alpha.\, \mathbf{0}$.*

Then S and T are interchangeable in all typing judgements.

Proof Sketch: For all Δ we prove that $\Delta, x : T \vdash P$ iff $\Delta, x : S \vdash P$. Let x_1, x_2, a_1 and a_2 be fresh names.

$$\Delta_i \overset{\text{def}}{=} \Delta, \ x_i{:}T, \ x_{3-i}{:}S$$

We will prove that $\Delta_i \vdash P\{x_1/x\}$ implies $\Delta_i \vdash P\{x_2/x\}$ for all $i \in \{1, 2\}$. From there it is enough to conclude using weakening, strengthening and injective substitutions. We use $D = \overline{a_1}x_1 \mid \overline{a_2}x_2 \mid a_1 y \mid a_2 y$ to simulate a substitution:

$$(\boldsymbol{\nu} x_1 y)(D \mid P\{x_1/x\}) \Rightarrow P\{x_2/x\}$$

We have to prove that $\Delta' = \Delta_i, a_1{:}T_{a_1}, a_2{:}T_{a_2}, y{:}T_y \vdash D$ for some types T_{a_1} T_{a_2}, T_y. We note a the subject of α. Using the plainness of the subtyping, we can suppose that a is any of a_1 or a_2 and that b is any of x_1, x_2 or y, so to apply the hypothesis on different cases. There are eight subcases, along the cases from the hypothesis, i, and the form of α.

- (1), $i = 1$, $\alpha = \overline{a_2}x_2$: $T_{a_1} = T_{a_2} = \sharp\, T$, $T_y = T$;
- (1), $i = 1$, $\alpha = a_1 y$: $T_{a_1} = \sharp\, T$, $T_{a_2} = \sharp\, S$, $T_y = S$;
- (2), $i = 1$, $\alpha = \overline{a_1}x_1$: $T_{a_1} = T_{a_2} = \sharp\, S$, $T_y = S$;
- (2), $i = 1$, $\alpha = a_2 y$: $T_{a_1} = \sharp\, T$, $T_{a_2} = \sharp\, S$, $T_y = T$;
- (1), $i = 2$, $\alpha = \overline{a_2}x_2$: $T_{a_1} = T_{a_2} = \sharp\, T$, $T_y = T$;
- (1), $i = 2$, $\alpha = a_2 y$: $T_{a_1} = \sharp\, S$, $T_{a_2} = \sharp\, T$, $T_y = S$;
- (2), $i = 2$, $\alpha = \overline{a_1}x_1$: $T_{a_1} = T_{a_2} = \sharp\, S$, $T_y = S$;
- (2), $i = 2$, $\alpha = a_1 y$: $T_{a_1} = \sharp\, S$, $T_{a_2} = \sharp\, T$, $T_y = T$.

In all these cases we prove that $\Delta' \vdash D$ using plainness and the hypothesis on α. Plainness also give us $\Delta' \vdash P\{x_1/x\}$. We use rules from (3) and Subject Reduction to get that $\Delta' \vdash P\{x_2/x\}$ from which strengthening is enough to conclude. ■

C. Structural congruence in πP (Section IV-A)

Definition 29 (Structural congruence)**.** *Structural congruence on πP, written $\equiv$, is the smallest congruence containing the associativity and commutativity of $\mid$ and the following rules:*

$$P \mid \mathbf{0} \equiv P \qquad \boldsymbol{\nu} a \mathbf{0} \equiv \mathbf{0} \qquad \boldsymbol{\nu} a \boldsymbol{\nu} b P \equiv \boldsymbol{\nu} b \boldsymbol{\nu} a P$$

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

D. Alternative definition of $\curlyvee$ (Section V-A)

Given an active context E, the set of *captured names* of E, $\mathrm{cn}(E)$, is defined as follows: $c \in \mathrm{cn}(E)$ iff the hole occurs in the scope of a restriction on c in E ($\mathrm{cn}(E)$ is included in the set of names that are bound in E, but might be distinct from it).

Definition 30 (Reachability / Joinability of names)**.** *We introduce $\varphi ::= a \leqslant b \mid a \curlyvee b$ in which $a \leqslant b$ is read "b is reachable from a", and $a \curlyvee b$ is read "a and b are joinable". In both cases, we have $\mathrm{n}(\varphi) = \{a, b\}$. We first define a judgement $\varphi_1, \varphi_2 \vdash \varphi$, as follows:*

$$\frac{}{a \leqslant b, b \leqslant c \vdash a \leqslant c} \qquad \frac{}{a \leqslant c, b \leqslant c \vdash a \curlyvee b}$$

$$\frac{}{a \curlyvee b, c \leqslant a \vdash c \curlyvee b} \qquad \frac{}{a \curlyvee b, c \leqslant b \vdash a \curlyvee c} \qquad \frac{\varphi_1, \varphi_2 \vdash \varphi}{\varphi_2, \varphi_1 \vdash \varphi}$$

We exploit this judgement to define how $a \leqslant b$ and $a \curlyvee b$ can be derived according to a process, or to an active context (we use $A ::= P \mid E$):

$$\textsc{Refl}\ \frac{}{A \triangleright a \leqslant a} \qquad \textsc{Deduct}\ \frac{A \triangleright \varphi_1 \qquad A \triangleright \varphi_2 \qquad \varphi_1, \varphi_2 \vdash \varphi}{A \triangleright \varphi}\ .$$

Then we define $\triangleright$ for processes:

$$\frac{}{b/a \triangleright a \leqslant b} \qquad \frac{P \triangleright \varphi}{P \mid R \triangleright \varphi} \qquad \frac{P \triangleright \varphi}{R \mid P \triangleright \varphi}$$

$$\frac{P \triangleright \varphi \qquad a \notin \mathrm{n}(\varphi)}{(\boldsymbol{\nu} a)P \triangleright \varphi}$$

and for contexts (symmetrically for $E \mid P$):

$$\frac{P \triangleright \varphi \qquad \mathrm{n}(\varphi) \cap \mathrm{cn}(E) = \emptyset}{P \mid E \triangleright \varphi} \qquad \frac{E \triangleright \varphi}{P \mid E \triangleright \varphi} \qquad \frac{E \triangleright \varphi}{(\boldsymbol{\nu} a)E \triangleright \varphi}$$

Lemma 31. *If P is a $\pi\mathtt{P}$ process, the relation $\leqslant_P$ defined by $\{(a,b) \mid P \triangleright a \leqslant b\}$ is a preorder.*

Proof: Thanks to the rule Refl, $\leqslant_P$ is reflexive and thanks to the rule Deduct and the fact that $a \leqslant b, b \leqslant c \vdash a \leqslant c$, $\leqslant_P$ is transitive, hence it is a preorder. ■

E. Coincidence of eager and by-need equivalences in $\pi\mathtt{P1}$ (Section VII)

Statement of Theorem 24: $\approxeq_{\pi\mathtt{P1bn}} = \approxeq_{\pi\mathtt{P1ea}}$.

Proof Sketch: The result follows from reflexivity of a relation we define below, between processes in the eager semantics and processes in the by-need semantics.

Lemma 32. *For $P \in \pi\mathtt{P1}$, we write $Eq(P)$ for the relation between names defined by $Eq(P)(a,b)$ iff $P \triangleright a \curlyvee b$.*
Then $Eq(P)$ is an equivalence relation.

Let $\mathcal{R}$ be the relation such that $P\ \mathcal{R}\ Q$ iff

$$P, Q \in \pi\mathtt{P1}\ \wedge\ Eq(P) = Eq(Q) = \varphi\ \wedge\ P =_\varphi Q$$

where $P =_\varphi Q$ iff P is obtained from Q by replacing some subjects in active prefixes with names related by $Eq(P)$.

We prove that $P\ \mathcal{R}\ Q$ entails the following:

1) if $C[P], C[Q] \in \pi\mathtt{P1}$ then $C[P]\ \mathcal{R}\ C[Q]$,
2) $P \Downarrow_a^{\mathrm{ea}}$ iff $Q \Downarrow_a^{\mathrm{bn}}$,
3) if $P \Longrightarrow_{\mathrm{ea}} P'$ then $Q \Longrightarrow_{\mathrm{bn}} Q'$ with $P'\ \mathcal{R}\ Q'$,
4) if $Q \Longrightarrow_{\mathrm{bn}} Q'$ then $P \Longrightarrow_{\mathrm{ea}} P'$ with $P'\ \mathcal{R}\ Q'$.

We call the union of relations satisfying these properties the *eager/by-need weak reduction-closed barbed congruence* for $\pi\mathtt{P1}$, written ${}^{\mathrm{ea}}_{1}{\approxeq}^{\mathrm{bn}}_{1}$.

1) $\mathcal{R}$ is clearly context-closed in $\pi\mathtt{P1}$.
2) $P \downarrow_a^{\mathrm{bn}}$ implies $P \Downarrow_a^{\mathrm{ea}}$ as each arc involved in the joinability condition generates a $\longrightarrow_{\mathrm{ea}}$ reduction, and $P \downarrow_a^{\mathrm{ea}}$ implies $P \downarrow_a^{\mathrm{bn}}$, as $P \longrightarrow_{\mathrm{ea}} P'$ implies $P \longrightarrow_{\mathrm{bn}} P'$.
3) By induction we suppose $P \longrightarrow_{\mathrm{ea}} P'$. If this is a renaming then $P =_\varphi P'$. If this is a communication then the corresponding subjects are equated by φ in Q, which means they are joinable i.e. the by need reduction is possible.
4) Again we suppose $Q \longrightarrow_{\mathrm{bn}} Q'$, with a communication on a and b with $a \curlyvee b$. The corresponding names a', b' in P are such that $a' \curlyvee a \curlyvee b \curlyvee b'$ i.e. $a' \curlyvee b'$ so a' and b' can be rewritten into a common name, letting the communication happen.

Since $\mathcal{R} \subseteq {}^{\mathrm{ea}}_{1}{\approxeq}^{\mathrm{bn}}_{1}$, for all $P \in \pi\mathtt{P1}$ we have $P\ {}^{\mathrm{ea}}_{1}{\approxeq}^{\mathrm{bn}}_{1}\ P$ which implies that $P \approxeq_{\pi\mathtt{P1bn}} Q$ iff $P \approxeq_{\pi\mathtt{P1ea}} Q$. ■

F. The Fusion calculus

Definition 33. *The syntax of the polyadic Fusion calculus [6] without matching and choice is the following. Structural congruence is defined as usual (Definition 29).*

$$P ::= \mathbf{0}\ \Big|\ P \mid P\ \Big|\ \overline{a}\widetilde{x}.\, P\ \Big|\ a\widetilde{x}.\, P\ \Big|\ \boldsymbol{\nu} a P\ .$$

We follow the reduction semantics of the Fusion calculus, from [22]. The side conditions for (10) are that $\widetilde{x}$ and $\widetilde{y}$ are of the same arity, that $\mathrm{dom}(\sigma) = \widetilde{z}$ and that $\sigma(x_i) = \sigma(y_i)$. Note that (2), from Section II, holds.

$$\frac{P \equiv P_1 \qquad P_1 \rightarrow_F Q_1 \qquad Q_1 \equiv Q}{P \rightarrow_F Q} \qquad \frac{P \rightarrow_F Q}{E[P] \rightarrow_F E[Q]}$$

$$(\boldsymbol{\nu}\widetilde{z})(R \mid a\widetilde{x}.\, P \mid \overline{a}\widetilde{y}.\, Q) \rightarrow_F (R \mid P \mid Q)\sigma \qquad (10)$$

G. Auxiliary results

a) Results involving name preorders:

Lemma 34. *If $P \triangleright a \curlyvee b$ and $\{a,b\} \subseteq \mathrm{fn}(P)$, then $P \equiv P'$ implies $P' \triangleright a \curlyvee b$.*

Proof: The predicate $P \triangleright \varphi$ only depends on the occurrences of arcs in P; those occurrences are trivially preserved by structural congruence, except that to keep track of alpha-conversion one must consider that P's binders also bind φ's names. Hence the statement only holds for free names. ■

Lemma 35. *If $P \simeq_{\mathrm{bn}} Q$ and $P \triangleright a \curlyvee b$. Then $Q \triangleright a \curlyvee b$.*

Proof: We characterise joinability using the context $E = (- \mid \overline{a}.\, f \mid b.\, g)$ where f and g are fresh: we easily prove that $R \triangleright a \curlyvee b$ iff $E[R] \longrightarrow_{\mathrm{bn}} R_1$ where $R_1 \downarrow_f^{\mathrm{bn}}$ and $R_1 \downarrow_g^{\mathrm{bn}}$. By definition of $\simeq_{\mathrm{bn}}$ we know that $E[P] \simeq_{\mathrm{bn}} E[Q]$ and we conclude playing the bisimulation game of $\simeq_{\mathrm{bn}}$. ■

Lemma 36. *If $P \mathcal{R} Q$ and $\mathcal{R}$ preserves $\curlyvee$ and parallel composition of arcs (in particular if $\mathcal{R}$ is a $\sim_{\mathrm{bn}}$-relation), then $P \triangleright a \leqslant b$ iff $Q \triangleright a \leqslant b$.*

Proof: Let P and Q be processes and f be a fresh name. Then $P \triangleright a \leqslant b$ iff $(P \mid f/b) \triangleright a \curlyvee f$ and similarly for Q. Thanks to the second hypothesis on $\mathcal{R}$ we have $(P \mid f/b)\, \mathcal{R}\, (Q \mid f/b)$ and we conclude with the second one. ■

b) Basic tools: Prefixes delimit the action of structural congruence.

Lemma 37. *Suppose π_1, π_2 are prefixes.*

1) *If $E[\pi_1.\, P_1] \equiv P'$ then there exist E' and P_1' such that $P_1 \equiv P_1'$, $P' = E'[\pi_1.\, P_1']$ and $E \triangleright a \curlyvee b$ iff $E' \triangleright a \curlyvee b$.*

Moreover for all Q_1 such that all names of $\mathrm{fn}(Q_1)$ are either in $\mathrm{fn}(P_1)$ or not captured by E then the latter are not captured by E' and $E[Q_1] \equiv E'[Q_1]$.

2) *If $G[\pi_1.\,P_1][\pi_2.\,P_2] \equiv P'$ then there exist G', P_1' and P_2' such that $P_1 \equiv P_1'$, $P_2 \equiv P_2'$ and $P' = G'[\pi_1.\,P_1][\pi_2.\,P_2]$ or $P' = G'[\pi_2.\,P_2'][\pi_1.\,P_1']$ and $G \triangleright a \curlyvee b$ iff $G' \triangleright a \curlyvee b$.*

Proof: Structural congruence can act under prefixes only using the fact that $\equiv$ is a congruence, i.e. using the rule "if $P \equiv P'$ then $C[P] \equiv C[P']$" for some arbitrary context C containing a prefix. For this rule we work an induction on C to get the same cutting as $E[\pi_1.\,P_1]$; all the other rules deriving $\equiv$ are handled by the corresponding case analysis on the context E. Note that the statement also holds when E is an arbitrary context. ■

Lemma 38. *If $P \equiv Q$ then $P \sim_{\mathrm{bn}} Q$.*

Proof: We show that $\equiv$ is a $\sim_{\mathrm{bn}}$-bisimulation. (The proof is not by induction over the derivation of $P \equiv Q$ because the fact that $\equiv$ is a congruence is not easy to handle.) The clauses 1), 2), 4) are easy – respectively handled by the fact that $\equiv$ is a congruence, Lemma 34 and the fact that $\equiv$ is symmetric – as is the clause 3) when $\mu = \tau$ – since $\xrightarrow{\tau}_{\mathrm{bn}} = \longrightarrow_{\mathrm{bn}}$ is stable by $\equiv$. For the remaining labels we examine the case where $\mu = bd$, the other case being similar. We know that $P = E[ac.\,P_1]$ with $E \triangleright a \curlyvee b$ and $P' = E[d/c \mid P_1]$. We use Lemma 37 to get $Q = E'[ac.\,P_1]$ which implies $Q \xrightarrow{bd} E'[d/c \mid P_1] \equiv P'$. ■

c) Proof techniques:

Definition 39 (By-need bisimulation up to $\sim_{\mathrm{bn}}$ and restriction)**.** *A relation $\mathcal{R}$ is a* by-need bisimulation up to $\sim_{\mathrm{bn}}$ and restriction *if $P \mathcal{R} Q$ implies:*

1) *$P \mid a/b \;\mathcal{R}\; Q \mid a/b$, for all names a, b;*
2) *if a and b appear free in P, then $P \triangleright a \curlyvee b$ implies $Q \triangleright a \curlyvee b$;*
3) *if $P \xrightarrow{\mu}_{\mathrm{bn}} P'$ (where the object part of μ is fresh, whenever $\mu \neq \tau$), then $Q \xrightarrow{\mu}_{\mathrm{bn}} Q'$ and there are $P'', Q'', \widetilde{x}$ s.t. $P' \sim_{\mathrm{bn}} \boldsymbol{\nu}\widetilde{x}\; P''$, $Q' \sim_{\mathrm{bn}} \boldsymbol{\nu}\widetilde{x}\; Q''$, and $P'' \mathcal{R} Q''$,*
4) *the converse of clauses (2) and (3).*

Lemma 40. *If $\mathcal{R}$ is a by-need bisimulation up to $\sim_{\mathrm{bn}}$ and restriction then $\mathcal{R} \subseteq \sim_{\mathrm{bn}}$.*

H. Soundness of $\sim_{\mathrm{bn}}$ (Section V-C)

We now move to the proof that $\sim_{\mathrm{bn}}$ is a congruence. What is missing is closure by parallel composition, which is rather delicate. This is because we defined the semantics of τ-actions with a reduction semantics. (The standard schema is to define a pure SOS semantics, show that it coincides with the reduction semantics, and then work with the SOS.)

For the proof of congruence we introduce *communication contexts*. These are, intuitively, the composition of two active contexts, one used for an input, the other for an output; such input and output may produce a τ-action. Communication contexts, ranged over by G, have two holes, each occurring exactly once.

$$G ::= P \mid G \;\Big|\; G \mid P \;\Big|\; \boldsymbol{\nu} a\, G \;\Big|\; E_1 \mid E_2 \;\; .$$

By convention the leftmost hole is the first one, the other is the second one. We write $P = G[ac.\,Q][\overline{b}d.\,R]$ if P is obtained from G with $ac.\,Q$, and the second hole with $\overline{b}d.\,R$.

Communication contexts can be used to decompose a $\xrightarrow{\tau}_{\mathrm{bn}}$ transition:

Lemma 41. *Suppose $P \xrightarrow{\tau}_{\mathrm{bn}} P'$ (that is, $P \longrightarrow_{\mathrm{bn}} P'$). Then one of the following statements holds:*

- *$P = G[\overline{a}b.\,Q][cd.\,R]$ and $P' {\sim_{\mathrm{bn}}} \boldsymbol{\nu} f\,(G[b/f \mid Q][f/d \mid R])$,*
- *$P = G[cd.\,R][\overline{a}b.\,Q]$ and $P' {\sim_{\mathrm{bn}}} \boldsymbol{\nu} f\,(G[f/d \mid R][b/f \mid Q])$,*

where $P \triangleright a \curlyvee c$ and f is fresh.

Proof: The two cases are similar, the main difficulty is to keep track of the structural congruence operations. If $P \longrightarrow_{\mathrm{bn}} P'$ it means that, $P \equiv E[\overline{a}b.\,Q_1 \mid ac.\,R_1]$ and $P' \equiv E[b/c \mid Q_1 \mid R_1]$. From the first relation we can get G such that $P = G[\overline{a}b.\,Q][cd.\,R]$ (with $G \triangleright a \curlyvee c$, $Q \equiv Q_1$ and $R \equiv R_1$), ignoring the symmetric case for which the output is the left argument of G. We extract the potential restrictions $\boldsymbol{\nu}\hat{b}$ and $\boldsymbol{\nu}\hat{d}$ ($\hat{b} = \emptyset$ if b is not bound and $\hat{b} = \{b\}$ if is captured by G) from G, yielding the much alike context G' (and $G \equiv (\boldsymbol{\nu}\hat{b}\hat{d})G'$). The interesting part is that we can write the reduction with the arc at the top, then use Lemma 44 and then structural congruence to put back b and d inside G.

$$\begin{aligned} P &\equiv (\boldsymbol{\nu}\hat{b}\hat{d})G'[\overline{a}b.\,Q][cd.\,R] \\ &\longrightarrow_{\mathrm{bn}} (\boldsymbol{\nu}\hat{b}\hat{d})(b/d \mid G'[Q][R]) \\ &\sim_{\mathrm{bn}} (\boldsymbol{\nu}\hat{b}\hat{d})((\boldsymbol{\nu} f)(b/f \mid f/d) \mid G'[Q][R]) \\ &\equiv (\boldsymbol{\nu} f)(\boldsymbol{\nu}\hat{b}\hat{d})(G'[b/f \mid Q][f/d \mid R]) \\ &\equiv (\boldsymbol{\nu} f)G[b/f \mid Q][f/d \mid R] \;\; . \end{aligned}$$

To conclude we need to relate this last process to P' which is done by proving that $E[b/d \mid Q_1 \mid R_1] \equiv (\boldsymbol{\nu}\hat{b}\hat{d})(b/d \mid G'[Q][R])$, which is done by keeping tracks of the derivation of $E[\overline{a}b.\,Q_1 \mid cd.\,R_1] \equiv P$. ■

Lemma 42. *Suppose $Q \xrightarrow{bf}_{\mathrm{bn}} Q'$, that b is not captured by E and f is fresh. Then $Q \mid E[\overline{b}d.\,R_1] \xrightarrow{\tau}_{\mathrm{bn}} \sim_{\mathrm{bn}} \boldsymbol{\nu} f\;(Q' \mid E[d/f \mid R_1])$.*

Lemma 43 (Congruence for restriction)**.** *If $P \sim_{\mathrm{bn}} Q$ then for all c, $\boldsymbol{\nu} cP \;\sim_{\mathrm{bn}}\; \boldsymbol{\nu} cQ$.*

Proof: Given a relation $\mathcal{R}$, we define

$$\begin{aligned} (\mathcal{R})^{\mathrm{Sub}} \;\;=\;\; &\{(P \mid \sigma,\, Q \mid \sigma).\; P\mathcal{R}Q \text{ and } \sigma \text{ is} \\ &\text{a parallel composition of arcs}\} \;\; . \end{aligned}$$

We show that $(\{(\boldsymbol{\nu} cP,\, \boldsymbol{\nu} cQ),\, P \sim_{\mathrm{bn}} Q\})^{\mathrm{Sub}}$ is a bisimulation up to $\equiv$. This is a consequence of the following observations:

- For any u, v, c, P such that $\{u, v\} \subseteq \mathrm{fn}(P)$ and $c \notin \{u, v\}$, we have $P \triangleright u \curlyvee v$ iff $\boldsymbol{\nu} cP \triangleright u \curlyvee v$.
- The visible transitions of our labelled transition system do not involve name extrusion, and we have that $P \xrightarrow{\alpha}_{\mathrm{bn}} P'$ iff $\boldsymbol{\nu} cP \xrightarrow{\alpha}_{\mathrm{bn}} \boldsymbol{\nu} cP'$ for $c \notin \mathrm{n}(\alpha)$.

- Suppose now $\boldsymbol{\nu}cP \xrightarrow{\tau}_{\mathrm{bn}} P'$. This means $P \xrightarrow{\tau}_{\mathrm{bn}} P_0$ for some P_0 s.t. $P' \equiv \boldsymbol{\nu}cP_0$. But then $Q \xrightarrow{\tau}_{\mathrm{bn}} Q_0$, $P_0 \sim_{\mathrm{bn}} Q_0$ and $\boldsymbol{\nu}cQ \xrightarrow{\tau}_{\mathrm{bn}} \boldsymbol{\nu}cQ_0$.

■

Lemma 44 (Transitivity of arcs)**.** *For all active context E we have: $E[a/c] \sim_{\mathrm{bn}} E[\boldsymbol{\nu}b(a/b \mid b/c)]$.*

Proof: Let $\mathcal{R}$ be the corresponding relation. We show that $\mathcal{R}$ is a $\sim_{\mathrm{bn}}$-bisimulation up to $\equiv$. Of course the relation is stable by parallel composition of arcs, since E can be an arbitrary active context. Concerning the $\curlyvee$ condition, the left-to-right implication is rather clear. From right to left, we must prove that we cannot get more from $\boldsymbol{\nu}b(a/b \mid b/c)$ than from a/c which is achieved by the restriction $\boldsymbol{\nu}b$. Now concerning the transitions we know from the $\curlyvee$ condition that the same names will be joinable through the preorder, independently of $\equiv$ or the context. The resulting processes will still stay in $\mathcal{R}$, up to $\equiv$. ■

Lemma 45 (Congruence for parallel composition)**.** *If $P \sim_{\mathrm{bn}} Q$ then also $P \mid R \sim_{\mathrm{bn}} Q \mid R$.*

Proof (Sketch): **Special case:** we first suppose that all arcs in R occur under at least one prefix. We show that

$$\{(P \mid R, Q \mid R),\ P \sim_{\mathrm{bn}} Q \text{ and } R \text{ does not contain active arcs}\}$$

is a bisimulation up to restriction and up to bisimilarity.

Suppose then $P \mid R \xrightarrow{\tau}_{\mathrm{bn}} U$, in which both P and R contribute (the other possibilities are easier).

Suppose P makes the input (the case of output is symmetric). In this case we have, using Lemma 41:

$$P = E[ac.\, P_1] \qquad R = F[\bar{b}d.\, R_1]$$

where $E \triangleright a \curlyvee b$ (since R has no arc), f is fresh and with $P' = E[f/c \mid P_1]$ and $R' = F[d/f \mid R_1]$:

$$U \sim_{\mathrm{bn}} \boldsymbol{\nu}f\,(P' \mid R') \enspace .$$

Using rule EN-INP, we also have $P \xrightarrow{bf}_{\mathrm{bn}} P'$. Hence, since $P \sim_{\mathrm{bn}} Q$, $Q \xrightarrow{bf}_{\mathrm{bn}} Q'$ and $P' \sim_{\mathrm{bn}} Q'$ for some Q', which gives $Q' = E'[a'c'.\, Q_1]$ for some a' s.t. $E' \triangleright a' \curlyvee b$, and $Q' = E'[f/c' \mid Q_1]$. From this, Lemma 42 gives us directly:

$$Q \mid R \xrightarrow{\tau}_{\mathrm{bn}} \sim_{\mathrm{bn}} \boldsymbol{\nu}f\,(Q' \mid R')$$

We can now extract the arc from R':

$$R' \ \equiv\ \boldsymbol{\nu}\widetilde{n}\left(R'' \mid \sigma\right) \enspace ,$$

where σ is a parallel composition of arcs and R'' contains no active arc. We then have

$$P' \mid R' \ \equiv\ (\boldsymbol{\nu}\widetilde{n})\left(P' \mid \sigma \mid R''\right) \enspace ,$$

and similarly for $Q' \mid R'$. We can conclude by remarking that $P' \sim_{\mathrm{bn}} Q'$ entails $P' \mid \sigma \sim_{\mathrm{bn}} Q' \mid \sigma$, and using up to restriction to remove the topmost restrictions.

General case: Consider now the case where R is an arbitrary process. We reason by induction on R, to show that $P \sim_{\mathrm{bn}} Q$ implies $P \mid R \sim_{\mathrm{bn}} Q \mid R$. The cases where R is a prefixed process or $R = \mathbf{0}$ are treated by the result above.

The case where $R = u/v$ holds by definition of $\sim_{\mathrm{bn}}$: $P \sim_{\mathrm{bn}} Q$ implies $P \mid u/v \ \sim_{\mathrm{bn}}\ Q \mid u/v$.

If $R = R_1 \mid R_2$, then by induction $P \mid R_1 \ \sim_{\mathrm{bn}}\ Q \mid R_1$, which gives, by induction again, $(P \mid R_1) \mid R_2 \ \sim_{\mathrm{bn}}\ (Q \mid R_1) \mid R_2$, hence the result by associativity of $|$.

Suppose now $R = \boldsymbol{\nu}cR'$. We can suppose w.l.o.g. $c \notin \mathrm{fn}(P) \cup \mathrm{fn}(Q)$. Then by induction $P \mid R' \ \sim_{\mathrm{bn}}\ Q \mid R'$, which gives, by Lemma 43, $(\boldsymbol{\nu}c)(P \mid R') \ \sim_{\mathrm{bn}}\ (\boldsymbol{\nu}c)(Q \mid R')$. Lemma 38 gives $(\boldsymbol{\nu}c)(P \mid R') \ \sim_{\mathrm{bn}}\ P \mid \boldsymbol{\nu}cR'$, and similarly for Q, hence $P \mid R \ \sim_{\mathrm{bn}}\ Q \mid R$. This concludes the proof. ■

Statement of Theorem 20: *Bisimilarity is a congruence.*

Proof: Follows from Lemmas 43 and 45, closure by prefixes being immediate. ■

Theorem 46 (Soundness)**.** *If $P \sim_{\mathrm{bn}} Q$ then $P \simeq_{\mathrm{bn}} Q$.*

Proof: Preservation of fresh barbs: when f does not appear in any arc, $P \downarrow_f^{\mathrm{bn}}$ is equivalent to $P \xrightarrow{\alpha}$ where α is an input or output label with subject f.

Preservation of general barbs: $P \downarrow_a^{\mathrm{bn}}$ is equivalent to $(P \mid \alpha.\, f) \xrightarrow{\tau}_{\mathrm{bn}} \downarrow_f^{\mathrm{bn}}$ for some α whose subject is a.

Closure under reduction holds trivially since $\longrightarrow_{\mathrm{bn}}$ coincides with $\xrightarrow{\tau}_{\mathrm{bn}}$ and finally, Theorem 20 guarantees closure by contexts. ■

I. Completeness of $\sim_{\mathrm{bn}}$ (Section V-C)

For a prefix α we write $\overline{\alpha}$ for the dual prefix, i.e. $\overline{\overline{a}b} = ab$ and $\overline{ab} = \overline{a}b$. Any prefix α can be also seen as a label.

Lemma 47. *Let P and P' be processes and f a name fresh w.r.t. P and such that $P' \not\downarrow_f^{\mathrm{bn}}$. Then $P \xrightarrow{\alpha}_{\mathrm{bn}} \equiv P'$ if and only if there exists a process P_1 such that $P_1 \downarrow_f^{\mathrm{bn}}$ and*

$$P \mid \overline{\alpha}.\, (\overline{f} \mid f) \longrightarrow_{\mathrm{bn}} P_1 \longrightarrow_{\mathrm{bn}} P' \enspace .$$

Proof: Let us consider the case where α is an input prefix bd, the output case being similar.

Left to right: since $\longrightarrow_{\mathrm{bn}}$ is stable by $\equiv$ we directly suppose that $P \xrightarrow{\alpha}_{\mathrm{bn}} P'$. Then $P = E[ac.\, Q]$ with $E \triangleright a \curlyvee b$ and $P' = E[d/c \mid Q]$. Then

$$\begin{aligned} P_\alpha &\overset{\mathrm{def}}{=} P \mid \overline{\alpha}.\, (\overline{f} \mid f) \\ &\equiv E[ac.\, Q \mid \overline{b}d.\, (\overline{f} \mid f)] \\ &\longrightarrow_{\mathrm{bn}} E[d/c \mid Q \mid \overline{f} \mid f] \overset{\mathrm{def}}{=} P_1 \\ &\longrightarrow_{\mathrm{bn}} E[d/c \mid Q] = P' \enspace . \end{aligned}$$

Right to left: since P_1 and f is fresh in P we know that $\overline{\alpha}$ has been triggered, that is, $P_\alpha \equiv E[ac.\, Q \mid \overline{b}d.\, (\overline{f} \mid f)]$ and $P' \equiv E[d/c \mid Q]$ since P' has no f barb. This means that P is of the form $P \equiv E[ac.\, Q]$. Hence $P \xrightarrow{\alpha}_{\mathrm{bn}} \equiv P'$. ■

Theorem 48 (Completeness)**.** *If $P \simeq_{\mathrm{bn}} Q$ then $P \sim_{\mathrm{bn}} Q$.*

Proof: We show that $\simeq_{\mathrm{bn}}$ is a $\sim_{\mathrm{bn}}$-bisimulation up to $\equiv$. The clause for preservation of $\curlyvee$ is treated with Lemma 35. The one about parallel composition of arcs is trivial, as well as the symmetry and the clause for the τ-transition. We are left with the case for an input or output transition α.

Suppose $P \xrightarrow{\alpha}_{\mathrm{bn}} P'$ and let f be a name fresh wrt to P, P' and Q. Lemma 47 provides us P_1 such that $P_1 \downarrow_f^{\mathrm{bn}}$ and a reduction scheme that we can transport to Q:

$$Q \mid \overline{\alpha}.\,(\overline{f} \,|\, f) \longrightarrow_{\mathrm{bn}} Q_1 \longrightarrow_{\mathrm{bn}} Q_2 \ \ .$$

We know that $P_1 \simeq_{\mathrm{bn}} Q_1$ and $P' \simeq_{\mathrm{bn}} Q_2$, hence $Q_1 \downarrow_f^{\mathrm{bn}}$ and $Q_2 \not\downarrow_f^{\mathrm{bn}}$ (since f is fresh for P'). Another application of Lemma 47 directly gives us $Q \xrightarrow{\alpha}_{\mathrm{bn}}\equiv Q_2$. ■

Statement of Theorem 21: *In* πP*, relations* $\sim_{\mathrm{bn}}$ *and* $\simeq_{\mathrm{bn}}$ *coincide.*

Proof: Consequence of Theorems 48 and 46. ■

J. Encoding Aπ *in* πP

1) Operational correspondence results: We say that $P \in \pi\mathrm{P}$ is *asynchronous* if the continuation of all outputs in P is $\mathbf{0}$. We can remark that the encoding of a process in Aπ is an asynchronous πP process.

We use the following properties of the encoding, where $\longrightarrow_\pi$ is the reduction in the π-calculus. Barbs in the π-calculus are defined in the standard way: $P \downarrow_a$ iff $P \equiv (\boldsymbol{\nu}\widetilde{c})(\alpha.\,P \mid R)$ where α is a prefix whose subject is a. (It is equivalent to $P = E[\alpha.\,P_1]$ for some active context E.)

Lemma 49. *Let P be any π-calculus process.*

1) *If $P \equiv Q$ then $\llbracket P \rrbracket \equiv \llbracket Q \rrbracket$;*
2) *if $\llbracket P \rrbracket \equiv \llbracket Q \rrbracket$ then $P \equiv Q$;*
3) *if $\llbracket P \rrbracket \equiv E_1[\overline{a}b.\,Q_1 \mid ax.\,R_1]$ then $Q_1 \equiv \llbracket Q \rrbracket$, $R_1 \equiv \llbracket R \rrbracket$ and $P \equiv E[\overline{a}b.\,Q \mid a(x).\,R]$ with $\llbracket E \rrbracket[\boldsymbol{\nu}x[\cdot]] \equiv E_1[\cdot]$.*
4) *if $P \longrightarrow_\pi P'$ then $\llbracket P \rrbracket \longrightarrow_{\mathrm{bn}}\simeq_{\mathrm{bn}} \llbracket P' \rrbracket$;*
5) *conversely, if $\llbracket P \rrbracket \longrightarrow_{\mathrm{bn}} P_1$ then there is P' such that $P \longrightarrow_\pi P'$ and $P_1 \simeq_{\mathrm{bn}} \llbracket P' \rrbracket$;*
6) *$P \downarrow_a$ iff $\llbracket P \rrbracket \downarrow_a$.*

Proof:

1) Straightforward.
2) We prove tediously but straightforwardly the following refined statement: if $\llbracket P \rrbracket \equiv R_1$ then there exist R such that $P \equiv R$ and we can obtain R_1 from $\llbracket R \rrbracket$ such that $R_1 \equiv \llbracket R \rrbracket$ but only by moving restrictions of input objects. In the case where $R_1 = \llbracket Q \rrbracket$ we prove that R is necessarily Q (the restrictions of input objects have only one possible position).
3) We combine techniques used in the previous item to get back the fact Q_1 and R_1 are structurally congruent to encoding of processes, and techniques from the proof of Lemma 37 to separate the transformations of $\equiv$ in the subterms Q_1, R_1 guarded by the prefixes $\overline{a}b$, ax from those in the rest of the term.
4) The reduction $\longrightarrow_\pi$ is quotiented by structural congruence, so in the induction proof there is a case handling the rule "if $P \equiv P_1 \longrightarrow_\pi P_1' \equiv P'$ then $P \longrightarrow_\pi\equiv P'$". Since $\llbracket P \rrbracket \equiv \llbracket P_1 \rrbracket$ and $\llbracket P_1' \rrbracket \equiv \llbracket P' \rrbracket$ we only need to know that $\llbracket P_1 \rrbracket \longrightarrow_{\mathrm{bn}}\simeq_{\mathrm{bn}} \llbracket P_1' \rrbracket$ by induction. We also need to know that $(\equiv\longrightarrow_{\mathrm{bn}}\simeq_{\mathrm{bn}}\equiv) \subseteq (\longrightarrow_{\mathrm{bn}}\simeq_{\mathrm{bn}})$ which is true by definition of $\longrightarrow_{\mathrm{bn}}$ and $\simeq_{\mathrm{bn}}$.
   Similarly since the reduction in π is also quotiented by active contexts we also remark that the encoding is compositional, and the encoding of an active context is still active. Also we have to prove that if $P \longrightarrow_{\mathrm{bn}}\simeq_{\mathrm{bn}} Q$ then $P \longrightarrow_{\mathrm{bn}}\simeq_{\mathrm{bn}} Q$ which is true by definition of $\longrightarrow_{\mathrm{bn}}$ and because $\simeq_{\mathrm{bn}}$ is a congruence.
   We now focus on the simple case of $\overline{a}b.\,P \mid a(x).\,Q \longrightarrow_\pi P \mid Q\{b/x\}$. The encoding of the left-hand side reduces into $\boldsymbol{\nu}x(\llbracket P \rrbracket \mid \nicefrac{b}{x} \mid \llbracket Q \rrbracket)$ and we know that x has no negative occurrence in $\llbracket Q \rrbracket$ so by Lemma 19 this process is equivalent to $\llbracket P \rrbracket \mid \llbracket Q \rrbracket\{b/x\}$ which is of the expected shape.
5) If $\llbracket P \rrbracket \longrightarrow_{\mathrm{bn}} Q$, since $\llbracket P \rrbracket$ does not have any arc, the reduction comes from a communication between two prefixes on the same name a: $\llbracket P \rrbracket \equiv E_1[\overline{a}b.\,\llbracket Q \rrbracket \mid ax.\,\llbracket R \rrbracket]$ with E binding x, and then keeping track of all actions operated by $\equiv$ we know that P_1 is of the form $P_1 \equiv E_1[\llbracket Q \rrbracket \mid \nicefrac{b}{x} \mid \llbracket R \rrbracket]$. We can recover $P \equiv E[\overline{a}b.\,Q \mid a(x).\,R] \longrightarrow_\pi E[Q \mid R\{b/x\}] \overset{\mathrm{def}}{=} P'$. Then $\llbracket P' \rrbracket = \llbracket E \rrbracket[\llbracket Q \rrbracket \mid \llbracket R \rrbracket\{b/x\}] \equiv E_1[\llbracket Q \rrbracket \mid \llbracket R \rrbracket\{b/x\}] \simeq_{\mathrm{bn}} P_1$.
6) The implication from left to right is straightforward by induction, but one has to remark that to test the input barb, one needs a synchronous tester $\overline{a}b.\,\omega$. (Note that input barbs are not tested in the *asynchronous* version of behavioural equivalences.) The other implication follows from the fact that there is no arc in $\llbracket P \rrbracket$ so $\llbracket P \rrbracket \downarrow_a$ if and only if $\llbracket P \rrbracket$ contains a prefix whose subject is a (which is equivalent to the fact P does, too).

■

Lemma 50 (Label-syntax correspondence)**.** *If P is only contains trivial arcs (of the form $\nicefrac{e}{e}$) and α is a prefix ac or $\overline{a}c$ then $P \xrightarrow{\alpha}_{\mathrm{bn}}\equiv P'$ iff $P \equiv E[\alpha.\,P_1]$ and $P' \equiv E[\nicefrac{c}{c} \mid P_1]$, with E binding neither a nor c (and P' has only trivial arcs).*

Moreover $P \downarrow_a^{\mathrm{bn}}$ iff $P \xrightarrow{\alpha}_{\mathrm{bn}}$ iff $P \equiv E[\alpha.\,P_1]$.

In addition if $\sigma \triangleright a \curlyvee b$ then $P \xrightarrow{ac}_{\mathrm{bn}} P'$ implies $P \mid \sigma \xrightarrow{bc}_{\mathrm{bn}} P' \mid \sigma$ (resp. $\overline{a}c$, $\overline{b}c$).

Lemma 51 (Label correspondences)**.** *Let P be any π process and f a fresh name.*

1) *If $P \xrightarrow{\overline{a}c}_\pi P'$ then $\llbracket P \rrbracket \xrightarrow{\overline{a}f}_{\mathrm{bn}}\equiv \nicefrac{c}{f} \mid \llbracket P' \rrbracket$.*
2) *If $P \xrightarrow{\overline{a}(c)}_\pi P'$ then $\llbracket P \rrbracket \xrightarrow{\overline{a}f}_{\mathrm{bn}}\equiv \boldsymbol{\nu}c(\nicefrac{c}{f} \mid \llbracket P' \rrbracket)$.*
3) *If $P \xrightarrow{a(x)}_\pi P'$ then $\llbracket P \rrbracket \xrightarrow{af}_{\mathrm{bn}}\equiv \boldsymbol{\nu}x(\nicefrac{f}{x} \mid \llbracket P' \rrbracket)$.*
4) *If $\llbracket P \rrbracket \xrightarrow{\overline{a}f}_{\mathrm{bn}} P_1$ then*
   a) *either $P \xrightarrow{\overline{a}c}_\pi P'$ with $P_1 \equiv \nicefrac{c}{f} \mid \llbracket P' \rrbracket$,*
   b) *or $P \xrightarrow{\overline{a}(c)}_\pi P'$ with $P_1 \equiv \boldsymbol{\nu}c(\nicefrac{c}{f} \mid \llbracket P' \rrbracket)$*
5) *If $\llbracket P \rrbracket \xrightarrow{af}_{\mathrm{bn}} P_1$ then*
   $P \xrightarrow{a(x)}_\pi P'$ with $P_1 \equiv \boldsymbol{\nu}x.\,(\nicefrac{f}{x} \mid \llbracket P' \rrbracket)$

Lemma 52 (Decomposition of transitions, asynchronous πP)**.** *Let P be an asynchronous πP term without visible arc, σ a parallel composition of arcs, and f, g some fresh names.*

1) *If $P \mid \sigma \xrightarrow{\tau}_{\mathrm{bn}} P_t$ then $P \xrightarrow{\overline{a}f}_{\mathrm{bn}} P_1 \xrightarrow{bg}_{\mathrm{bn}} P_2$ with $P_t \sim_{\mathrm{bn}} (\boldsymbol{\nu} fg)(P_2 \mid f/g) \mid \sigma$ and $\sigma \rhd a \curlyvee b$.*
2) *Suppose $P \xrightarrow{\overline{a}f}_{\mathrm{bn}} P_1 \xrightarrow{ag}_{\mathrm{bn}} P_2$ and $\sigma \rhd a \curlyvee b$. Then $P \mid \sigma \xrightarrow{\tau}_{\mathrm{bn}}\sim_{\mathrm{bn}} (\boldsymbol{\nu} fg)(P_2 \mid f/g) \mid \sigma$.*

This result is directly a consequence of the syntax of asynchronous πP as for similar results in Aπ. We use $\sim_{\mathrm{bn}}$ for renaming and concatenating fresh names using Lemma 44.

2) Full abstraction for the encoding of Aπ*:* One inclusion in the full abstraction result actually holds for the whole π-calculus:

Lemma 53. *Let P and Q be π terms. Then $\llbracket P\rrbracket \simeq_{\mathrm{bn}} \llbracket Q\rrbracket$ implies $P \simeq_\pi Q$.*

Proof: The relation $\{(P,Q) \mid \llbracket P\rrbracket \simeq_{\mathrm{bn}} \llbracket Q\rrbracket\}$ is reduction-closed (consequence of Lemma 49), barb-preserving (consequence of Lemma 49), and context-closed: if C is a π context then there exists a πP context C_1 such that $\llbracket C[P]\rrbracket = C_1[\llbracket P\rrbracket]$, similarly for Q; hence $\llbracket P\rrbracket \simeq_{\mathrm{bn}} \llbracket Q\rrbracket$ implies $\llbracket C[P]\rrbracket \simeq_{\mathrm{bn}} \llbracket C[Q]\rrbracket$. ∎

Lemma 54. *Let P and Q be asynchronous π-terms. Then $P \simeq_\pi Q$ implies $\llbracket P\rrbracket \simeq_{\mathrm{bn}} \llbracket Q\rrbracket$.*

Proof: Thanks to Theorem 46 and to the characterisation of barbed congruence by ground bisimilarity in the asynchronous π-calculus [5], we only have to prove that $P \sim_{\mathrm{g}} Q$ implies $\llbracket P\rrbracket \sim_{\mathrm{bn}} \llbracket Q\rrbracket$. We do so by showing that the following relation is a $\sim_{\mathrm{bn}}$-bisimulation up to restriction and $\sim_{\mathrm{bn}}$:

$$\mathcal{R} \overset{\text{def}}{=} (\sim_{\mathrm{g}})^{\mathrm{Sub}} \overset{\text{def}}{=} \{(\llbracket P\rrbracket \mid \sigma, \llbracket Q\rrbracket \mid \sigma) \bigm| P \sim_{\mathrm{g}} Q\}$$

where σ stands for any parallel composition of arcs. In order to do that, we rely on Lemma 51 ($\llbracket P\rrbracket$ is arc-free) to relate non-τ transitions in π and πP, as well as on Lemma 52 to decompose τ-transitions into visible transitions.

We analyse all possible transitions from $\llbracket P\rrbracket \mid \sigma$. We omit intermediate steps to focus on the relevant details.

1) $\llbracket P\rrbracket \mid \sigma \xrightarrow{af}_{\mathrm{bn}}\sim_{\mathrm{bn}} \boldsymbol{\nu} x(f/x \mid \llbracket P'\rrbracket \mid \sigma)$ with $P \xrightarrow{b(x)}_\pi P'$ for some b such that $\sigma \rhd a \curlyvee b$. Drawing the $\sim_{\mathrm{g}}$-diagram yields eventually $\llbracket Q\rrbracket \xrightarrow{bf}_{\mathrm{bn}}\sim_{\mathrm{bn}} \boldsymbol{\nu} x(f/x \mid \llbracket Q'\rrbracket)$. We add σ to derive a transition along the original label af, and relate in $\mathcal{R}$ the resulting processes.
2) $\llbracket P\rrbracket \mid \sigma \xrightarrow{\overline{a}f}_{\mathrm{bn}}\sim_{\mathrm{bn}} \boldsymbol{\nu}\hat{c}(c/f \mid \llbracket P'\rrbracket)$ with $P \xrightarrow{\boldsymbol{\nu}\hat{c}bc}_\pi P'$ with $\hat{c} \in \{\emptyset, \{c\}\}$ and $\sigma \rhd a \curlyvee b$. The reasoning is similar to the previous case.
3) $\llbracket P\rrbracket \mid \sigma \xrightarrow{\tau}_{\mathrm{bn}} P_t \mid \sigma$ with

$$\llbracket P\rrbracket \xrightarrow{\overline{a}f}_{\mathrm{bn}}\xrightarrow{bg}_{\mathrm{bn}} \boldsymbol{\nu}\hat{c}x(c/f \mid g/x \mid \llbracket P''\rrbracket) \overset{\text{def}}{=} P_2$$

$$P \xrightarrow{\boldsymbol{\nu}\hat{c}ac}_\pi\xrightarrow{b(x)}_\pi P''$$

such that $\sigma \rhd a \curlyvee b$ and $P_t \sim_{\mathrm{bn}} \boldsymbol{\nu} fg(P_2 \mid f/g)$. We can again play the ground bisimilarity game and use Lemma 52 to get the same relations on the Q side, to finally get $P \sim_{\mathrm{g}} Q$ and thus:

$$(\llbracket P''\rrbracket \mid \sigma')\ \mathcal{R}\ (\llbracket Q''\rrbracket \mid \sigma')$$

with $\sigma' = \sigma \mid c/f \mid f/g \mid g/x$. We use the up to restriction technique on f, g, x, and $\hat{c}$.

The relation $\mathcal{R}$ is symmetric, and clearly satisfies the clause about joinability and the clause about the addition of arcs. Thus $\mathcal{R}$ is a $\sim_{\mathrm{bn}}$-bisimulation up to restriction and $\sim_{\mathrm{bn}}$. ∎

Theorem 55 (Full abstraction)**.** *Suppose P, Q are processes from the asynchronous π-calculus,* Aπ*. Then $P \simeq_{\mathrm{A}\pi} Q$ iff $\llbracket P\rrbracket \simeq_{\mathrm{bn}} \llbracket Q\rrbracket$.*

K. Encoding of Explicit Fusions

Definition 56. *Let $P \rhd a = b$ be the judgement conjunction of $P \rhd a \leqslant b$ and $P \rhd b \leqslant a$.*

In the following we note φ_P the relation $\{(a,b) \mid P \rhd a\varphi b\}$, e.g. $a \curlyvee_P b$ for the joinability $a \leqslant_P b$ for the reachability or $a =_P b$ for the equality. We will note $P =_{a,b} Q$ iff $P\{b/a\} = Q\{b/a\}$ i.e. if the only difference between P and Q is the exchange of some a and b. We will also write $a = b$ for $\llbracket a = b\rrbracket$ which is $a/b \mid b/a$.

Lemma 57. *If $P =_{a,b} Q$ then $\varphi_{P|a=b} = \varphi_{Q|a=b}$.*

Proof: By symmetry we only consider inclusion. We use induction on the derivation of $(P \mid a = b) \rhd \varphi$ along Definition 30. Only the base case is interesting, when P and Q are arcs and φ is of the form $d \leqslant e$. Then if $\mathrm{n}(\varphi) \subseteq \{a,b\}$ then $(a = b) \rhd \varphi$; if $P \neq Q$ then (P,Q) can only be of the form $(a_1/c, a_2/c)$ (or, resp., $(c/a_1, c/a_2)$) where $a_i \in \{a,b\}$. In this last case φ must be $c \leqslant a_i$ (resp. $a_i \leqslant c$) which is easily achieved by $(a_2/c \mid a = b)$ (resp. $(a_2/c \mid a = b)$). ∎

We extend the definition of $=_{a,b}$ to predicates: $\varphi =_{a,b} \psi$ iff φ and ψ differ only by a, b swaps. Lemma 57 can be slightly generalised:

Lemma 58. *If $P =_{a,b} Q$, $\varphi =_{a,b} \psi$ then $\varphi_{P|a=b} = \psi_{Q|a=b}$.*

Proof: By Lemma 57 we only have to prove that if $R = S \mid a = b$ then $R \rhd \varphi$ implies $R \rhd \psi$, which is easy, since for each case there is a rule of Definition 30 that uses either a/b or b/a to replace one a with a b or vice versa. ∎

Lemma 59. *If $P =_{a,b} Q$ then $(P \mid a = b) \sim_{\mathrm{bn}} (Q \mid a = b)$.*

Proof: Let $\mathcal{R}$ be the corresponding relation, quantifying over every P and Q. We prove that $\mathcal{R}$ is a $\sim_{\mathrm{bn}}$-bisimulation:

1) invariance under arcs is trivial;
2) is implied by Lemma 57;
3) we use Lemma 58 to ensure the communication is possible (when $\mu = \tau$) or that the subject of μ can be related to the subject of the prefix (when $\mu \neq \tau$). The resulting processes are still related through $=_{a,b}$ since this relation commutes with $\equiv$ and contexts.

We conclude by symmetry of $=_{a,b}$. ∎

Lemma 60. *If P and Q are prefix-free, and if their preorders coincide on free names, then $P \sim_{\mathrm{bn}} Q$.*

Proof: The corresponding relation is a $\sim_{\mathrm{bn}}$-bisimulation: all condition checks are straightforward, even when we add arcs since Definition 30 is compositional: $\texttt{preor}(P \mid Q)$ only depends on $\texttt{preor}(P)$ and $\texttt{preor}(Q)$. ■

Lemma 61. *For every fusion process P if $[\![P]\!] \triangleright a \leqslant b$ or $[\![P]\!] \triangleright a \curlyvee b$ then $[\![P]\!] \triangleright a = b$ and $P \equiv P \mid a = b$ (i.e. a and b are related through P's fusions).*

Proof: First we prove that $[\![P]\!] \triangleright a \leqslant b$ implies $[\![P]\!] \triangleright b \leqslant a$ by induction on the derivation of the first judgement. The only interesting case is when we use an arc b/a: then we know that there is the other arc a/b next to b/a, so this is enough. We also know that this is coming from $a = b$ in the original process. Now if the hypothesis is about $a \curlyvee b$ we know that there is a name u such that $a \leqslant u$ and $b \leqslant u$ and we use the first part of the proof to prove $u \leqslant a$ and $u \leqslant b$ which you can compose to get $a \leqslant b$ and $b \leqslant a$. ■

Statement of Theorem 22: *Suppose P and Q are processes of the fusion calculus.*

1) *If $P \equiv Q$ then $[\![P]\!] \simeq_{\mathrm{bn}} [\![Q]\!]$;*
2) *if $P \longrightarrow_{\mathrm{EF}} P'$ then $[\![P]\!] \longrightarrow_{\mathrm{bn}} \approxeq_{\mathrm{bn}} [\![P']\!]$;*
3) *conversely, if $[\![P]\!] \longrightarrow_{\mathrm{bn}} Q$, then $Q \approxeq_{\mathrm{bn}} [\![P']\!]$ for some P' such that $P \longrightarrow_{\mathrm{EF}} P'$.*

Proof: 1) Thanks to Theorem 21, it is enough to prove $[\![P]\!] \sim_{\mathrm{bn}} [\![Q]\!]$, which we do by induction on the derivation of $P \equiv Q$. The standard base cases like associativity are translated directly into structural congruent processes that are therefore related through $\sim_{\mathrm{bn}}$. The other base cases that those dedicated to fusions:

- $[\![a = b \mid P]\!] \sim_{\mathrm{bn}} [\![a = b \mid P\{a/b\}]\!]$ by Lemma 59,
- $[\![a = b \mid b = c]\!] \sim_{\mathrm{bn}} [\![a = c \mid b = c]\!]$ by Lemma 59,
- $[\![a = b]\!] \equiv [\![b = a]\!]$ by commutativity of $|$,
- $[\![a = a]\!] \sim_{\mathrm{bn}} [\![0]\!]$ by Lemma 60,
- $[\![(\boldsymbol{\nu} a)a = b]\!] \sim_{\mathrm{bn}} [\![0]\!]$ by Lemma 60.

We conclude thanks to the fact that $\sim_{\mathrm{bn}}$ is a congruence and an equivalence relation.

2) Thanks to 1) and the fact $\longrightarrow_{\mathrm{bn}}$ is stable by active contexts we only consider the base case of the reduction relation: $R \stackrel{\mathrm{def}}{=} \overline{a}b.\, P \mid ac.\, Q \longrightarrow_{\mathrm{EF}} b = c \mid P \mid Q \stackrel{\mathrm{def}}{=} R'$. Since $\approxeq_{\mathrm{bn}}$ is stable by $\equiv$ and active contexts, we just have to consider the following: $[\![R]\!] \longrightarrow_{\mathrm{bn}} (\boldsymbol{\nu} wy)(b/c \mid w/y \mid wb.\, [\![P]\!] \mid \overline{y}\langle c\rangle.\, [\![Q]\!])$ which has only one deterministic reduction to $[\![R']\!] \mid (\boldsymbol{\nu} wy)(w/y)$ which is strongly bisimilar to $[\![R']\!]$ by Lemma 60.

3) In $[\![R]\!]$ the only visible prefixes $\pi.\, P$ are the form $[\![\pi'.\, P']\!]$. Suppose that $[\![R]\!] \longrightarrow_{\mathrm{bn}} S$ comes from the communication between $\pi_1.\, P$ and $\pi_2.\, Q$ of subjects a and b. We know that $[\![R]\!] \triangleright a \curlyvee b$ which means thanks to Lemma 61 that the communication is possible between $\pi_1'.\, P'$ and $\pi_2'.\, Q'$: for some R', $R \longrightarrow_{\mathrm{EF}} R'$. The process S is then one step away to create the next step and free arcs (corresponding to the encoding of the fusion just created) the continuations $[\![P']\!]$ and $[\![Q']\!]$ which places us into a situation similar to 2). ■

Full abstraction for fair testing in CCS

Tom Hirschowitz*

CNRS and Université de Savoie

Abstract. In previous work with Pous, we defined a semantics for CCS which may both be viewed as an innocent presheaf semantics and as a concurrent game semantics. It is here proved that a behavioural equivalence induced by this semantics on CCS processes is fully abstract for fair testing equivalence.
The proof relies on a new algebraic notion called *playground*, which represents the 'rule of the game'. From any playground, two languages, equipped with labelled transition systems, are derived, as well as a strong, functional bisimulation between them.

Keywords: Programming languages; categorical semantics; presheaf semantics; game semantics; concurrency; process algebra.

1 Introduction

Motivation and previous work Innocent game semantics, invented by Hyland and Ong [20], led to fully abstract models for a variety of functional languages, where programs are interpreted as strategies in a game. Presheaf models [22, 6] were introduced by Joyal et al. as a semantics for process algebras, in particular Milner's CCS [28]. Previous work with Pous [19] (HP) proposes a semantics for CCS, which reconciles these apparently very different approaches. Briefly, (1) on the one hand, we generalise innocent game semantics to both take seriously the possibility of games with more than two players and consider strategies which may accept plays in more than one way; (2) on the other hand, we refine presheaf models to take parallel composition more seriously. This leads to a model of CCS which may both be seen as a concurrent game semantics, and as an innocent presheaf model, as we now briefly recall.

To see that presheaf models are a concurrent, non-innocent variant of game semantics, recall that the base category, say $\mathbb{C}$, for such a presheaf model typically has as objects sequences of labels, or configurations in event structures, morphisms being given by prefix inclusion. Such objects may be understood as plays in some game. Now, in standard game semantics, a strategy is a prefix-closed (non-empty) set of plays. Unfolding the definition, this is the same as a functor $\mathbb{C}^{op} \to 2$, where 2 is the poset category $0 \leqslant 1$: the functor maps a play to 1 when it is accepted by the strategy, and to 0 otherwise. It is known since

* Partially funded by the French ANR projet blanc "Formal Verification of Distributed Components" PiCoq ANR 2010 BLAN 0305 01 and CNRS PEPS CoGIP.

Harmer and McCusker [15] that this notion of strategy does not easily adapt to non-determinism or concurrency. Presheaf semantics only slightly generalises it by allowing strategies to accept a play in several ways. A strategy S now maps each play p to a *set* $S(p)$. The play is accepted when $S(p)$ is non-empty, and, because there are then no functions $S(p) \to \emptyset$, being accepted remains a prefix-closed property of plays. The passage from 2 to more general sets allows to express branching-time semantics.

This links presheaf models with game models, but would be of little interest without the issue of *innocence*. Game models, indeed, do not always accept *any* prefix-closed set of plays S as a strategy: they demand that any choice of move in S depends only on its *view*. E.g., consider the CCS process $P = (a|(b \oplus c))$, where $\oplus$ denotes internal choice, and a candidate strategy accepting the plays $\epsilon, (a), (b), (c), (ab)$, but not (ac). This strategy refuses to choose c after a has been played. Informally, there are two players here, one playing a and the other playing $b \oplus c$; the latter should have no means to know whether a has been played or not. We want to rule out this strategy on the grounds that it is not innocent.

Our technical solution for doing so is to refine the notion of play, making the number of involved players more explicit. Plays still form a category, but they admit a subcategory of *views*, which represent a single player's possible perceptions of the game. This leads us to two equivalent categories of strategies. In the first, strategies are presheaves on views. In the second category, strategies are certain presheaves on arbitrary plays, satisfying an innocence condition. Parallel composition, in the game semantical sense, is best understood in the former category: it merely amounts to copairing. Parallel composition, in the CCS sense, which in standard presheaf models is a complex operation based on some labelling of transitions or events, is here just a move in the game. The full category of plays is necessary for understanding the global behaviour of strategies. It is in particular needed to define our semantic variant of fair testing equivalence, described below. One may think of presheaves on views as a syntax, and of innocent presheaves on plays as a semantics. The combinatorics of passing from local (views) to global (arbitrary plays) are dealt with by right Kan extension.

Discussion of main results In this paper, we further study the semantics of HP, to demonstrate how close it is to operational semantics. For this, we provide two results. The most important, in the author's view, is full abstraction w.r.t. *fair testing semantics*. But the second result might be considered more convincing by many: it establishes that our semantics is fully abstract w.r.t. weak bisimilarity. The reason why it is here considered less important is that it relies on something external to the model itself, namely an LTS for strategies, constructed in an *ad hoc* way. Considering that a process calculus is defined by its reduction semantics, rather than by its possibly numerous LTSs, testing equivalences, which rely on the former, are more intrinsic than various forms of bisimilarity.

Now, why consider fair testing among the many testing equivalences? First of all, let us mention that we could probably generalise our result to any reasonable testing equivalence. Any testing equivalence relies on a 'testing predicate' $\perp$. E.g., for fair testing, it is the set of processes from which any unsuccessful, finite reduction sequence extends to a successful one. We conjecture that for any other predicate $\perp'$, if $\perp'$ is stable under weak bisimilarity, i.e, $P \simeq Q \in \perp'$ implies $P \in \perp'$, then we may interpret the resulting equivalence in terms of strategies, and get a fully abstract semantics. However, this paper is already quite complicated, and pushes generalisation rather far in other respects (see below). We thus chose to remain concrete about the considered equivalence. It was then natural to consider fair testing, as it is both one of the most prominent testing equivalences, and one of the finest. It was introduced independently by Natarajan and Cleaveland [30], and by Brinksma et al. [3, 33] (under the name of *should* testing in the latter paper), with the aim of reconciling the good properties of observation congruence [29] w.r.t. divergence, and the good properties of previous testing equivalences [7] w.r.t. choice. Typically, $a.b + a.c$ and $a.(b \oplus c)$ (where $+$ denotes guarded choice and $\oplus$ denotes internal choice) are not observation congruent, which is perceived as excessive discriminating power of observation congruence. Conversely, $(!\tau) \mid a$ and a are not must testing equivalent, which is perceived as excessive discriminating power of must testing equivalence. Fair testing rectifies both defects, and has been the subject of further investigation, as summarised, e.g., in Cacciagrano et al. [5].

Overview We now give a bit more detail on the contents, warning the reader that this paper is only an extended abstract, and that more technical details may be found in a (submitted) long version [18]. After recalling the game from HP in Section 2, as well as strategies and our semantic fair testing equivalence $\sim_f$ in Section 3, we prove that the translation $(\!|-|\!)$ of HP from CCS to strategies is such that $P \sim_{f,s} Q$ iff $(\!|P|\!) \sim_f (\!|Q|\!)$, where $\sim_{f,s}$ is standard fair testing equivalence (Theorem 4.6).

Our first attempts at proving this where obscured by easy, yet lengthy case analyses over moves. This prompted the search for a way of factoring out what holds 'for all moves'. The result is the notion of *playground*, surveyed in Section 4.1. It is probably not yet in a mature state, and hopefully the axioms will simplify in the future. We show how the game recalled above organises into such a playground $\mathbb{D}^{CCS}$. We then develop the theory in Section 4.2, defining, for any playground $\mathbb{D}$, two LTSs, $\mathcal{T}_{\mathbb{D}}$ and $\mathcal{S}_{\mathbb{D}}$, of *process terms* and *strategies*, respectively, over an alphabet $\mathbb{F}_{\mathbb{D}}$. We then define a map $[\![-]\!] \colon \mathcal{T}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}}$ between them, which we prove is a strong bisimulation.

Returning to the case of CCS in Section 4.3, we obtain that $\mathcal{S}_{\mathbb{D}^{CCS}}$ indeed has strategies as states, and that $\sim_f$ may be characterised in terms of this LTS. Furthermore, unfolding the definition of $\mathcal{T}_{\mathbb{D}^{CCS}}$, we find that its states are terms in a language containing CCS. So, we have maps $\mathrm{ob}(CCS) \overset{\theta}{\hookrightarrow} \mathrm{ob}(\mathcal{T}_{\mathbb{D}^{CCS}}) \xrightarrow{[\![-]\!]} \mathrm{ob}(\mathcal{S}_{\mathbb{D}^{CCS}})$, where ob takes the set of vertices, and with $[\![-]\!] \circ \theta = (\!|-|\!)$. Now, a problem is that CCS and the other two are LTSs on different alphabets, respectively $\mathbb{A}$ and $\mathbb{F}_{\mathbb{D}^{CCS}}$. We thus define morphisms $\mathbb{A} \xleftarrow{\xi} \mathcal{L} \xrightarrow{\chi} \mathbb{F}_{\mathbb{D}^{CCS}}$ and obtain by successive change of base (pullback when rewinding an arrow, postcomposition when following one) a strong bisimulation $[\![-]\!] \colon \mathcal{T}^{\mathbb{A}}_{\mathbb{D}_{CCS}} \to \mathcal{S}^{\mathbb{A}}_{\mathbb{D}_{CCS}}$ over $\mathbb{A}$. We then prove that θ, viewed as a map $\mathrm{ob}(CCS) \hookrightarrow \mathrm{ob}(\mathcal{T}^{\mathbb{A}}_{\mathbb{D}_{CCS}})$, is included in weak bisimilarity, which yields for all P, $P \simeq_{\mathbb{A}} (\!|P|\!)$ (Corollary 4.5). Finally, drawing inspiration from Rensink et al. [33], we prove that CCS and $\mathcal{S}^{\mathbb{A}}_{\mathbb{D}_{CCS}}$ both have enough $\mathbb{A}$-*trees*, in a suitable sense, and that this, together with Corollary 4.5, entails the main result.

Related work Trying to reconcile two mainstream approaches to denotational semantics, we have designed a (first version of a) general framework aiming at an effective theory of programming languages. Other such frameworks exist [31, 32, 36, 10, 4, 2, 17, 1], but most of them, with the notable exception of Kleene coalgebra, attempt to organise the traditional techniques of syntax with variable binding and reduction rules into some algebraic structure. Here, as in Kleene coalgebra, syntax and its associated LTS are derived notions. Our approach may thus be seen as an extension of Kleene coalgebra to an innocent/multi-player setting, yet ignoring quantitative aspects.

In another sense of the word 'framework', recent work of Winskel and colleagues [34] investigates a general notion of concurrent game, based on earlier work by Melliès [26]. In our approach, the idea is that each programming language is interpreted as a playground, and that morphisms of playgrounds denote translations between languages. Winskel et al., instead, construct a (large) bicategory, into which each programming language should embed. Beyond this crucial difference, both approaches use presheaves and factorisation systems, and contain a notion of innocent, concurrent strategy. The precise links between the original notion of innocence, theirs, and ours remain to be better investigated.

Melliès's work [27], although in a deterministic and linear setting, incorporates some 'concurrency' into plays by presenting them as string diagrams. Our innocentisation procedure further bears some similarity with Harmer et al.'s [14] presentation of innocence based on a distributive law. Hildebrandt's approach to fair testing equivalence [16] uses closely related techniques, e.g., presheaves and sheaves — indeed, our innocence condition may be viewed as a sheaf condition. However, (1) his model falls in the aforementioned category of presheaf models for which parallel composition is a complex operation; and (2) he uses sheaves to correctly incorporate infinite behaviour in the model, which is different from our notion of innocence. Finally, direct inspiration is drawn from Girard [12], one of whose aims is to bridge the gap between syntax and semantics.

Perspectives We plan to adapt our semantics to more complicated calculi like π, the Join and Ambients calculi, functional calculi, possibly with extra features (e.g., references, data abstraction, encryption), with a view to eventually generalising it. Preliminary investigations already led to a playground for π, whose adequacy remains to be established. More speculative directions include (1) defining a notion of morphisms for playgrounds, which should induce translations between strategies, and find sufficient conditions for such morphisms to preserve, resp. reflect testing equivalences; (2) generalising playgrounds to apply them beyond programming language semantics; in particular, preliminary work shows that playgrounds easily account for cellular automata; this raises the question of how morphisms of playgrounds would compare with existing notions of simulations between cellular automata [8]; (3) trying and recast the issue of deriving transition systems (LTSs) from reductions [35] in terms of playgrounds.

Notation Set is the category of sets; set is a skeleton of the category of finite sets, namely the category of finite ordinals and arbitrary maps between them; ford is the category of finite ordinals and monotone maps between them. For any category $\mathbb{C}$, $\widehat{\mathbb{C}} = [\mathbb{C}^{op}, \mathsf{Set}]$ denotes the category of presheaves on $\mathbb{C}$, while $\widehat{\mathbb{C}}^f = [\mathbb{C}^{op}, \mathsf{set}]$ and $\hat{\mathbb{C}} = [\mathbb{C}^{op}, \mathsf{ford}]$ respectively denote the categories of presheaves of finite sets and of finite ordinals. One should distinguish, e.g., 'presheaf of finite sets' $\mathbb{C}^{op} \to \mathsf{set}$ from 'finite presheaf of sets' $F \colon \mathbb{C}^{op} \to \mathsf{Set}$. The latter means that the disjoint union $\sum_{c \in \mathrm{ob}(\mathbb{C})} F(c)$ is finite. Throughout the paper, any finite ordinal n is seen as $\{1, \ldots, n\}$ (rather than $\{0, \ldots, n-1\}$).

The notion of LTS that we'll use here is a little more general than the usual one, but this does not change much. We thus refer to the long version for details. Let us just mention that we work in the category Gph of reflexive graphs, and that the category of LTSs over A is for us the slice category Gph/A. LTSs admit a standard change of base functor given by pullback, and its left adjoint given by postcomposition. Given any LTS $p \colon G \to A$, an edge in G is *silent* when it is mapped by p to an identity edge. This straightforwardly yields a notion of weak bisimilarity over A, which is denoted by $\simeq_A$.

Our (infinite) CCS terms are coinductively generated by the typed grammar

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P|Q} \qquad \frac{\Gamma, a \vdash P}{\Gamma \vdash \nu a.P} \qquad \frac{\ldots \qquad \Gamma \vdash P_i \qquad \ldots}{\Gamma \vdash \sum_{i \in n} \alpha_i.P_i}\ (n \in \mathbb{N}),$$

where α_i is either a, $\overline{a}$, for $a \in \Gamma$, or $\heartsuit$. The latter is a 'tick' move used in the definition of fair testing equivalence. As a syntactic facility, we here understand Γ as ranging over $\mathbb{N}$, i.e., the free names of a process always are $1 \ldots n$ for some n. E.g., Γ, a denotes just $n+1$, and $a \in \Gamma$ means $a \in \{1, \ldots, \Gamma\}$.

Definition 1.1. *Let $\mathbb{A}$ be the reflexive graph with vertices given by finite ordinals, edges $\Gamma \to \Gamma'$ given by $\varnothing$ if $\Gamma \neq \Gamma'$, and by $\Gamma + \Gamma + \{id, \heartsuit\}$ otherwise, $id \colon \Gamma \to \Gamma$ being the identity edge on Γ. Elements of the first summand are denoted by $a \in \Gamma$, while elements of the second summand are denoted by $\overline{a}$.*

We view terms as a graph CCS over $\mathbb{A}$ with the usual transition rules. The graph $\mathbb{A}$ only has 'endo'-edges; some LTSs below do use more general graphs.

2 Recalling the game

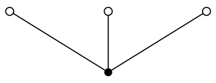
2.1 Positions, Moves, and Plays

In this section, we define plays in our game. For lack of space, we cannot be completely formal. A formal definition, with a gentle introduction to the required techniques, may be found in HP (Section 3). Here is a condensed account. We start by defining a category $\mathbb{C}$. Then, *positions* in our game are defined to be particular finite presheaves in $\widehat{\mathbb{C}}^f$. *Moves* in our game are defined as certain *cospans* $X \xrightarrow{s} M \xleftarrow{t} Y$ in $\widehat{\mathbb{C}}^f$, where t indicates that Y is the *initial* position of the move, while s indicates that X is the *final* position. *Plays* are then defined as finite composites of moves in the bicategory $\mathsf{Cospan}(\widehat{\mathbb{C}}^f)$ of cospans in $\widehat{\mathbb{C}}^f$. By construction, positions and plays form a subbicategory, called $\mathbb{D}_v^{CCS}$.

In order to motivate the definition of our base category $\mathbb{C}$, recall that (directed, multi) graphs may be seen as presheaves over the category freely generated by the graph with two objects $\star$ and $[1]$, and two edges $s, t \colon \star \to [1]$. Any presheaf G represents the graph with vertices in $G(\star)$ and edges in $G[1]$, the source and target of any $e \in G[1]$ being respectively $G(s)(e)$ and $G(t)(e)$. A way to visualise how such presheaves represent graphs is to compute their *categories of elements* [25]. Recall that the category of elements $\int G$ for a presheaf G over $\mathbb{C}$ has as objects pairs (c, x) with $c \in \mathbb{C}$ and $x \in F(c)$, and as morphisms $(c, x) \to (d, y)$ all morphisms $f \colon c \to d$ in $\mathbb{C}$ such that $F(f)(y) = x$. This category admits a canonical functor π_F to $\mathbb{C}$, and F is the colimit of the composite $\int F \xrightarrow{\pi_F} \mathbb{C} \xrightarrow{\mathsf{y}} \widehat{\mathbb{C}}$ with the Yoneda embedding. Hence, e.g., the category of elements for the representable presheaf over $[1]$ is the poset $(\star, s) \to ([1], id_{[1]}) \leftarrow (\star, t)$, which could be pictured as , thus recovering some graphical intuition.

We now define our base category $\mathbb{C}$. Let us first give the raw definition, and then explain. $\mathbb{C}$ is freely generated from the graph $\mathbb{G}$, defined as follows, plus some equations. As objects, $\mathbb{G}$ has (1) an object $\star$, (2) an object $[n]$ for all $n \in \mathbb{N}$, (3) objects $o_{n,i}$ (output), $\iota_{n,i}$ (input), ν_n (channel creation), π_n^l (left fork), π_n^r (right fork), π_n (fork), $\heartsuit_n$ (tick), $\tau_{n,i,m,j}$ (synchronisation), for all $i \in n, j \in m, n, m \in \mathbb{N}$. $\mathbb{G}$ has edges, for all n, (1) $s_1^n, \ldots, s_n^n \colon \star \to [n]$, (2) $s^c, t^c \colon [n] \to c$, for all $c \in \{\pi_n^l, \pi_n^r, \heartsuit_n\} \cup (\cup_{i \in n}\{o_{n,i}, \iota_{n,i}\})$, (3) $[n+1] \xrightarrow{s^{\nu_n}} \nu_n \xleftarrow{t^{\nu_n}} [n]$, (4) $\pi_n^l \xrightarrow{l^n} \pi_n \xleftarrow{r^n} \pi_n^r$, $o_{n,i} \xrightarrow{\epsilon^{n,i,m,j}} \tau_{n,i,m,j} \xleftarrow{\rho^{n,i,m,j}} \iota_{m,j}$, for all $i \in n, j \in m$. In the following, we omit superscripts when clear from context. As equations, we require, for all n, m, $i \in n$, and $j \in m$, (1) $s^c \circ s_i^n = t^c \circ s_i^n$, (2) $s^{\nu_n} \circ s_i^{n+1} = t^{\nu_n} \circ s_i^n$, (3) $l \circ t = r \circ t$, (4) $\epsilon \circ t \circ s_i = \rho \circ t \circ s_j$.

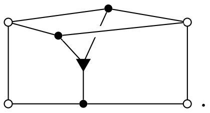
$$\begin{tikzcd}
(\star, s_1) \arrow[dr] & (\star, s_2) \arrow[d] & (\star, s_3) \arrow[dl] \\
& ([3], id_{[3]}) &
\end{tikzcd}$$



In order to explain this seemingly arbitrary definition, let us compute a few categories of elements for representable presheaves. Let us start with an easy one, that of $[3]$ (we implicitly identify any $c \in \mathbb{C}$ with $\mathsf{y}c$). An easy computation shows that it is the poset pictured above. We will think of it as a position with one player $([3], id_{[3]})$ connected to three channels, and draw it as above, where the bullet represents the player, and circles represent channels. (The graphical representation is slightly ambiguous, but nevermind.) In particular, elements over $[3]$ represent ternary players, while elements over $\star$ represent channels. *Positions* are finite presheaves empty except perhaps on $\star$ and $[n]$'s. Let $\mathbb{D}_h^{CCS}$ be the subcategory of $\widehat{\mathbb{C}}^f$ consisting of positions and monic arrows between them.

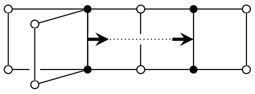
A more difficult category of elements is that of π_2. It is the poset generated by the graph on the left:

$$\begin{tikzcd}
lss_1 = rss_1 \arrow[r] \arrow[rrr, bend left] \arrow[dd, equal] & ls \arrow[d] & & rs \arrow[d] & lss_2 = rss_2 \arrow[l] \arrow[lll, bend right] \arrow[dd, equal] \\
& l \arrow[r] & id_{\pi_2} & r \arrow[l] & \\
lts_1 = rts_1 \arrow[rr] & & lt = rt \arrow[ul] \arrow[ur] & & lts_2 = rts_2 \arrow[ll]
\end{tikzcd}$$

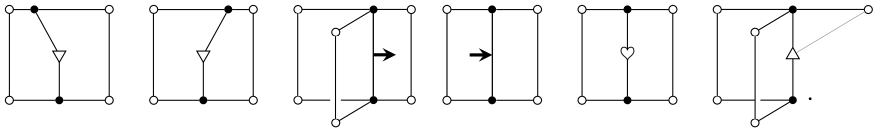


.

We think of it as a binary player (lt) forking into two players (ls and rs), and draw it as on the right. The vertical edges on the outside are actually identities: the reason we draw separate vertices is to identify the top and bottom parts of the picture as the respective images of both legs of the following cospan. First, consider the inclusion $[2] \mid [2] \hookrightarrow \pi_2$: its domain is any pushout of $[s_1, s_2] \colon (\star + \star) \to [2]$ with itself, i.e., the position consisting of two binary players sharing their channels; and the inclusion maps it to the top part of the picture. Similarly, we have a map $[2] \hookrightarrow \pi_2$ given by the player lt and its channels (the bottom part). The cospan $[2] \mid [2] \to \pi_2 \leftarrow [2]$ is called the *local fork move* of arity 2.



For lack of space, we cannot spell out all such categories of elements and cospans. We give pictorial descriptions for $(m, j, n, i) = (3, 3, 2, 1)$ of $\tau_{m,j,n,i}$ on the right and of π^l_n, π^r_n, $o_{m,j}$, $\iota_{n,i}$, $\heartsuit_n$, and ν_n below:

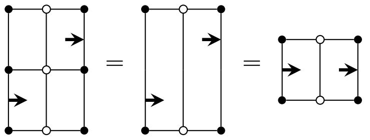


.

In each case, the representable is the middle object of a cospan determined by the top and bottom parts of the picture. E.g., for synchronisation we have $[m]\,{}_j|_i\,[n] \xrightarrow{s} \tau_{m,j,n,i} \xleftarrow{t} [m]\,{}_j|_i\,[n]$, where $[m]\,{}_j|_i\,[n]$ denotes the position X with one m-ary player x, one n-ary player y, such that $X(s_j)(x) = X(s_i)(y)$. Note that there is a crucial design choice in defining the legs of these cospans, which amounts to choosing initial and final positions for our moves.

$$\begin{tikzcd}
& I \arrow[dl] \arrow[d] \arrow[dr] & \\
X \arrow[r] & M & Y \arrow[l]
\end{tikzcd} \tag{1}$$

These cospans altogether form the set of *local moves*, and are the 'seeds' for (global) moves, in the following sense. Calling an *interface* any presheaf consisting only of channels, local moves may be equipped with a canonical interface, consisting of the channels of their initial position. If $X \xrightarrow{s} M \xleftarrow{t} Y$ is a local move (with final position X), and I is its canonical interface, we obtain a commuting diagram (1) in $\widehat{\mathbb{C}^f}$ (with all arrows monic). For any morphism $I \to Z$ to some position Z, pushing $I \to X$, $I \to M$, and $I \to Y$ along $I \to Z$ yields, by universal property of pushout, a new cospan, say $X' \to M' \leftarrow Y'$. Letting *(global) moves* be all cospans obtained in this way, and plays be all composites of moves in $\mathsf{Cospan}(\widehat{\mathbb{C}^f})$, we obtain, as promised a subbicategory $\mathbb{D}^{CCS}_v$.



(2)

Passing from local to global moves allows moves to occur in larger positions. Furthermore, we observe that plays feature some concurrency. For instance, composing two global moves as on the right, we obtain a play in which the order of appearance of moves is no longer visible. In passing, this play embeds into a synchronisation, but is not one, since the input and output moves are not related. This play may be understood as each player communicating with the outside world. We conclude with a useful classification of moves.

Definition 2.1. *A move is* full *iff it is neither a left nor a right fork. We call* $\mathbb{F}$ *the graph of global, full moves.*

Intuitively, a move is full when its final position contains all possible avatars of involved players.

3 Behaviours, strategies, and fair testing

3.1 Behaviours

$$\begin{tikzcd} U \arrow[r] & U' \\ X \arrow[u] \arrow[r] & X' \arrow[u] \end{tikzcd}$$

Recall from HP the category $\mathbb{E}$ whose objects are maps $U \leftarrow X$ in $\widehat{\mathbb{C}}$, such that there exists a play $Y \to U \leftarrow X$, i.e., objects are plays, where we forget the final position. Its morphisms $(U \leftarrow X) \to (U' \leftarrow X')$ are commuting diagrams as on the right with all arrows monic. Morphisms $U \to U'$ in $\mathbb{E}$ represent extensions of U, both spatially (i.e., embedding into a larger position) and dynamically (i.e., adding more moves).

We may relativise this category $\mathbb{E}$ to a particular position X, yielding a category $\mathbb{E}(X)$ of plays on X: take the fibre over X of the functor $\mathrm{cod}\colon \mathbb{E} \to \mathbb{D}_h^{CCS}$ mapping any play $U \leftarrow X$ to its initial position X. The objects of $\mathbb{E}(X)$ are just plays $(U \leftarrow X)$ on X, and morphisms are morphisms of plays whose lower border is id_X. This leads to a category of 'naive' strategies, called behaviours.

Definition 3.1. *The category* B_X *of* behaviours *on* X *is the category* $\widehat{\mathbb{E}(X)}^f$ *of presheaves of finite sets on* $\mathbb{E}(X)$*.*

Behaviours suffer from the deficiency of allowing unwanted cooperation between players. HP (Example 12) exhibits a behaviour where players choose with whom they synchronise, which clearly is not allowed in CCS.

3.2 Strategies

To rectify this, we consider the full subcategory $\mathbb{V}$ of $\mathbb{E}$ consisting of *views*, i.e., compositions of basic local moves. We relativise views to a position X, as follows. Let, for any $n \in \mathbb{N}$, $[n]$ denote the single n-ary player, i.e., a single player connected to n distinct channels. Players of X are in 1-1 correspondence with pairs (n, x), with $x\colon [n] \to X$ in $\mathbb{D}_h^{ccs}$. Relativisation of $\mathbb{V}$ to X is given by the category $\mathbb{V}_X$ with as objects all pairs (V, x), where $x\colon [n] \to X$, and V is a view with initial position $[n]$. Morphisms are induced by those of $\mathbb{E}$.

Definition 3.2. *The category* S_X *of* strategies *on* X *is the category* $\widehat{\mathbb{V}_X}$ *of presheaves of finite ordinals on* $\mathbb{V}_X$*.*

This rules out undesired behaviours. Recall from HP how to map strategies to behaviours: let first $\mathbb{E}_X$ be the category obtained as $\mathbb{V}_X$ from all plays instead of just views. Then, starting from a strategy S, let S' be obtained by right Kan extension of $i \circ S$ (by $\mathbb{V}_X^{op} \hookrightarrow \mathbb{E}_X^{op}$ being full and faithful), and let $\overline{S} = S' \circ j$. The assignment $S \mapsto \overline{S}$ extends to a full and faithful functor $\overline{(-)}\colon \mathsf{S}_X \to \mathsf{B}_X$. Furthermore, $\overline{(-)}$ admits a left adjoint, which we call *innocentisation*, maping naive strategies (behaviours) to innocent ones. By standard results [24], we have for any S: $\overline{S}(U) = \int_{v \in \mathbb{V}_X} S(v)^{\mathbb{E}_X(v,U)}$. Equivalently, $\overline{S}(U)$ is a limit of $(\mathbb{V}_X/U)^{op} \xrightarrow{\text{dom}} \mathbb{V}_X^{op} \xrightarrow{S} \mathsf{ford} \hookrightarrow \mathsf{set}$.

$$\begin{tikzcd}
\mathbb{V}_X^{op} \arrow[r, hook] \arrow[d, "S"'] & \mathbb{E}_X^{op} \arrow[d, "S'"'] & \mathbb{E}(X)^{op} \arrow[l, hook', "j"] \arrow[dl, "\overline{S}"] \\
\mathsf{ford} \arrow[r, hook, "i"] & \mathsf{set},
\end{tikzcd}$$

3.3 Decomposition: a syntax for strategies

Our definition of strategies is rather semantic in flavour. Indeed, presheaves are akin to domain theory. However, they also lend themselves well to a syntactic description. First, it is shown in HP that strategies on an arbitrary position X are in 1-1 correspondence with families of strategies indexed by the players of X. Recall that $[n]$ is the position consisting of one n-ary player, and that players of X may be defined as elements of $\mathrm{Pl}(X) = \sum_{n \in \mathbb{N}} \mathbb{D}_h^{ccs}([n], X)$.

Proposition 3.3. *We have* $\mathsf{S}_X \cong \prod_{(n,x) \in \mathrm{Pl}(X)} \mathsf{S}_{[n]}$*. For any* $S \in \mathsf{S}_X$*, we denote by* $S_{(n,x)}$ *the component corresponding to* $(n, x) \in \mathrm{Pl}(X)$ *under this isomorphism.*

This result yields a construction letting two strategies interact along an *interface*, i.e., a position consisting only of channels. This will be the basis of our semantic definition of fair testing equivalence. Consider any pushout Z of $X \leftarrow I \rightarrow Y$ where I is an interface. We have

Corollary 3.4. $\mathsf{S}_Z \cong \mathsf{S}_X \times \mathsf{S}_Y$*.*

Proof. We have $\mathbb{V}_Z \cong \mathbb{V}_X + \mathbb{V}_Y$, and conclude by universal property of coproduct.

We denote by $[S, T]$ the image of $(S, T) \in \mathsf{S}_X \times \mathsf{S}_Y$ under this isomorphism.

So, strategies over arbitrary positions may be decomposed into strategies over 'typical' players $[n]$. Let us now explain that strategies over such players may be further decomposed. For any strategy S on $[n]$ and basic move $b\colon [n'] \to [n]$, let the *residual* $S \cdot b$ of S after b be the strategy playing like S after b, i.e., for all $v \in \mathbb{V}_{[n']}$, $(S \cdot b)(v) = S(b \bullet v)$, where $\bullet$ denotes composition in $\mathbb{D}_v^{ccs}$. S is almost determined by its residuals. The only information missing from the $S \cdot b$'s to reconstruct S is the set of initial states and how they relate to the initial states of each $(S \cdot b)$. Thus, for any position X, let id^v_X denote the identity play on X (i.e., nothing happens). For any initial state $\sigma \in S(id_{[n]})$, let $S_{|\sigma}$ be the restriction of S to states derived from σ, i.e., for all v, those $\sigma' \in S(v)$ which are mapped to σ under the restriction $S(!)\colon S(v) \to S(id_{[n]})$. S is determined by its set $S(id_{[n]})$ of initial states, plus the function $(\sigma, b) \mapsto (S_{|\sigma} \cdot b)$. Since $S(id_{[n]})$ is a finite ordinal m, we have for all n:

Theorem 3.5. $\mathsf{S}_{[n]} \cong \sum_{m \in \mathbb{N}} (\prod_{b\colon [n'] \to [n]} \mathsf{S}_{[n']})^m \cong (\prod_{b\colon [n'] \to [n]} \mathsf{S}_{[n']})^\star$.

This result may be understood as saying that strategies form a fixpoint of a certain (polynomial [23]) endofunctor of $\mathsf{Set}/\mathbb{I}$, where $\mathbb{I}$ is the set of 'typical' players $[n]$. This may be strengthened to show that they form a terminal coalgebra, i.e, that they are in bijection with infinite terms in the following typed grammar, with judgements $n \vdash_{\mathsf{D}} D$ and $n \vdash S$, where D is called a *definite prestrategy* and S is a *strategy*:

$$\frac{\dots\ n_b \vdash S_b\ \dots\ (\forall b\colon [n_b] \to [n] \in [\mathbb{B}]_n)}{n \vdash_{\mathsf{D}} \langle (S_b)_{b \in [\mathbb{B}]_n} \rangle} \qquad \frac{\dots\ n \vdash_{\mathsf{D}} D_i\ \dots\ (\forall i \in m)}{n \vdash \oplus_{i \in m} D_i}\ (m \in \mathbb{N}),$$

where $[\mathbb{B}]_n$ denotes the set of all isomorphism classes of basic moves from $[n]$. We need to use isomorphism classes here, because strategies may not distinguish between different, yet isomorphic basic moves. This achieves the promised syntactic description of strategies. We may readily define the translation of CCS processes, coinductively, as follows. For processes with channels in Γ, we define

$$\begin{array}{rlrl}
(\!|\sum_{i \in n} \alpha_i.P_i|\!) &= \langle b \mapsto \oplus_{\{i \in n \mid b = (\!|\alpha_i|\!)\}} (\!|P_i|\!) \rangle & (\!|a|\!) &= \iota_{\Gamma,a} \\
(\!|\nu a.P|\!) &= \langle \nu_\Gamma \mapsto (\!|P|\!), \_ \mapsto \varnothing \rangle & (\!|\overline{a}|\!) &= o_{\Gamma,a} \\
(\!|P \mid Q|\!) &= \langle \pi^l_\Gamma \mapsto (\!|P|\!), \pi^r_\Gamma \mapsto (\!|Q|\!), \_ \mapsto \varnothing \rangle & (\!|\heartsuit|\!) &= \heartsuit_\Gamma.
\end{array}$$

E.g., $a.P + a.Q + \bar{b}.R$ is mapped to $\langle \iota_{\Gamma,a} \mapsto ((\!|P|\!) \oplus (\!|Q|\!)), o_{\Gamma,b} \mapsto (\!|R|\!), \_ \mapsto \varnothing \rangle$.

3.4 Semantic fair testing

We may now recall our semantic analogue of fair testing equivalence.

Definition 3.6. Closed-world *moves are (the global variants of)* ν,$\heartsuit$,π_n, *and* $\tau_{n,i,m,j}$. *A play is* closed-world *when it is a composite of closed-world moves.*

Let a closed-world play be *successful* when it contains a $\heartsuit$ move. Let then $\perp\!\!\!\perp_Z$ denote the set of behaviours B such that for any unsuccessful, closed-world play $U \leftarrow Z$ and $\sigma \in B(U)$, there exists $f\colon U \to U'$, with U' closed-world and successful, and $\sigma' \in B(U')$ such that $B(f)(\sigma') = \sigma$. Finally, let us say that a triple (I, h, S), for any $h\colon I \to X$ and strategy $S \in \mathsf{S}_X$, *passes* the test consisting of a morphism $k\colon I \to Y$ of positions and a strategy $T \in \mathsf{S}_Y$ iff $\overline{[S, T]} \in \perp\!\!\!\perp_Z$, where Z is the pushout of h and k. Let $S^{\perp\!\!\!\perp}$ denote the set of all such (k, T).

Definition 3.7. *For any* $h\colon I \to X$, $h'\colon I \to X'$, $S \in \mathsf{S}_X$, *and* $S' \in \mathsf{S}_{X'}$, $(I, h, S) \sim_f (I, h', S')$ *iff* $(I, h, S)^{\perp\!\!\!\perp} = (I, h', S')^{\perp\!\!\!\perp}$.

This yields an equivalence relation, analogous to standard fair testing equivalence, which we hence also call fair testing equivalence.

We have defined a translation $(\!|-|\!)$ of CCS processes to strategies, which raises the question of whether it preserves or reflects fair testing equivalence. The rest of the paper is devoted to proving that it does both.

4 Playgrounds and main result

4.1 Playgrounds: a theory of individuality and atomicity

$$\begin{tikzcd}
X \arrow[r, "h"] \arrow[d, "u"'] & X' \arrow[r, "k"] \arrow[d, "u'"'] & X'' \arrow[d, "u''"] \\
Y \arrow[r, "h'"] \arrow[d, "v"'] \arrow[ur, Rightarrow, shorten=4mm, "\alpha"'] & Y' \arrow[r, "k'"] \arrow[d, "v'"'] \arrow[ur, Rightarrow, shorten=4mm, "\alpha'"'] & Y'' \arrow[d, "v''"] \\
Z \arrow[r, "h''"'] \arrow[ur, Rightarrow, shorten=4mm, "\beta"'] & Z' \arrow[r, "k''"'] \arrow[ur, Rightarrow, shorten=4mm, "\beta'"'] & Z'',
\end{tikzcd}$$

We start by trying to give an idea of the notion of playground. To start with, we organise the game into a *(pseudo) double category* [13, 11]. This is a weakening of Ehresmann's double categories [9], where one direction has non strictly associative composition. Although we consider proper pseudo double categories, we often may treat them safely as double categories. A pseudo double category $\mathbb{D}$ consists of a set $\mathrm{ob}(\mathbb{D})$ of *objects*, shared by two categories $\mathbb{D}_h$ and $\mathbb{D}_v$. $\mathbb{D}_h$ is called the *horizontal* category of $\mathbb{D}$, and $\mathbb{D}_v$ is the *vertical* category. Composition in $\mathbb{D}_h$ is denoted by $\circ$, while we use $\bullet$ for $\mathbb{D}_v$. $\mathbb{D}$ is furthermore equipped with a set of *double cells* α, which have vertical, resp. horizontal, domain and codomain, denoted by $\mathrm{dom}_v(\alpha)$, $\mathrm{cod}_v(\alpha)$, $\mathrm{dom}_h(\alpha)$, and $\mathrm{cod}_h(\alpha)$. We picture this as, e.g., α above, where $u = \mathrm{dom}_h(\alpha)$, $u' = \mathrm{cod}_h(\alpha)$, $h = \mathrm{dom}_v(\alpha)$, and $h' = \mathrm{cod}_v(\alpha)$. $\mathbb{D}$ is furthermore equipped with operations for composing double cells: $\circ$ composes them along a common vertical morphism, $\bullet$ composes along horizontal morphisms. Both vertical compositions (of morphisms and double cells) may only be associative up to coherent isomorphism. The full axiomatisation is given by Garner [11], and we here only mention the *interchange law*, which says that the two ways of parsing the above diagram coincide: $(\beta' \circ \beta) \bullet (\alpha' \circ \alpha) = (\beta' \bullet \alpha') \circ (\beta \bullet \alpha)$.

Example 4.1. Returning to the game, we have seen that positions are the objects of the category $\mathbb{D}_h^{CCS}$, whose morphisms are embeddings of positions. But positions are also the objects of the bicategory $\mathbb{D}_v^{CCS}$, whose morphisms are plays. It should seem natural to define a pseudo double category structure with double cells given by commuting diagrams as on the right in $\widehat{\mathbb{C}}$. Here, Y is the initial position and X is the final one; all arrows are mono. This indeed forms a pseudo double category $\mathbb{D}^{CCS}$. Furthermore, for any double category $\mathbb{D}$, let $\mathbb{D}_H$ be the category with objects all morphisms of $\mathbb{D}_v$, and with morphisms $u \to u'$ all double cells α such that $\mathrm{dom}_h(\alpha) = u$ and $\mathrm{cod}_h(\alpha) = u'$. A crucial feature of $\mathbb{D}^{CCS}$ is that the canonical functor $\mathrm{cod}_v \colon \mathbb{D}_H \to \mathbb{D}_h$ mapping any such α to $\mathrm{cod}_v(\alpha)$ is a Grothendieck fibration [21]. This means that one may canonically 'restrict' a play, say $u' \colon X' \to Y'$, along a horizontal morphism $h' \colon Y \to Y'$, and obtain a universal cell as α above, in a suitable sense.

$$\begin{tikzcd}
X \arrow[r, "h"'] \arrow[d, "s"'] & X' \arrow[d, "s'"] \\
U \arrow[r, "k" description] & V \\
Y \arrow[r, "l"] \arrow[u, "t"] & Y' \arrow[u, "t'"']
\end{tikzcd}$$

$$\begin{tikzcd} d \arrow[r, "y"] \arrow[d, dashed, "v^{y,M}"'] \arrow[dr, phantom, "\overset{\alpha^{y,M}}{\dashrightarrow}"] & Y \arrow[d, "M"] \\ d^{y,M} \arrow[r, dashed, "y^M"] & X \end{tikzcd}$$

Playgrounds are pseudo double categories with extra data and axioms, the first of which is that cod_v should be a fibration. To give a brief idea of further axioms, a playground $\mathbb{D}$ is equipped with a set of objects $\mathbb{I}$, called *individuals*, which correspond to our 'typical' players above. Let $\mathrm{Pl}(X) = \sum_{d \in \mathbb{I}} \mathbb{D}_h(d, X)$ denote the set of players of X. It also comes with classes $\mathbb{F}$ and $\mathbb{B}$ of *full*, resp. *basic* moves; and every play (i.e., vertical morphism) is assumed to admit a decomposition into moves in $\mathbb{F} \cup \mathbb{B}$ (hence *atomicity*). Basic moves are assumed to have individuals as both domain and codomain, and *views* are defined to be composites of basic moves. The crucial axiom for innocence to behave well assumes that, for any position Y and player $y \colon d \to Y$, there exists a cell $\alpha^{y,M}$ as above, with $v^{y,M}$ a view, which is unique up to canonical isomorphism of such. Intuitively: any player in the final position of a play has an essentially unique view of the play. A last, sample axiom shows how some sequentiality is enforced, which is useful to tame the concurrency observed in (2). It says that any double cell as in the center below, where b is a basic move and M is any move, decomposes in exactly one of the forms on the left and right:

$$\begin{tikzcd} A \arrow[r] \arrow[d] & X \arrow[d] & & A \arrow[r, "h"] \arrow[d, "w"'] & X \arrow[d, "u"] & & A \arrow[r] \arrow[d] & X \arrow[d] \\ B \arrow[r] \arrow[d] \arrow[ur, phantom, "\alpha_1"] & Y \arrow[d] & \leftsquigarrow & B \arrow[d, "b"'] \arrow[ur, phantom, "\alpha"] & Y \arrow[d, "M"] & \rightsquigarrow & B \arrow[d] \arrow[ur, phantom, "\alpha_1"] & Y \arrow[d] \\ C \arrow[r] \arrow[ur, phantom, "\alpha_2"] & Z & & C \arrow[r, "k"'] & Z & & C \arrow[r] \arrow[ur] & Z. \arrow[u, phantom, "\alpha_2"] \end{tikzcd}$$

The idea is that, C being an individual, if M has a non-trivial restriction to C, then b must be one of its views. Again, for the formal definition, see [18].

Proposition 4.2. $\mathbb{D}^{CCS}$ *forms a playground (basic moves being the* local *ones).*

4.2 Syntaxes and labelled transition systems

Notions of residuals and restrictions defined above for CCS are easily generalised to arbitrary playgrounds. They lead to the exact same syntax as in the concrete case (below Theorem 3.5). They further yield a first, naive LTS over full moves for strategies. The intuition is that there is a transition $S \xrightarrow{M} S'$, for any full move M, when $S \cdot M = S'$. (Residuals $S \cdot M$ are here defined analogously to the case of basic moves $S \cdot b$ above.) An issue with this LTS is that $S \cdot M$ may have several possible initial states, and we have seen that it makes more sense to restrict to a single state before taking residuals. We thus define our LTS $\mathbb{S}_{\mathbb{D}}$ to have as vertices pairs (X, S) of a position X and a *definite* strategy S, i.e., a strategy with exactly one initial state (formally, $S_{(d,x)}(id_d) = 1$ for all $(d, x) \in \mathrm{Pl}(X)$ — recalling that id_d is an (initial) object in $\mathbb{V}_d$). We then say that there is a transition $(X, S) \xrightarrow{M} (X', S')$ for any full move $M \colon X' \to X$, when $S' = (S \cdot M)_{|\sigma'}$, for some initial state σ' of $S \cdot M$.

Example 4.3. Consider a strategy of the shape $S = \langle \pi_n^r \mapsto S_1, \pi_n^l \mapsto S_2, \_ \mapsto \emptyset \rangle$ on $[n]$, with definite S_1 and S_2. There is a π_n transition to the position with two n-ary players x_1 and x_2, equipped with S_1 and S_2, respectively. If now S_1 and S_2 are not definite, any π_n transition has to pick initial states $\sigma_1 \in S_1(id_{[n]})$ and $\sigma_2 \in S_2(id_{[n]})$, i.e., $S \xrightarrow{\pi_n} [(S_1)_{|\sigma_1}] \mid [(S_2)_{|\sigma_2}]$. Here, we use a shorthand notation for pairs (X, S), defined as follows. First, for any strategy S over $[n]$ and position X with exactly one n-ary player x and names in Γ, we denote by $\Gamma \vdash [x : S](a_1, \ldots, a_n)$ the pair (X, S), where $a_i = X(s_i)(x)$, for all $i \in n$. If now X has several players, say $x_1, \ldots, x_p$, of respective arities $n_1, \ldots, n_p$, and $S_1, \ldots, S_p$ are strategies of such arities, we denote by $\Gamma \vdash [x_1 : S_1](a_1^1, \ldots, a_{n_1}^1) \mid \ldots \mid [x_p : S_p](a_1^p, \ldots, a_{n_p}^p)$ the pair $(X, [S_1, \ldots, S_p])$. When they are irrelevant, we often omit Γ, the x_j's, and the a_i^j's, as in our example.

Beyond the one for strategies, there is another syntax one can derive from any playground. Instead of relying on basic moves as before, one now relies on full moves. Thinking of full moves as inference rules (e.g., in natural deduction), the premises of the rule for any full $M \colon X \to Y$ should be those players (d_x, x) of X whose view through M is non-trivial, i.e., is a basic move. We call this set of players $\mathrm{Pl}(M)$. The natural syntax rule is thus the first one above (glossing over some details), which defines *process terms* T. We add a further rule for guarded sum allowing to choose between several moves. One has to be a little careful here, and only allow moves $M \colon X \to Y$ such that $\mathrm{Pl}(M)$ is a singleton. This yields the second rule above, where $n \in \mathbb{N}$, and $\forall i \in n$, M_i is such a move and d_i is the arity of the unique element of $\mathrm{Pl}(M_i)$. Calling T_d the set of infinite terms for this syntax, there is a natural translation map $[\![-]\!] \colon \mathsf{T}_d \to \mathsf{S}_d$ to strategies, for all $d \in \mathbb{I}$, which looks a lot like $(\!|-|\!)$, and an LTS $\mathcal{T}_{\mathbb{D}}$, whose vertices are pairs (X, T) of a position X, with $T \in \prod_{d, x \in \mathrm{Pl}(X)} \mathsf{T}_d$. The main result on playgrounds is

$$\frac{\ldots \ d_x \vdash T_x \ \ldots}{d \vdash M\langle (T_x)_{x \in \mathrm{Pl}(M)} \rangle}$$

$$\frac{\ldots \ d_i \vdash T_i \ \ldots \ (\forall i \in n)}{d \vdash \sum_{i \in n} M_i.T_i}$$

Theorem 4.4. *The map* $[\![-]\!] \colon \mathcal{T}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}}$ *is a functional, strong bisimulation.*

4.3 Change of base and main result

The LTS $\mathcal{S}_{\mathbb{D}^{CCS}}$ obtained for $\mathbb{D}^{CCS}$ is much too fine to be relevant for bisimilarity to make behavioural sense. E.g., the translations of $a|b$ and $b|a$ are not bisimilar. Indeed, labels, i.e., full moves in $\mathbb{F}_{\mathbb{D}^{CCS}}$, bear the information of which player is involved in the transition. So both strategies have a π_Γ translation to a position with two Γ-ary players, say x_1 and x_2. But then, $a \mid b$ has a transition where x_1 plays an input on a, which $b \mid a$ cannot match. Refining the above notation, and omitting $(\!|-|\!)$, we may write the former transitions as $[a \mid b] \xrightarrow{\pi_\Gamma} [a] \mid [b] \xrightarrow{x_1, \iota_{\Gamma,a}} [0] \mid [b]$. There is another problem with this LTS, namely that there are undue transitions. E.g., we have $[\nu a.a] \xrightarrow{\nu_0} [a] \xrightarrow{\iota(a), a} 0$. The transition system does not yet take privacy of channels into account.

Let us first rectify the latter deficiency. To this end, we pull back our LTS $\mathcal{S}_{\mathbb{D}^{CCS}} \to \mathbb{F}_{\mathbb{D}^{CCS}}$ along a morphism of graphs $\mathcal{L} \to \mathbb{F}_{\mathbb{D}^{CCS}}$ defined as follows. Let $\mathcal{L}$

have *interfaced positions* as vertices, i.e., morphisms $h\colon I \to X$ from an interface to a position. I specifies the public channels, and hence we let edges $h \to h'$ be commuting diagrams of the shape (1), where M may be any full move (X being the final position), except inputs and outputs on a channel outside the image of I. We then straightforwardly define $\chi\colon \mathcal{L} \to \mathbb{F}_{\mathbb{D}^{CCS}}$ to map h to X and any diagram above to M. The pullback $\mathcal{S}^{\mathcal{L}}_{\mathbb{D}^{CCS}} \to \mathcal{L}$ of $\mathcal{S}_{\mathbb{D}^{CCS}}$ along χ is rid of undue communications on private channels.

To rectify the other deficiency mentioned above, recalling from Definition 1.1 that $\mathbb{A}$ is the alphabet for CCS, we define a morphism $\xi\colon \mathcal{L} \to \mathbb{A}$ by mapping $(I \to X)$ to its set $I(\star)$ of channels, and any M to (1) $\heartsuit$ if M is a tick move, (2) *id* if M is a synchronisation, a fork, or a channel creation, (3) a if M is an input on $a \in I(\star)$, (4) $\overline{a}$ if M is an output on $a \in I(\star)$. (Positions are formally defined as presheaves to set, hence channels directly form a finite ordinal number.) It is here crucial to have restricted attention to $\mathcal{L}$ beforehand, otherwise we would not know what to do with communications on private channels. Let $\mathcal{S}^{\mathbb{A}}_{\mathbb{D}^{CCS}} = \xi_!(\mathcal{S}^{\mathcal{L}}_{\mathbb{D}^{CCS}})$ be the post-composition of $\mathcal{S}^{\mathcal{L}}_{\mathbb{D}^{CCS}} \to \mathcal{L}$ with ξ.

The obtained LTS $\mathcal{S}^{\mathbb{A}}_{\mathbb{D}^{CCS}} \to \mathbb{A}$ is now ready for our purposes. Proceeding similarly for the LTS $\mathcal{T}_{\mathbb{D}^{CCS}}$ of process terms, we obtain a strong, functional bisimulation $[\![-]\!]\colon \mathrm{ob}(\mathcal{T}^{\mathbb{A}}_{\mathbb{D}^{CCS}}) \to \mathrm{ob}(\mathcal{S}^{\mathbb{A}}_{\mathbb{D}^{CCS}})$ over $\mathbb{A}$. We then prove that $\theta\colon \mathrm{ob}(CCS) \hookrightarrow \mathrm{ob}(\mathcal{T}^{\mathbb{A}}_{\mathbb{D}^{CCS}})$ is included in weak bisimilarity over $\mathbb{A}$, and, easily, that $(\!|-|\!) = [\![-]\!] \circ \theta$.

Corollary 4.5. *For all P, $P \simeq_{\mathbb{A}} (\!|P|\!)$.*

Furthermore, we prove that $\sim_f$ coincides with the standard, LTS-based definition of fair testing, i.e., $P \sim_{f,s} Q$ iff for all sensible T, $(P \mid T \in \bot_s) \Leftrightarrow (Q \mid T \in \bot_s)$, where $P \in \bot_s$ iff any $\heartsuit$-free reduction sequence $P \Rightarrow P'$ extends to one with $\heartsuit$. To obtain our main result, we finally generalise an observation of Rensink and Vogler [33], which essentially says that for fair testing equivalence in CCS, it is sufficient to consider a certain class of tree-like tests, called *failures*. We first slightly generalise the abstract setting of De Nicola and Hennessy [7] for testing equivalences, e.g., to accomodate the fact that strategies are indexed over interfaces. This yields a notion of *effective graph*. We then show that, for any effective graph G over an alphabet A, the result on failures goes through, provided G *has enough A-trees*, in the sense that, up to mild conditions, for any tree t over A, there exists $x \in G$ such that $x \simeq_A t$. Consequently, for any relation $R\colon G \relbar\joinrel\mapstochar\joinrel\rightarrow G'$ between two such effective graphs with enough A-trees, if R is included in weak bisimilarity over A, then R preserves and reflects fair testing equivalence. We thus obtain our main result:

Theorem 4.6. *For any $\Gamma \in \mathbb{N}$, let I_Γ be the interface consisting of Γ channels, and $h_\Gamma\colon I_\Gamma \to [\Gamma]$ be the canonical inclusion. For any CCS processes P and Q over Γ, we have $P \sim_{f,s} Q$ iff $(I_\Gamma, h_\Gamma, (\!|P|\!)) \sim_f (I_\Gamma, h_\Gamma, (\!|Q|\!))$.*

Remark 4.7. Until now, we have considered arbitrary, infinite CCS processes. Let us now restrict ourselves to recursive processes (e.g., in the sense of HP). We obviously still have that $(\!|P|\!) \sim_f (\!|Q|\!)$ implies $P \sim_{f,s} Q$. The converse is less obvious and may be stated in very simple terms: suppose you have two recursive CCS processes P and Q and a test process T, possibly non-recursive, distinguishing P from Q; is there any recursive T' also distinguishing P from Q?

References

[1] B. Ahrens. Initiality for typed syntax and semantics. In C.-H. L. Ong, R. J. G. B. de Queiroz, eds., *WoLLIC*, vol. 7456 of *Lecture Notes in Computer Science*. Springer, 2012.
[2] M. M. Bonsangue, J. J. M. M. Rutten, A. Silva. A Kleene theorem for polynomial coalgebras. In L. de Alfaro, ed., *FOSSACS*, vol. 5504 of *Lecture Notes in Computer Science*. Springer, 2009.
[3] E. Brinksma, A. Rensink, W. Vogler. Fair testing. In I. Lee, S. A. Smolka, eds., *CONCUR*, vol. 962 of *Lecture Notes in Computer Science*. Springer, 1995.
[4] R. Bruni, U. Montanari. Cartesian closed double categories, their lambda-notation, and the pi-calculus. In *LICS '99*. IEEE Computer Society, 1999.
[5] D. Cacciagrano, F. Corradini, C. Palamidessi. Explicit fairness in testing semantics. *Logical Methods in Computer Science*, 5(2), 2009.
[6] G. L. Cattani, G. Winskel. Presheaf models for concurrency. In D. van Dalen, M. Bezem, eds., *CSL*, vol. 1258 of *Lecture Notes in Computer Science*. Springer, 1996.
[7] R. De Nicola, M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34, 1984.
[8] M. Delorme, J. Mazoyer, N. Ollinger, G. Theyssier. Bulking I: An abstract theory of bulking. *Theoretical Computer Science*, 412(30), 2011.
[9] C. Ehresmann. *Catégories et structures*. Dunod, 1965.
[10] F. Gadducci, U. Montanari. The tile model. In G. D. Plotkin, C. Stirling, M. Tofte, eds., *Proof, Language, and Interaction*. The MIT Press, 2000.
[11] R. Garner. *Polycategories*. PhD thesis, University of Cambridge, 2006.
[12] J.-Y. Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3), 2001.
[13] M. Grandis, R. Pare. Limits in double categories. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 40(3), 1999.
[14] R. Harmer, M. Hyland, P.-A. Melliès. Categorical combinatorics for innocent strategies. In *LICS*. IEEE Computer Society, 2007.
[15] R. Harmer, G. McCusker. A fully abstract game semantics for finite nondeterminism. In *LICS '99*, 1999.
[16] T. T. Hildebrandt. Towards categorical models for fairness: fully abstract presheaf semantics of SCCS with finite delay. *Theoretical Computer Science*, 294(1/2), 2003.
[17] T. Hirschowitz. Cartesian closed 2-categories and permutation equivalence in higher-order rewriting. Preprint, 2010.
[18] T. Hirschowitz. Full abstraction for fair testing in CCS. Draft available from http://lama.univ-savoie.fr/~hirschowitz, 2012.
[19] T. Hirschowitz, D. Pous. Innocent strategies as presheaves and interactive equivalences for CCS. *Scientific Annals of Computer Science*, 22(1), 2012. Selected papers from ICE '11.
[20] J. M. E. Hyland, C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2), 2000.
[21] B. Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.
[22] A. Joyal, M. Nielsen, G. Winskel. Bisimulation and open maps. In *LICS '93*. IEEE Computer Society, 1993.
[23] J. Kock. Polynomial functors and trees. *International Mathematics Research Notices*, 2011(3), 2011.
[24] S. Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer, 2nd edition, 1998.
[25] S. MacLane, I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer, 1992.
[26] P.-A. Melliès. Asynchronous games 2: the true concurrency of innocence. In *Proc. CONCUR '04*, vol. 3170 of *LNCS*. Springer Verlag, 2004.
[27] P.-A. Melliès. Game semantics in string diagrams. In *LICS*. IEEE, 2012.
[28] R. Milner. *A Calculus of Communicating Systems*, vol. 92 of *LNCS*. Springer, 1980.
[29] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
[30] V. Natarajan, R. Cleaveland. Divergence and fair testing. In Z. Fülöp, F. Gécseg, eds., *ICALP*, vol. 944 of *Lecture Notes in Computer Science*. Springer, 1995.
[31] T. Nipkow. Higher-order critical pairs. In *LICS '91*. IEEE Computer Society, 1991.
[32] G. D. Plotkin. A structural approach to operational semantics. DAIMI Report FN-19, Computer Science Department, Aarhus University, 1981.
[33] A. Rensink, W. Vogler. Fair testing. *Inf. Comput.*, 205(2), 2007.
[34] S. Rideau, G. Winskel. Concurrent strategies. In *LICS '11*. IEEE Computer Society, 2011.
[35] P. Sewell. From rewrite rules to bisimulation congruences. In D. Sangiorgi, R. de Simone, eds., *CONCUR*, vol. 1466 of *Lecture Notes in Computer Science*. Springer, 1998.
[36] D. Turi, G. D. Plotkin. Towards a mathematical operational semantics. In *LICS '97*, 1997.

Coalgebraic up-to techniques

Damien Pous[1]⋆

CNRS, LIP, ENS Lyon, France
damien.pous@ens-lyon.fr

1 The concrete case of finite automata

A simple algorithm for checking language equivalence of finite automata consists in trying to compute a *bisimulation* that relates them. This is possible because language equivalence can be characterised coinductively, as the largest bisimulation.

More precisely, consider an automaton $\langle S, t, o\rangle$, where S is a (finite) set of states, $t : S \to \mathcal{P}(S)^A$ is a non-deterministic transition function, and $o : S \to 2$ is the characteristic function of the set of accepting states. Such an automation gives rise to a determinised automaton $\langle \mathcal{P}(S), t^\sharp, o^\sharp\rangle$, where $t^\sharp : \mathcal{P}(S) \to \mathcal{P}(S)^A$ and $o^\sharp : \mathcal{P}(S) \to 2$ are the natural extensions of t and o to sets. A *bisimulation* is a relation R between sets of states such that for all sets of states X, Y, $X \mathrel{R} Y$ entails:

1. $o^\sharp(X) = o^\sharp(Y)$, and
2. for all letter a, $t_a^\sharp(X) \mathrel{R} t_a^\sharp(Y)$.

The coinductive characterisation is the following one: *two sets of states recognise the same language if and only if they are related by some bisimulation.*

Taking inspiration from concurrency theory [4,5], one can improve this proof technique by weakening the second item in the definition of bisimulation: given a function f on binary relations, a *bisimulation up to f* is a relation R between states such that for all sets X, Y, $X \mathrel{R} Y$ entails:

1. $o^\sharp(X) = o^\sharp(Y)$, and
2. for all letter a, $t_a^\sharp(X) \mathrel{f(R)} t_a^\sharp(Y)$.

For well-chosen functions f, bisimulations up to f are contained in a bisimulation, so that the improvement is sound. So is the function mapping each relation to its equivalence closure. In this particular case, one recover the standard algorithm by Hopcroft and Karp [2]: two sets can be skipped whenever they can already be related by a sequence of pairwise related states.

One can actually do more, by considering the function c mapping each relation to its congruence closure: the smallest equivalence relation which contains

⋆ Appeared as an invited talk in Proc. CALCO'13, vol. 8089 of LNCS, pages 34-35, Springer, 2013. Work partially funded by the PiCoq and PACE projects, ANR-10-BLAN-0305 and ANR-12IS02001 the argument, and which is compatible w.r.t. set union:

$$\frac{}{X\ c(R)\ X} \qquad \frac{Y\ c(R)\ X}{X\ c(R)\ Y} \qquad \frac{X\ c(R)\ Y \qquad Y\ c(R)\ Z}{X\ c(R)\ Z}$$

$$\frac{X\ R\ Y}{X\ c(R)\ Y} \qquad \frac{X_1\ c(R)\ Y_1 \qquad X_2\ c(R)\ Y_2}{X_1 \cup X_2\ c(R)\ Y_1 \cup Y_2}\ .$$

This is how we obtained HKC [1], an algorithm that can be exponentially faster than Hopcroft and Karp's algorithm or more recent antichain algorithms [7].

2 Generalisation to coalgebra

The above ideas generalise nicely, using the notion of λ-bialgebras [3].

Let T be a monad, F an endofunctor, and λ a distributive law $TF \Rightarrow FT$, a λ-*bialgebra* is a triple $\langle X, \alpha, \beta\rangle$, where $\langle X, \alpha\rangle$ is a F-coalgebra, $\langle X, \beta\rangle$ a T-algebra, and $\alpha \circ \beta = F\beta \circ \lambda_X \circ T\alpha$. Given such a λ-bialgebra, FT-algebra generalise non-deterministic automata: take $X \mapsto 2 \times X^A$ for F, and $X \mapsto \mathcal{P}_f X$ for T. Determinisation through the powerset construction can be generalised as follows [6], when the functor F has a final coalgebra $\langle \varOmega, \omega\rangle$:

$$\begin{tikzcd}
X \arrow[r, "\eta"] \arrow[d, "\alpha"'] & TX \arrow[r, "!"] \arrow[dl, "\alpha^\sharp"] & \varOmega \arrow[d, "\omega"] \\
FTX \arrow[rr, "F!"] & & F\varOmega
\end{tikzcd}$$

Bisimulations up-to can be expressed in a natural way in such a framework. One can in particular consider bisimulations up to congruence, where the congruence is taken w.r.t. the monad T: the fact that λ is a distributive law ensures that this improvement is always sound.

References

1. F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In *POPL*, pages 457–468. ACM, 2013.
2. J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 114, Cornell University, December 1971.
3. B. Klin. Bialgebras for structural operational semantics: An introduction. *TCS*, 412(38):5043–5069, 2011.
4. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
5. D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8:447–479, 1998.
6. A. Silva, F. Bonchi, M. Bonsangue, and J. Rutten. Generalizing the powerset construction, coalgebraically. In *Proc. FSTTCS*, volume 8 of *LIPIcs*, pages 272–283. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.
7. M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer, 2006.

CARTESIAN CLOSED 2-CATEGORIES AND PERMUTATION EQUIVALENCE IN HIGHER-ORDER REWRITING

TOM HIRSCHOWITZ

Abstract. We propose a semantics for permutation equivalence in higher-order rewriting. This semantics takes place in cartesian closed 2-categories, and is proved sound and complete.

1. Introduction

It is known since the end of the 80's that 2-categories with finite products provide a semantics for term rewriting [3]. *Higher-order rewriting* [10, 17, 14, 15] is a framework for specifying rewrite systems on terms with variable binding. Many results from standard term rewriting have been generalised to higher-order rewriting, notably normalisation or confluence results. An important tool for confluence results is the notion of *permutation equivalence*, which was generalised to the higher-order case by Bruggink [1]. He defines a calculus of *proof terms* for specifying reductions in a higher-order rewrite system.

We here propose a categorical semantics for a variant of this calculus, in terms of *cartesian closed 2-categories*. We first define *cartesian closed 2-signatures*, which generalise higher-order rewrite systems, and organise them into a category Sig. We then construct an adjunction

$$\begin{tikzcd} \mathsf{Sig} \arrow[r, bend left, "\mathcal{H}"] \arrow[r, phantom, "\perp"] & \mathsf{2CCCat}, \arrow[l, bend left, "\mathcal{W}"] \end{tikzcd} \tag{1.1}$$

where $\mathsf{2CCCat}$ is the category of small cartesian closed 2-categories. From a given higher-order rewrite system S, the functor $\mathcal{H}$ constructs a cartesian closed 2-category, whose 2-cells are Bruggink's proof terms modulo permutation equivalence, which we prove is the free cartesian closed 2-category generated by S.

We review a number of examples and non-examples, and sketch an extension to deal with the latter.

Related work. Our cartesian closed 2-signatures are a 2-dimensional refinement of *cartesian closed sketches* [16, 4, 9]. Bruggink's calculus of permutation equivalence is close in spirit to Hilken's 2-categorical semantics of the simply-typed λ-calculus [7], but technically different and generalised to arbitrary higher-order rewrite systems. Capriotti [2] proposes a semantics of so-called *flat* permutation equivalence in sesquicategories. More related work is discussed in Section 4.2.

1991 *Mathematics Subject Classification.* D.1.1;D.3.1;F.3.2;F.4.1;

Key words and phrases. Cartesian closed 2-categories, lambda calculus, higher-order rewriting, combinatory reduction systems, categorical semantics.

This work has been partially funded by the French ANR projet blanc "Curry Howard pour la Concurrence" CHOCO BLAN07-1_189926.

2. Cartesian closed signatures and categories

We start by recalling the well-known, or at least folklore, adjunction between what we here call *(cartesian closed) 1-signatures* and cartesian closed categories.

For any set X, define *types* over X by the grammar:

$$A, B, \ldots \in \mathcal{L}_0(X) \quad ::= \quad x \mid 1 \mid A \times B \mid B^A,$$

with $x \in X$.

Proposition 1. *$\mathcal{L}_0$ defines a monad on* Set.

Let the set of *sequents* over a set X be $\mathcal{S}_0(X) = \mathcal{L}_0(X)^* \times \mathcal{L}_0(X)$, i.e., sequents are pairs of a list of types and a type. The assignment $X \mapsto \mathcal{S}_0(X)$ extends to an endofunctor on Set.

Definition 1. *A* 1-signature *consists of a set X_0 of* sorts*, and an $\mathcal{S}_0(X_0)$-indexed set X_1 of* operations*, or equivalently a map $X_1 \to \mathcal{S}_0(X_0)$.*

A *morphism of 1-signatures* $(X_0, X_1) \to (Y_0, Y_1)$ is a pair (f_0, f_1) where $f_i \colon X_i \to Y_i$ such that

$$\begin{tikzcd}
X_1 \arrow[r, "f_1"] \arrow[d] & Y_1 \arrow[d] \\
\mathcal{S}_0(X_0) \arrow[r, "\mathcal{S}_0(f_0)"'] & \mathcal{S}_0(Y_0)
\end{tikzcd}$$

commutes. Morphisms compose in the obvious way, and we have:

Proposition 2. *Composition of morphisms is associative and unital, and hence 1-signatures and their morphisms form a category* Sig_1.

There is a well-known adjunction

$$\begin{tikzcd}
\mathsf{Sig}_1 \arrow[r, bend left, "\mathcal{H}_1"] \arrow[r, phantom, "\perp"] & \mathsf{CCCat} \arrow[l, bend left, "\mathcal{W}_1"]
\end{tikzcd}$$

between 1-signatures and the category CCCat of small cartesian closed categories (with chosen structure) and (strict) cartesian closed functors.

The functor $\mathcal{W}_1$ sends a cartesian closed category $\mathcal{C}$ to the signature with sorts $\mathcal{C}_0$, its set of objects, and with operations $A_1, \ldots, A_n \to A$ the set $\mathcal{C}(\llbracket A_1 \times \ldots \times A_n \rrbracket, \llbracket A \rrbracket)$, where $\llbracket - \rrbracket$ denotes the function $\mathcal{L}_0(\mathcal{C}_0) \to \mathcal{C}_0$ defined by induction:

$$\begin{array}{rcll}
\llbracket c \rrbracket & = & c & c \in \mathcal{C}_0 \\
\llbracket 1 \rrbracket & = & 1 & \\
\llbracket A \times B \rrbracket & = & \llbracket A \rrbracket \times \llbracket B \rrbracket & \\
\llbracket B^A \rrbracket & = & \llbracket B \rrbracket^{\llbracket A \rrbracket}. &
\end{array} \tag{2.1}$$

Conversely, given a 1-signature X, consider the simply-typed λ-calculus with base types in X_0 and constants in X_1. Terms modulo $\beta\eta$ form a category $\mathcal{H}_1(X)$ with objects all types over X_0 and morphisms $A \to B$ all terms of type B with one free variable of type A.

A less often formulated observation, which is useful to us, is that the adjunction $\mathcal{H}_1 \dashv \mathcal{W}_1$ decomposes into two adjunctions

$$\begin{tikzcd}
\mathsf{Sig}_1 \arrow[r, bend left, "\mathcal{L}_1"] \arrow[r, phantom, "\perp"] & \mathcal{L}_1\text{-Alg} \arrow[l, bend left, "\mathcal{U}_1"] \arrow[r, bend left, "\mathcal{F}_1"] \arrow[r, phantom, "\perp"] & \mathsf{CCCat}, \arrow[l, bend left, "\mathcal{V}_1"]
\end{tikzcd}$$

where $\mathcal{L}_1$-Alg is the category of algebras for the monad $\mathcal{L}_1$ defined as follows (and $\mathcal{L}_1$ is shorthand for the functor $X \mapsto (\mathcal{L}_1(X), \mu)$).

For any 1-signature X, let $\mathcal{L}_1(X)$ denote the 1-signature with

- as sorts the set X_0, and
- as operations $\Gamma \vdash A$ the λ-terms $\Gamma \vdash M \colon A$, modulo $\beta\eta$.

$\mathcal{L}_1$ extends to an endofunctor on Sig_1, whose action on morphisms of 1-signatures $X \xrightarrow{f} Y$ substitutes constants $c \in X_1$ with $f_1(c)$. We obtain

Proposition 3. $\mathcal{L}_1$ *is a monad on* Sig_1.

The functor $\mathcal{V}_1$ sends any cartesian closed category $\mathcal{C}$ to the $\mathcal{L}_1$-algebras $(\mathcal{C}_0, \mathcal{C}_1)$ defined as follows. First, $\mathcal{C}_0$ is the set of objects of $\mathcal{C}$. It has a canonical $\mathcal{L}_0$-algebra structure, say $h_0 \colon \mathcal{L}_0(\mathcal{C}_0) \to \mathcal{C}_0$, obtained by interpreting type constructors in $\mathcal{C}$ as in (2.1). Extending this to contexts G by $h_0(G) = \prod_i h_0(G_i)$, let the operations in $\mathcal{C}_1(G, A)$ be the 1-cells in $\mathcal{C}(h_0(G), h_0(A))$. Beware: the domain and codomain of such an operation are really G and A, not $h_0(G)$ and $h_0(A)$. Similarly, interpreting the λ-calculus in $\mathcal{C}$, the 1-signature $(\mathcal{C}_0, \mathcal{C}_1)$ has a canonical $\mathcal{L}_1$-algebra structure, say $h_1 \colon \mathcal{L}_1(\mathcal{C}_0, \mathcal{C}_1) \to (\mathcal{C}_0, \mathcal{C}_1)$:

$$\begin{array}{rcl}
h_1(G \vdash x_i \colon G_i) & = & \pi_i \\
h_1(G \vdash () \colon 1) & = & ! \\
h_1(G \vdash c(M_1, \ldots, M_n)) & = & c \circ (h_1(M_1), \ldots, h_1(M_n)) \\
h_1(G \vdash \lambda x \colon A.M \colon B^A) & = & \varphi(h_1(G, x \colon A \vdash M \colon B)) \\
h_1(G \vdash MN \colon B) & = & ev \circ (h_1(M), h_1(N)) \\
h_1(G \vdash (M, N) \colon A \times B) & = & (h_1(M), h_1(N)) \\
h_1(G \vdash \pi M \colon A) & = & \pi \circ M \\
h_1(G \vdash \pi' M \colon A) & = & \pi' \circ M,
\end{array}$$

where ! is the unique morphism $h_0(G) \to 1$, φ is the bijection $\mathcal{C}(h_0(G, A), h_0(B)) \cong \mathcal{C}(h_0(G), h_0(B^A))$, and ev is the structure morphism $h_0(B^A \times A) \to h_0(B)$.

$\mathcal{L}_1$-algebras are much like cartesian closed categories whose objects are freely generated by their set of sorts. A perhaps useful analogy here is with multicategories $\mathcal{M}$, seen as being close to monoidal categories whose objects are freely generated by those of $\mathcal{M}$ by tensor and unit. Here, the functor $\mathcal{F}_1$ sends any $\mathcal{L}_1$-algebra (X, h) to the cartesian closed category with

- objects the types over X_0, i.e., $\mathcal{L}_0(X_0)$,
- morphisms $A \to B$ the set of operations in $X_1(A, B)$.

This canonically forms a cartesian closed category, with structure induced by the $\mathcal{L}_1$-algebra structure. We define it in more detail in dimension 2 in Section 7.2.

3. Cartesian closed 2-signatures

Given a 1-signature X, let $X_{||}$ denote the set of pairs of parallel operations, i.e., pairs of operations M, N above the same sequent. Otherwise said, $X_{||}$ is the pullback

$$\begin{tikzcd}
X_{||} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & X_1 \arrow[d] \\
X_1 \arrow[r] & \mathcal{S}_0(X_0).
\end{tikzcd}$$

Any morphism $f\colon X \to Y$ of 1-signatures yields a function $f_{||}\colon X_{||} \to Y_{||}$, via the dashed arrow (obtained by universal property of pullback) in

$$\begin{tikzcd}
X_{||} \arrow[rr] \arrow[dd] \arrow[dr, dashed] \arrow[ddrr, phantom, "\lrcorner", very near start] & & X_1 \arrow[dr, "f_1"] \arrow[dd] & \\
& Y_{||} \arrow[rr] \arrow[dd] \arrow[ddrr, phantom, "\lrcorner", very near start] & & Y_1 \arrow[dd] \\
X_1 \arrow[rr] \arrow[dr, "f_1"'] & & \mathcal{S}_0(X_0) \arrow[dr, "\mathcal{S}_0(f_0)"'] & \\
& Y_1 \arrow[rr] & & \mathcal{S}_0(Y_0).
\end{tikzcd}$$

Definition 2. *A* 2-signature *consists of a 1-signature X, plus a set X_2 of* reduction rules *with a function $X_2 \to \mathcal{L}_1(X)_{||}$.*

A *morphism of 2-signatures* $(X, X_2) \to (Y, Y_2)$ is a pair (f, f_2) where $f\colon X \to Y$ is a morphism of 1-signatures and $f_2\colon X_2 \to Y_2$ makes the diagram

$$\begin{tikzcd}
X_2 \arrow[r, "f_2"] \arrow[d] & Y_2 \arrow[d] \\
\mathcal{L}_1(X)_{||} \arrow[r, "\mathcal{L}_1(f_1)_{||}"'] & \mathcal{L}_1(Y)_{||}
\end{tikzcd}$$

commute. We obtain:

Proposition 4. *Composition of morphisms is associative and unital, and hence 2-signatures and their morphisms form a category* Sig.

4. Examples

4.1. **Higher-order rewrite systems.** The prime example of a 2-signature is that for the pure λ-calculus: it has a sort t and operations

$$a\colon t \times t \to t \qquad\qquad \ell\colon t^t \to t,$$

with a reduction rule β above the pair

$$x\colon t^t, y\colon t \vdash a(\ell(x), y), x(y)\colon t$$

in $\mathcal{L}_1(\{t\}, \{\ell, a\})_{||}$. Categorically, this will yield a 2-cell

$$\begin{tikzcd}
& t \times t \arrow[dr, "a"] \arrow[d, phantom, "\Downarrow\beta"] & \\
t^t \times t \arrow[ur, "\ell \times t"] \arrow[rr, bend right, "ev"'] & & t.
\end{tikzcd}$$

This is an example of a *higher-order rewrite system* in the sense of Nipkow [14]. Nipkow's definition is formally different, but his higher-order rewrite systems are in bijection with 2-signatures $h\colon X_2 \to \mathcal{L}_1(X)_{||}$ such that for all rules $r \in X_2$, letting $(\Gamma \vdash M, N\colon A) = h(r)$:

- M is not a variable,
- A is a sort,
- each variable occurring in Γ occurs free in M.

These restrictions help formulating and proving decidability problems on higher-order rewrite systems, whose extension to our setting we leave open.

Let us now anticipate over our main results below and state our soundness and completeness theorem. Given a higher-order rewrite system X, i.e., a 2-signature satisfying the above conditions, let $\mathcal{R}(X)$ be the following locally-preordered 2-category. It has:

- objects are types in $\mathcal{L}_0(X_0)$;
- morphisms $A \to B$ are λ-terms in $\mathcal{L}_1(X)(A \vdash B)$, modulo $\beta\eta$;
- given two parallel morphisms M and N, there is one 2-cell $M \to N$ exactly when there is a sequence of reductions $M \to^* N$ in the usual sense [14].

Proposition 5. *$\mathcal{R}(X)$ is 2-cartesian closed.*

$\mathcal{R}(X)$ and $\mathcal{H}(X)$ have the same objects and morphisms. But because our inference rules for forming reductions are the same as deduction rules for proving the existence of a reduction in the usual sense, we may send any reduction $P \colon M \to N$ to the unique reduction $M \to N$ in $\mathcal{R}(X)$.

Theorem 1 (Soundness and completeness)**.** *This defines an identity-on-objects, identity-on-morphisms, locally full cartesian closed 2-functor $\mathcal{R}(X) \xrightarrow{!} \mathcal{H}(X)$.*

4.2. **Theories with binding.** Understanding reduction rules as equations, it is easy to define the free cartesian closed category generated by a 2-signature. This yields an adjunction

(4.1)
$$\begin{tikzcd} \mathsf{Sig} \arrow[r, bend left, "\mathcal{H}'"] \arrow[r, phantom, "\perp"] & \mathsf{CCCat}. \arrow[l, bend left, "\mathcal{W}'"] \end{tikzcd}$$

This adjunction provides a categorical semantics for theories with binding, which is more general than other approaches by Fiore and Hur [6], Hirschowitz and Maggesi [8], and Zsidó [18], and which is in line with Lambek's seminal paper [11].

If I understand correctly, the motivation for Fiore and Hur's subtle approach is the will to explain the λ-calculus by strictly less than itself. The present framework does not obey this specification, and instead tends to view the λ-calculus as a universal (parameterised) theory with binding.

We end this section by giving a formal construction of the adjunction (4.1). Cartesian closed categories form a full, reflective subcategory of $\mathsf{2CCCat}$, via the functor $\mathcal{J} \colon \mathsf{2CCCat} \to \mathsf{CCCat}$ sending a cartesian closed 2-category $\mathcal{C}$ to the cartesian closed category with:

- objects those of $\mathcal{C}$,
- morphisms those of $\mathcal{C}$, modulo the congruence generated by $f \sim g$ iff there exists a 2-cell $f \to g$.

Here, $\mathcal{J}(\mathcal{C})$ is thought of as the free cartesian closed 2-category with trivial 2-cells (i.e., 0 or 1). The desired adjunction is obtained by composing the adjunctions

$$\begin{tikzcd} \mathsf{Sig} \arrow[r, bend left, "\mathcal{H}"] \arrow[r, phantom, "\perp"] & \mathsf{2CCCat} \arrow[l, bend left, "\mathcal{W}"] \arrow[r, bend left, "\mathcal{J}"] \arrow[r, phantom, "\perp"] & \mathsf{CCCat}. \arrow[l, bend left] \end{tikzcd}$$

4.3. **Non-examples.** Non-examples are given by calculi whose reduction semantics is defined on terms modulo a so-called *structural congruence*, e.g., CCS [12], or the π-calculus [5, 13].

For example, consider the CCS term $(a \mid 0) \mid \overline{a}$. In CCS, it is *structurally* equivalent to $(a \mid \overline{a}) \mid 0$, which then reduces to $0 \mid 0$.

In order to account for this, we would have to consider a 2-signature with reduction rules for structural congruence, here $(M_1 \mid M_2) \mid M_3 \to M_1 \mid (M_2 \mid M_3)$ for associativity, and $M \mid N \to N \mid M$ for commutativity. But then, these reductions count as proper reductions, which departs from the desired computational behaviour. For example, the term $a \mid a$ has an infinite reduction sequence, using commutativity.

Anticipating the development in the next sections, a potential solution is to extend 2-signatures to *2-theories*. For any 2-signature X, let $X_{||}$ denote the set of pairs of reduction rules r, s with a common type $G \vdash M \to N \colon A$. A 2-theory is a 2-signature X, together with a set of equations between parallel reductions, i.e., a subset X_3 of $\mathcal{L}(X)_{||}$.

The main adjunction announced above (1.1) extends to an adjunction between 2-theories and cartesian closed 2-categories. Using equations, we may specify that any reduction $M \to M$ using only structural rules be the identity on M, and consider the computational behaviour of a 2-category to consist of its non-invertible morphisms, as proposed by Hilken [7]. A question is whether for a given calculus this can be done with finitely many equations.

5. A 2-lambda-calculus

We now begin the construction of Adjunction (1.1). We start in this section by defining a monad $\mathcal{L}$ on Sig, which we will use to factor Adjunction (1.1) as

$$\begin{tikzcd} \mathsf{Sig} \arrow[r, bend left, "\mathcal{L}"] \arrow[r, phantom, "\perp"] & \mathcal{L}\text{-Alg} \arrow[l, bend left, "\mathcal{U}"] \arrow[r, bend left, "\mathcal{F}"] \arrow[r, phantom, "\perp"] & \mathsf{2CCCat}, \arrow[l, bend left, "\mathcal{V}"] \end{tikzcd}$$

where:

- $\mathcal{L}$-Alg is the category of $\mathcal{L}$-algebras,
- $\mathcal{L} \colon \mathsf{Sig} \to \mathcal{L}$-Alg is a shortcut for $X \mapsto (\mathcal{L}^2 X \xrightarrow{\mu} \mathcal{L}X)$,
- $\mathcal{U}(\mathcal{L}X \xrightarrow{h} X) = X$,
- $\mathsf{2CCCat}$ is the category of cartesian closed 2-categories, which we define in Section 6.

The left-hand adjunction holds by $\mathcal{L}$ being a monad, thus we concentrate in Section 7 on establishing the right-hand one.

But for now, let us define the monad $\mathcal{L}$.

5.1. **Syntax.** Given a 2-signature $X = ((X_0, X_1), h \colon X_2 \to \mathcal{L}_1(X)_{||})$ (actually $\mathcal{L}_1(X)$ is $\mathcal{L}_1(X_0, X_1)$), we construct a new 2-signature $\mathcal{L}(X)$, whose reduction rules represent reduction sequences in the "higher-order rewrite system" defined by X, modulo permutation equivalence. The 2-signature $\mathcal{L}(X)$ has the same base 1-signature (X_0, X_1), and as reduction rules the terms of a 2λ-calculus (in the sense of Hilken [7]) modulo permutation equivalence, which we now define.

First, terms, called *reductions*, are defined by induction in Figure 1. The typing judgement has the shape $\Gamma \vdash P : M \to N \colon A$, where A is a type in $\mathcal{L}_0(X_0)$, Γ is a list of pairs of a variable and a type, with no variable appearing more than once, M and N are terms of type $\Gamma \vdash A$ modulo $\beta\eta$, and P is a reduction. In the sequel, we often forget the variables in such pairs $(\Gamma \vdash A)$, and identify them with sequents in $\mathcal{S}_0(X_0)$.

When clear from context, we abbreviate substitutions $[M_1/x_1, \ldots, M_n/x_n]$ by $[M_1, \ldots, M_n]$. For a context G, G_i denotes its ith type. Also, for $(M, N) \in \mathcal{L}_1(X)_{||}$, we let $X(M, N)$ be the set of all reduction rules $r \in X_2$ such that $h(r) = (M, N)$. We write $X(\Gamma \vdash M, N \colon A)$ to indicate the common type of M and N. Similarly, $X(G \vdash A)$ denotes the set of operations in X_1 above $G \vdash A$.

$$\frac{\dots \qquad \Gamma \vdash P_i : M_i \to N_i \colon G_i \qquad \dots}{\Gamma \vdash r(P_1, \dots, P_n) : M[M_1, \dots, M_n] \to N[N_1, \dots, N_n] \colon A} \; (r \in X(G \vdash M, N \colon A))$$

$$\frac{\Gamma \vdash P : M_1 \to M_2 \colon A \qquad \Gamma \vdash Q : M_2 \to M_3 \colon A}{\Gamma \vdash P \,;_{M_2}\, Q : M_1 \to M_3 \colon A} \qquad \Gamma, x \colon A, \Delta \vdash x : x \to x \colon A$$

$$\Gamma \vdash () : () \to () \colon 1$$

$$\frac{\Gamma \vdash P_1 : M_1 \to N_1 \colon G_1 \qquad \dots \qquad \Gamma \vdash P_n : M_n \to N_n \colon G_n}{\Gamma \vdash c(P_1, \dots, P_n) : c(M_1, \dots, M_n) \to c(N_1, \dots, N_n) \colon A} \; (c \in X_1(G \vdash A))$$

$$\frac{\Gamma, x \colon A \vdash P : M \to N \colon B}{\Gamma \vdash \lambda x \colon A.P : \lambda x \colon A.M \to \lambda x \colon A.N \colon B^A}$$

$$\frac{\Gamma \vdash P : M \to M' \colon B^A \qquad \Gamma \vdash Q : N \to N' \colon A}{\Gamma \vdash PQ : MN \to M'N' \colon B}$$

$$\frac{\Gamma \vdash P : M \to M' \colon A \qquad \Gamma \vdash Q : N \to N' \colon B}{\Gamma \vdash (P, Q) : (M, N) \to (M', N') \colon A \times B}$$

$$\frac{\Gamma \vdash P : M \to N \colon A \times B}{\Gamma \vdash \pi_{A,B} P : \pi_{A,B} M \to \pi_{A,B} N \colon A} \qquad \frac{\Gamma \vdash P : M \to N \colon A \times B}{\Gamma \vdash \pi'_{A,B} P : \pi'_{A,B} M \to \pi'_{A,B} N \colon B}$$

FIGURE 1. Reductions

5.2. **Substitution.** Next, we define substitution, which has "type"

$$\frac{\Gamma \vdash Q : N \to N' \colon \Delta \qquad \Delta \vdash P : M \to M' \colon A}{\Gamma \vdash P[Q] : M[N] \to M'[N'] \colon A,} \tag{5.1}$$

i.e., given a reduction P and a tuple of reductions Q, it produces a reduction of the indicated type, which we denote $P[Q]$. Here, we denote by $\Gamma \vdash Q : N \to N' \colon \Delta$ a tuple of reductions $\Gamma \vdash Q_i : N_i \to N'_i \colon \Delta_i$, for $1 \le i \le |\Delta|$.

The definition is a bit tricky:

- first we define *left whiskering*, which has "type"
$$\frac{\Gamma \vdash Q : N \to N' \colon \Delta \qquad \Delta \vdash M \colon A}{\Gamma \vdash M[Q] : M[N] \to M'[N'] \colon A,}$$
- then we define *right whiskering*, which has "type"
$$\frac{\Gamma \vdash N \colon \Delta \qquad \Delta \vdash P : M \to M' \colon A}{\Gamma \vdash P[N] : M[N] \to M'[N] \colon A,}$$
(where N denotes a tuple),
- then we define substitution by
$$P[Q] = (P[N] \,;_{M'[N]}\, M'[Q]).$$

There is of course another legitimate definition, namely

$$P[Q] = (M[Q] \,;_{M[N']}\, P[N']).$$

The two will be equated by permutation equivalence in the next section.

Left whiskering is defined by induction, with $\Delta = (x_1 \colon A_1, \ldots, x_n \colon A_n)$ and $Q = (Q_1, \ldots, Q_n)$, by:

$$\begin{array}{rcl} ()[Q] & = & () \\ x_i[Q] & = & Q_i \\ c(M_1, \ldots, M_p)[Q] & = & c(M_1[Q], \ldots, M_p[Q]) \\ (\lambda x \colon B.M)[Q] & = & \lambda x \colon B.(M[Q, x]) \ \ (\text{for } x \notin \mathrm{dom}(\Delta)) \\ (MN)[Q] & = & (M[Q]N[Q]) \\ (M, N)[Q] & = & (M[Q], N[Q]) \\ (\pi_{A,B} M)[Q] & = & \pi_{A,B}(M[Q]) \\ (\pi'_{A,B} M)[Q] & = & \pi'_{A,B}(M[Q]). \end{array}$$

Right whiskering is defined by induction, with $\Delta = (x_1 \colon A_1, \ldots, x_n \colon A_n)$ and $N = (N_1, \ldots, N_n)$, by:

$$\begin{array}{rcl} (r(P_1, \ldots, P_p))[N] & = & r(P_1[N], \ldots, P_p[N]) \\ (P_1 ;_{M''} P_2)[N] & = & (P_1[N] ;_{M''[N]} P_2[N]) \\ ()[N] & = & () \\ x_i[N] & = & N_i \\ c(P_1, \ldots, P_p)[N] & = & c(P_1[N], \ldots, P_p[N]) \\ (\lambda x \colon B.P')[N] & = & \lambda x \colon B.(P'[N, x]) \ \ (\text{for } x \notin \mathrm{dom}(\Delta)) \\ (P_1 P_2)[N] & = & (P_1[N] P_2[N]) \\ (P_1, P_2)[N] & = & (P_1[N], P_2[N]) \\ (\pi_{A,B} P')[N] & = & \pi_{A,B}(P'[N]) \\ (\pi'_{A,B} P')[N] & = & \pi'_{A,B}(P'[N]). \end{array}$$

Definition 3. *Let* $P[Q] = (P[N] ;_{M'[N]} M'[Q])$.

Proposition 6. *Given reductions P and Q as above, the capture-avoiding substitution $P[Q]$ is a well-typed reduction* $\Gamma \vdash P[Q] : M[N] \to M'[N'] \colon A$.

Similarly, there is a weakening operation with "type"

$$\frac{\Gamma \vdash P : M \to N \colon A}{\Gamma, x \colon B \vdash P : M \to N \colon A.} \ (x \notin \Gamma)$$

5.3. **Permutation equivalence.** We now define *permutation equivalence* on reductions, by the equations in Figures 3 and 4, in Appendix A. The *congruence* rules in Figure 3 are bureaucratic: they just say that permutation equivalence is a congruence. The *category* rules make reductions of a given type $\Gamma \vdash A$ into a category. In Figure 4, the *beta and eta* rules mirror the term-level beta and eta rules. Finally, the *lifting* rules lift composition of reductions towards toplevel.

So, $\mathcal{L}(X)$ has sorts X_0, operations X_1, and as reduction rules in $\mathcal{L}(X)(G \vdash M, N \colon A)$ all reductions $G \vdash P : M \to N \colon A$, modulo the equations.

This easily extends to:

Proposition 7. $\mathcal{L}$ *is a functor* $\mathsf{Sig} \to \mathsf{Sig}$.

Now, consider $\mathcal{L}\mathcal{L}(X)$. We define a mapping $\mu_X \colon \mathcal{L}\mathcal{L}(X) \to \mathcal{L}(X)$, by induction on reductions. The typing rule for reduction rules specialises to:

$$\frac{\begin{array}{c}(R \in \mathcal{L}(X)(G \vdash M, N \colon A)) \\ \Gamma \vdash P_1 : M_1 \to N_1 \colon G_1 \qquad \ldots \qquad \Gamma \vdash P_n : M_n \to N_n \colon G_n\end{array}}{\Gamma \vdash R(P_1, \ldots, P_n) : M[M_1, \ldots, M_n] \to N[N_1, \ldots, N_n] \colon A} \ .$$

We set $\mu(R(P_1,\dots,P_n)) = R[\mu(P_1),\dots,\mu(P_n)]$. The other cases just propagate the substitution:

$$\begin{array}{rcl}
P\,;Q & \mapsto & \mu(P)\,;\mu(Q) \\
x & \mapsto & x \\
() & \mapsto & () \\
c(P_1,\dots,P_n) & \mapsto & c(\mu(P_1),\dots,\mu(P_n)) \\
\lambda x\colon A.P & \mapsto & \lambda x\colon A.\mu(P) \\
PQ & \mapsto & \mu(P)\mu(Q) \\
(P,Q) & \mapsto & (\mu(P),\mu(Q)) \\
\pi P & \mapsto & \pi(\mu(P)) \\
\pi' P & \mapsto & \pi'(\mu(P)).
\end{array}$$

Lemma 1. *This defines a natural transformation $\mu\colon \mathcal{L}^2 \to \mathcal{L}$, which makes the diagram*

$$\begin{tikzcd}
\mathcal{L}^3 \arrow[r, "\mathcal{L}\mu"] \arrow[d, "\mu\mathcal{L}"'] & \mathcal{L}^2 \arrow[d, "\mu"] \\
\mathcal{L}^2 \arrow[r, "\mu"'] & \mathcal{L}
\end{tikzcd}$$

commute.

Similarly, there is a natural transformation $\eta\colon id \to \mathcal{L}$, sending each $r \in X(G \vdash M, N\colon A)$ to the reduction $G \vdash r(x_1,\dots,x_n) : M \to N\colon A$, and we have:

Lemma 2. *The diagram*

$$\begin{tikzcd}
\mathcal{L} \arrow[r, "\eta\mathcal{L}"] \arrow[dr, equal] & \mathcal{L}^2 \arrow[d, "\mu"] & \mathcal{L} \arrow[l, "\mathcal{L}\eta"'] \arrow[dl, equal] \\
& \mathcal{L} &
\end{tikzcd}$$

commutes.

Corollary 1. *$(\mathcal{L},\mu,\eta)$ is a monad on* Sig.

A crucial result is:

Proposition 8. *For all $\Gamma \vdash Q : N \to N'\colon \Delta$ and $\Delta \vdash P : M \to M'\colon A$, we have:*

$$\Gamma \vdash P[Q] \equiv (M[Q] \,;_{M[N']} P[N']) : M[N] \to M'[N']\colon A.$$

Proof. We proceed by induction on P. Most cases are bureaucratic. Consider for instance $P = c(P_1,\dots,P_p)$. Then, by definition:

$$P[Q] = (c(P_1[N],\dots,P_p[N]) \,;_{c(M_1'[N],\dots,M_p'[N])} c(M_1'[Q],\dots,M_p'[Q]).$$

By the third lifting rule, this is $\equiv$-related to

$$c(P_1[N] \,;_{M_1'[N]} M_1'[Q],\dots,P_p[N] \,;_{M_p'[N]} M_p'[Q]).$$

By p applications of the induction hypothesis, we obtain

$$c(M_1[Q] \,;_{M_1[N']} P_1[N'],\dots,M_p[Q] \,;_{M_p[N']} P_p[N']),$$

which by lifting again yields the desired result:

$$c(M_1[Q],\dots,M_p[Q]) \,;_{c(M_1[N'],\dots,M_p[N'])} c(P_1[N'],\dots,P_p[N']).$$

The case where something actually happens is $P = r(P_1,\dots,P_p)$, with $r \in X(G \vdash M_0, M_0'\colon A)$ and each $\Delta \vdash P_i : M_i \to M_i'\colon G_i$. Then, the left-hand side is

$$r(P_1[N],\dots,P_p[N]) \,;_{M_0[M_1,\dots,M_n][N]} M_0'[M_1',\dots,M_p'][Q].$$

By lifting, omitting indices of vertical compositions, we have

$$r(P_1[N], \dots, P_p[N]) \equiv r(M_1[N], \dots, M_p[N]) \mathbin{;} M'_0[P_1[N], \dots, P_p[N]].$$

Observing that $M'_0[M'_1, \dots, M'_p][Q] = M'_0[M'_1[Q], \dots, M'_p[Q]]$, the whole is $\equiv$-related to

$$\begin{array}{l} r(M_1[N], \dots, M_p[N]); \\ M'_0[P_1[N], \dots, P_p[N]]; \\ M'_0[M'_1[Q], \dots, M'_p[Q]], \end{array}$$

i.e., by lifting (inductively):

$$\begin{array}{l} r(M_1[N], \dots, M_p[N]); \\ M'_0[(P_1[N] \mathbin{;} M'_1[Q]), \dots, (P_p[N] \mathbin{;} M'_1[Q])]. \end{array}$$

This is by induction hypothesis $\equiv$-related to

$$\begin{array}{l} r(M_1[N], \dots, M_p[N]); \\ M'_0[(M_1[Q] \mathbin{;} P_1[N']), \dots, (M_1[Q] \mathbin{;} P_p[N'])], \end{array}$$

i.e., by lifting again to

$$\begin{array}{l} r(M_1[N], \dots, M_p[N]); \\ M'_0[M_1[Q], \dots, M_1[Q]]; \\ M'_0[P_1[N'], \dots, P_p[N']]. \end{array}$$

The second lifting rule then yields

$$\begin{array}{l} r(M_1[Q], \dots, M_p[Q]); \\ M'_0[P_1[N'], \dots, P_p[N']]. \end{array}$$

And then the first lifting rule yields

$$\begin{array}{l} M_0[M_1[Q], \dots, M_1[Q]]; \\ r(M_1[N'], \dots, M_p[N']); \\ M'_0[P_1[N'], \dots, P_p[N']], \end{array}$$

so, by the second lifting rule again:

$$\begin{array}{l} M_0[M_1[Q], \dots, M_1[Q]]; \\ r(P_1[N'], \dots, P_p[N']), \end{array}$$

i.e., the right-hand side. □

6. Cartesian closed 2-categories

6.1. **Definition.** In a 2-category $\mathcal{C}$, a diagram $A \xleftarrow{p} C \xrightarrow{q} B$ is a *product diagram* iff for all object D, the induced functor

$$\mathcal{C}(D, C) \xrightarrow{\Delta} \mathcal{C}(D, C) \times \mathcal{C}(D, C) \xrightarrow{\mathcal{C}(D,p) \times \mathcal{C}(D,q)} \mathcal{C}(D, A) \times \mathcal{C}(D, B)$$

is an isomorphism. Because this family of functors is 2-natural in D, the inverse functors will also be 2-natural.

Similarly, an object 1 of $\mathcal{C}$ is *terminal* iff for all D the unique functor

$$\mathcal{C}(D, 1) \xrightarrow{!} 1$$

is an isomorphism (where the right-hand 1 is the terminal category).

Definition 4. *A* 2-category with finite products*, or fp 2-category, is a 2-category* $\mathcal{C}$*, equipped with a terminal object and a 2-functor*

$$\mathcal{C} \times \mathcal{C} \xrightarrow{\times} \mathcal{C},$$

plus, for all A and B, a product diagram

$$A \xleftarrow{p} A \times B \xrightarrow{q} B.$$

In such an fp 2-category $\mathcal{C}$, given objects A and B, an *exponential* for them is a pair of an object B^A and a morphism $ev\colon A \times B^A \to B$, such that for all D, the functor

$$\begin{tikzcd}
 & \mathcal{C}(A,A) \times \mathcal{C}(D,B^A) \arrow[r, "\times"] & \mathcal{C}(A \times D, A \times B^A) \arrow[dr, "\mathcal{C}(A \times D, ev)"] & \\
\mathcal{C}(D,B^A) \arrow[ur, "(id_A!, id)"] & & & \mathcal{C}(A \times D, B)
\end{tikzcd}$$

is an isomorphism. As above, because this family of functors is 2-natural in D, the inverse functors will also be 2-natural.

Definition 5. *A* cartesian closed 2-category*, or cartesian closed 2-category, is an fp 2-category, equipped with a choice of exponentials for all pairs of objects. The category* $\mathsf{2CCCat}$ *has cartesian closed 2-categories as objects, and stricly structure-preserving functors between them as morphisms.*

7. Main adjunction

7.1. **Right adjoint.** Given a cartesian closed 2-category $\mathcal{C}$, define $\mathcal{V}(\mathcal{C}) = (\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2)$ as follows. First, let as in Section 2 $(\mathcal{C}_0, \mathcal{C}_1) = \mathcal{V}_1(\mathcal{C})$, and recall the canonical $\mathcal{L}_0$ and $\mathcal{L}_1$-algebra structures h_0 and h_1. Let then the reduction rules in $\mathcal{C}_2(G \vdash M, N\colon A)$ be the 2-cells in $\mathcal{C}(h_0(G), h_0(A))(h_1(M), h_1(N))$, abbreviated to $\mathcal{C}(G,A)(M,N)$ in the sequel.

This signature $\mathcal{V}\mathcal{C}$ has a canonical $\mathcal{L}$-algebra structure $h_2\colon \mathcal{L}(\mathcal{V}\mathcal{C}) \to \mathcal{V}\mathcal{C}$, which we define by induction over terms in Figure 2. In the case for λ, φ denotes the structure isomorphism $\mathcal{C}((\prod \Gamma) \times A, B) \cong \mathcal{C}(\prod \Gamma, B^A)$.

In order for the definition to make sense as a morphism $\mathcal{L}(\mathcal{V}\mathcal{C}) \to \mathcal{V}\mathcal{C}$, we have to check its compatibility with the equations. We have first:

Lemma 3. *For all $\Delta \vdash Q : N \to N'\colon \Gamma$ and $\Gamma \vdash P : M \to M'\colon A$ in $\mathcal{L}(\mathcal{V}\mathcal{C})$,*

$$\begin{tikzcd}
\Delta \arrow[r, bend left, "{M[N]}", ""{name=U, below}] \arrow[r, bend right, "{M'[N']}"', ""{name=D, above}] \arrow[Rightarrow, from=U, to=D, "{h_2(P[Q])}"] & A
\end{tikzcd}
=
\begin{tikzcd}
\Delta \arrow[r, bend left, "N", ""{name=U1, below}] \arrow[r, bend right, "N'"', ""{name=D1, above}] \arrow[Rightarrow, from=U1, to=D1, "h_2(Q)"] & \Gamma \arrow[r, bend left, "M", ""{name=U2, below}] \arrow[r, bend right, "M'"', ""{name=D2, above}] \arrow[Rightarrow, from=U2, to=D2, "h_2(P)"] & A.
\end{tikzcd}$$

Proof. By induction on P and the axioms for cartesian closed 2-categories. □

Lemma 4. *Any two equated reductions are mapped to the same 2-cell in $\mathcal{C}$.*

Proof. We proceed by induction on the proof of the considered equation. The rules of Figure 3 hold because, in $\mathcal{C}$, vertical composition is associative and unital, and equality is a congruence. The beta rule is less easy, so we spell it out.

The left-hand reduction is interpreted in $\mathcal{C}$ as

$$\begin{tikzcd}
\prod \Gamma \arrow[r, bend left, "{(\varphi M, N)}", ""{name=U, below}] \arrow[r, bend right, "{(\varphi M', N')}"', ""{name=D, above}] \arrow[Rightarrow, from=U, to=D, "{(\varphi P, Q)}"] & B^A \times A \arrow[r, "ev"] & B
\end{tikzcd}$$

which is equal to

$(G \vdash x_i : x_i \to x_i \colon G_i) \mapsto (id_{\pi_i} \colon \pi_i \to \pi_i \colon \prod G \to G_i)$

$(G \vdash () : () \to () \colon 1) \mapsto (id_! \colon ! \to ! \colon \prod G \to 1)$

$(\Gamma \vdash c(P_1, \dots, P_n) : c(M_1, \dots, M_n) \to c(N_1, \dots, N_n) \colon A) \mapsto$

$$\begin{tikzcd}
\prod \Gamma \arrow[r, bend left=50, "{(M_1, \dots, M_n)}"] \arrow[r, bend right=50, "{(N_1, \dots, N_n)}"'] \arrow[r, phantom, "\Downarrow P"] & \prod G \arrow[r, "c"] & A
\end{tikzcd} \quad (c \in \mathcal{C}_1(G, A), P = (P_1, \dots, P_n))$$

$(\Gamma \vdash r(P_1, \dots, P_n) : M[M_1, \dots, M_n] \to N[N_1, \dots, N_n] \colon A) \mapsto$

$$\begin{tikzcd}
\prod \Gamma \arrow[r, bend left=50, "{(M_1, \dots, M_n)}"] \arrow[r, bend right=50, "{(N_1, \dots, N_n)}"'] \arrow[r, phantom, "\Downarrow P"] & \prod G \arrow[r, bend left=50, "M"] \arrow[r, bend right=50, "N"'] \arrow[r, phantom, "\Downarrow r"] & A
\end{tikzcd} \qquad (P = (P_1, \dots, P_n))$$

$$(G \vdash P \,;_{M_2} Q : M_1 \to M_3 \colon A) \mapsto \quad \begin{tikzcd}
\prod G \arrow[r, bend left=60, "M_1"] \arrow[r] \arrow[r, bend right=60, "M_3"'] \arrow[r, phantom, shift left=2ex, "\Downarrow P"] \arrow[r, phantom, shift right=2ex, "\Downarrow Q"] & A
\end{tikzcd}$$

$(\Gamma \vdash \lambda x \colon A.P : \lambda x \colon A.M \to \lambda x \colon A.N \colon B^A) \mapsto \varphi(P \colon M \to N \colon (\prod \Gamma) \times A \to B)$

$$(\Gamma \vdash PQ : MN \to M'N' \colon B) \mapsto \quad \begin{tikzcd}
\prod \Gamma \arrow[r, bend left=50, "{(M, N)}"] \arrow[r, bend right=50, "{(M', N')}"'] \arrow[r, phantom, "\Downarrow (P,Q)"] & B^A \times A \arrow[r, "ev"] & B
\end{tikzcd}$$

$$(\Gamma \vdash (P, Q) : (M, N) \to (M', N') \colon A \times B) \mapsto \quad \begin{tikzcd}
\prod \Gamma \arrow[r, bend left=50, "{(M, N)}"] \arrow[r, bend right=50, "{(M', N')}"'] \arrow[r, phantom, "\Downarrow (P,Q)"] & A \times B
\end{tikzcd}$$

$$(\Gamma \vdash \pi_{A,B} P : \pi_{A,B} M \to \pi_{A,B} N \colon A) \mapsto \quad \begin{tikzcd}
\prod \Gamma \arrow[r, bend left=50, "M"] \arrow[r, bend right=50, "N"'] \arrow[r, phantom, "\Downarrow P"] & A \times B \arrow[r, "\pi"] & A
\end{tikzcd}$$

$$(\Gamma \vdash \pi'_{A,B} P : \pi'_{A,B} M \to \pi'_{A,B} N \colon B) \mapsto \quad \begin{tikzcd}
\prod \Gamma \arrow[r, bend left=50, "M"] \arrow[r, bend right=50, "N"'] \arrow[r, phantom, "\Downarrow P"] & A \times B \arrow[r, "\pi'"] & B
\end{tikzcd}$$

Figure 2. The $\mathcal{L}$-algebra structure on $\mathcal{V}(\mathcal{C})$

$$\begin{tikzcd}[column sep=large]
\prod\Gamma \arrow[r, bend left=50, "{(id,N)}"] \arrow[r, bend right=50, "{(id,N')}"'] \arrow[r, phantom, "\Downarrow (id,Q)"] & \prod\Gamma\times A \arrow[r, bend left=50, "\varphi M\times A"] \arrow[r, bend right=50, "\varphi M'\times A"'] \arrow[r, phantom, "\Downarrow \varphi P\times A"] & B^A\times A \arrow[r, "ev"] & B
\end{tikzcd}$$

which is in turn equal (by cartesian closedness of $\mathcal{C}$) to:

$$\begin{tikzcd}[column sep=large]
\prod\Gamma \arrow[r, bend left=50, "{(id,N)}"] \arrow[r, bend right=50, "{(id,N')}"'] \arrow[r, phantom, "\Downarrow (id,Q)"] & \prod\Gamma\times A \arrow[r, bend left=50, "M"] \arrow[r, bend right=50, "M'"'] \arrow[r, phantom, "\Downarrow P"] & B
\end{tikzcd}$$

and hence to the right-hand side of the equation by Lemma 3. The other beta and eta rules similarly hold by the properties of products, internal homs, and terminal object in $\mathcal{C}$.

The lifting rules hold by (particular cases of) the interchange law in $\mathcal{C}$ and functoriality of the structural isomorphisms

$$\mathcal{C}(A\times B,C)\cong\mathcal{C}(B,C^A) \qquad \text{and} \qquad \mathcal{C}(C,A\times B)\cong\mathcal{C}(C,A)\times\mathcal{C}(C,B),$$

which concludes the proof. □

This assignment extends to cartesian closed functors and we have:

Proposition 9. *$\mathcal{V}$ is a functor* $\mathsf{2CCCat}\to\mathsf{Sig}$.

7.2. **Left adjoint.** Given an $\mathcal{L}$-algebra $h\colon\mathcal{L}(X)\to X$, we now construct a cartesian closed 2-category $\mathcal{F}(X,h)$. It has:

- objects the types in $\mathcal{L}_0(X_0)$;
- 1-cells $A\to B$ the terms in $\mathcal{L}_1(X_0,X_1)(A,B)$;
- 2-cells $M\to N\colon A\to B$ the reduction rules in $X_2(M,N)$.

We then must define the cartesian closed 2-category structure, and we start with the 2-category structure. Composition of 1-cells $A\xrightarrow{M}B\xrightarrow{N}C$ is defined to be $A\xrightarrow{N[M]}C$. Vertical composition of 2-cells

$$\begin{tikzcd}[column sep=huge]
A \arrow[r, bend left=50, "M_1"] \arrow[r, "M_2" description] \arrow[r, bend right=50, "M_3"'] \arrow[r, phantom, shift left=3ex, "\Downarrow\alpha"] \arrow[r, phantom, shift right=3ex, "\Downarrow\beta"] & B
\end{tikzcd}$$

is given by $h(\eta(\alpha)\,;_{M_2}\,\eta(\beta))$.

Horizontal composition of 2-cells

$$\begin{tikzcd}[column sep=large]
A \arrow[r, bend left=50, "M"] \arrow[r, bend right=50, "M'"'] \arrow[r, phantom, "\Downarrow\alpha"] & B \arrow[r, bend left=50, "N"] \arrow[r, bend right=50, "N'"'] \arrow[r, phantom, "\Downarrow\beta"] & C
\end{tikzcd} \tag{7.1}$$

is obtained as $h(\beta(\eta(\alpha)))$.

To show that this yields a 2-category structure, the only non obvious point is the interchange law. We deal with it using the following series of results. First, consider the *left whiskering*

$$\begin{tikzcd} A \arrow[r, bend left, "M"] \arrow[r, bend right, "M'"'] \arrow[r, phantom, "\Downarrow \alpha"] & B \arrow[r, "N"] & C \end{tikzcd}$$

of a 2-cell α by a 1-cell N, i.e., the composition $id_N \circ \alpha = h((h(N))(\eta(\alpha)))$.

Lemma 5. *We have:* $h((h(N))(\eta(\alpha))) = h(N[\eta(\alpha)])$.

Proof. Indeed, consider the term $N(\eta(\eta(\alpha)))$ in $\mathfrak{L}(\mathfrak{L}(X))$. Its images by $h \circ \mathfrak{L}(h)$ and $h \circ \mu$ coincide, and are respectively $h((h(N))(\eta(\alpha)))$, i.e., $id_N \circ \alpha$, and $h(N[\eta(\alpha)])$. □

Similarly, consider the *right whiskering*

$$\begin{tikzcd} A \arrow[r, "M"] & B \arrow[r, bend left, "N"] \arrow[r, bend right, "N'"'] \arrow[r, phantom, "\Downarrow \gamma"] & C \end{tikzcd}$$

of a 2-cell γ by a 1-cell M, i.e., the composition $\gamma \circ id_N = h(\gamma(\eta(h(M))))$.

Lemma 6. *We have:* $h(\gamma(\eta(h(M)))) = h(\gamma(M))$.

Proof. Consider $(\eta\gamma)(\eta M)$ in $\mathfrak{L}(\mathfrak{L}(X))$. Its images by $h \circ \mathfrak{L}(h)$ and $h \circ \mu$ coincide, and are respectively $h(\gamma(\eta(h(M))))$ and $h(\gamma(M))$. □

Now, we prove that the two sensible ways of mimicking horizontal composition using whiskering coincide with actual horizontal composition:

Lemma 7. *For any cells as in* (7.1)*,*

$$(\beta \circ id_M) \,;\, (id_{N'} \circ \alpha) = \beta \circ \alpha = (id_N \circ \alpha) \,;\, (\beta \circ id_{M'}).$$

Proof. Consider first the reduction $\eta(\beta(M)) \,;\, \eta(N'[\eta(\alpha)])$ in $\mathfrak{L}(\mathfrak{L}(X))$. Taking $h \circ \mathfrak{L}(h)$ and $h \circ \mu$ as above respectively yields

- $h(\eta(h(\beta(M))) \,;\, \eta(h(N'[\alpha])))$, and
- $h(\beta(M) \,;\, N'[\eta(\alpha)]) = h(\beta(\eta(\alpha)))$,

hence the left-hand equality. Then consider $\eta(N[\eta(\alpha)]) \,;\, \eta(\gamma(M'))$. Evaluating as before yields the right-hand equality. □

Finally, consider any configuration like:

$$\begin{tikzcd} A \arrow[r, bend left=40, "M"] \arrow[r, "M'" description] \arrow[r, bend right=40, "M''"'] \arrow[r, phantom, shift left=2.5ex, "\Downarrow \alpha"] \arrow[r, phantom, shift right=2.5ex, "\Downarrow \beta"] & B \arrow[r, "N"] & C. \end{tikzcd}$$

Lemma 8. *We have* $(id_N \circ \alpha) \,;\, (id_N \circ \beta) = id_N \circ (\alpha \,;\, \beta)$.

Proof. Consider $\eta(N[\eta(\alpha)]) \,;\, \eta(N[\eta(\beta)])$. Evaluating yields equality of

- $h(\eta(h(N[\eta(\alpha)])) \,;\, \eta(h(N[\eta(\beta)])))$, i.e., the left-hand side, and
- $h(N[\eta(\alpha)] \,;\, N[\eta(\beta)])$, i.e., $h(N[\eta(\alpha) \,;\, \eta(\beta)])$ by lifting.

But now consider $N[\eta(\eta(\alpha) \,;\, \eta(\beta))]$. Evaluating yields equality of

- $h(N[\eta(\alpha) \,;\, \eta(\beta)])$, as above, and
- $h(N[\eta(h(\eta(\alpha) \,;\, \eta(\beta)))])$, i.e., $h(N[\eta(\alpha \,;\, \beta)])$ (where $\alpha \,;\, \beta$ denotes vertical composition in our candidate 2-category), i.e., the right-hand side. □

Lemma 9. *The interchange law holds, i.e., for all reduction rules as in*

$$\begin{tikzcd}[column sep=huge]
A \arrow[r, bend left=60, "M_1"{name=U1}] \arrow[r, "M_2"{name=V1}] \arrow[r, bend right=60, "M_3"'{name=D1}] & B \arrow[r, bend left=60, "N_1"{name=U2}] \arrow[r, "N_2"{name=V2}] \arrow[r, bend right=60, "N_3"'{name=D2}] & C,
\arrow[Rightarrow, from=U1, to=V1, "\alpha"]
\arrow[Rightarrow, from=V1, to=D1, "\beta"]
\arrow[Rightarrow, from=U2, to=V2, "\gamma"]
\arrow[Rightarrow, from=V2, to=D2, "\theta"]
\end{tikzcd}$$

we have

$$(\gamma \,;\, \theta) \circ (\alpha \,;\, \beta) = (\gamma \circ \alpha) \,;\, (\theta \circ \beta).$$

Proof. By the previous results, we have

$$\begin{aligned}
&(\gamma \,;\, \theta) \circ (\alpha \,;\, \beta)\\
&= ((\gamma \,;\, \theta) \circ M_1) \,;\, (N_3 \circ (\alpha \,;\, \beta))\\
&= (\gamma \circ M_1) \,;\, (\theta \circ M_1) \,;\, (N_3 \circ \alpha) \,;\, (N_3 \circ \beta)\\
&= (\gamma \circ M_1) \,;\, (N_2 \circ \alpha) \,;\, (\theta \circ M_2) \,;\, (N_3 \circ \beta)\\
&= (\gamma \circ \alpha) \,;\, (\theta \circ \beta).
\end{aligned}$$

□

Now, let us show cartesian closedness. We have a bijection of hom-sets $\mathcal{L}_1(X)(C \vdash A \times B) \cong \mathcal{L}_1(X)(C \vdash A) \times \mathcal{L}_1(X)(C \vdash B)$, given by

$$\begin{array}{rcl}
\mathcal{L}_1(X)(C \vdash A \times B) & \to & \mathcal{L}_1(X)(C \vdash A) \times \mathcal{L}_1(X)(C \vdash B)\\
M & \mapsto & \pi M, \pi' M
\end{array}$$

and

$$\begin{array}{rcl}
\mathcal{L}_1(X)(C \vdash A) \times \mathcal{L}_1(X)(C \vdash B) & \to & \mathcal{L}_1(X)(C \vdash A \times B)\\
M, N & \mapsto & (M, N).
\end{array}$$

These are mutually inverse thanks to the beta and eta rules for products in the simply-typed λ-calculus.

On 2-hom-sets, we have

$$\begin{array}{rcl}
\mathcal{L}(X)(C \vdash M, N \colon A \times B) & \to & \mathcal{L}(X)(C \vdash \pi M, \pi N \colon A) \times \mathcal{L}(X)(C \vdash \pi' M, \pi' N \colon B)\\
P & \mapsto & \pi P, \pi' P
\end{array}$$

and (omitting C)

$$\begin{array}{rcl}
\mathcal{L}(X)(M_1, N_1 \colon A) \times \mathcal{L}(X)(M_2, N_2 \colon B) & \to & \mathcal{L}(X)((M_1, M_2), (N_1, N_2) \colon A \times B)\\
P_1, P_2 & \mapsto & (P_1, P_2),
\end{array}$$

which are mutually inverse thanks to the beta and eta rules for products in Figure 4. We use these to define the desired isomorphism (u, v)

$$X_2(C \vdash M, N \colon A \times B) \cong X_2(C \vdash \pi M, \pi N \colon A) \times X_2(C \vdash \pi' M, \pi' N \colon B),$$

as in the diagrams

$$\begin{tikzcd}
X_2(M, N) \arrow[r, "u"] \arrow[d, "\eta"'] & X_2(\pi M, \pi N) \times X_2(\pi' M, \pi' N)\\
\mathcal{L}(X)(M, N) \arrow[r, "\cong"'] & \mathcal{L}(X)(\pi M, \pi N) \times \mathcal{L}(X)(\pi' M, \pi' N) \arrow[u, "h \times h"']
\end{tikzcd}$$

and

$$\begin{tikzcd}
X_2(\pi M, \pi N) \times X_2(\pi' M, \pi' N) \arrow[r, "v"] \arrow[d, "\eta \times \eta"'] & X_2(M, N)\\
\mathcal{L}(X)(\pi M, \pi N) \times \mathcal{L}(X)(\pi' M, \pi' N) \arrow[r, "\cong"'] & \mathcal{L}(X)(M, N). \arrow[u, "h"']
\end{tikzcd}$$

Starting from $r \in X_2(M, N)$, we obtain

$$v(u(r)) = h(\eta(h(\pi(\eta(r)))), \eta(h(\pi'(\eta(r))))).$$

But consider $(\eta(\pi\eta(r)), \eta(\pi'\eta(r)))$ in $\mathcal{L}(\mathcal{L}X)$; its images by $h \circ \mathcal{L}h$ and $h \circ \mu$ are respectively:

- $h(\eta(h(\pi(\eta r))), \eta(h(\pi'(\eta r))))$, and
- $h(\pi\eta(r), \pi'\eta(r))$, i.e., $h(\eta(r))$, i.e., r,

which must be equal because h is an $\mathcal{L}$-algebra, hence $v \circ u = id$.

Conversely, starting from $(r, s) \in X_2(M_1, M_2) \times X_2(N_1, N_2)$, we obtain the pair with components

$$h(\pi(\eta(h(\eta(r), \eta(s))))) \qquad \text{and} \qquad h(\pi'(\eta(h(\eta(r), \eta(s))))).$$

Considering $\pi(\eta(\eta(r), \eta(s))) \in \mathcal{L}(\mathcal{L}(X))$, its images by $h \circ \mathcal{L}(h)$ and $h \circ \mu$ are respectively:

- $h(\pi(\eta(h(\eta(r), \eta(s)))))$, and
- $h(\pi(\eta(r), \eta(s))) = h(\eta(r)) = r$.

As above, they must be equal, and by symmetry the second component is s, and we have proved $u \circ v = id$. Similar reasoning for the terminal object and internal homs leads to:

Proposition 10. *This yields a cartesian closed 2-category structure on $\mathcal{C}$.*

This extends to morphisms of $\mathcal{L}$-algebras, so we have constructed a functor $\mathcal{F}\colon \mathcal{L}\text{-Alg} \to \mathsf{2CCCat}$.

7.3. **Adjunction.** Consider any $\mathcal{L}$-algebra (X, h). What does $(Y, k) = \mathcal{V}(\mathcal{F}(X, h))$ look like? Sorts in Y_0 are types in $\mathcal{L}_0(X_0)$. Operations $Y_1(G \vdash A)$ are terms in $\mathcal{L}_1(X_0, X_1)(\prod G \vdash A)$. Reduction rules in $Y_2(G \vdash M, N\colon B)$ are reductions in $\mathcal{L}(X)(\prod G \vdash M', N'\colon B)$, where $M' = M[\pi_1 x/x_1, \ldots, \pi_n x/x_n]$ (and similarly for N').

Let η_X send:

- each sort $\iota \in X_0$ to the type $\iota \in \mathcal{L}_0(X_0)$,
- each operation $c \in X(G \vdash A)$ to the term $c(\pi_1 x, \ldots, \pi_n x)$, and
- each reduction rule $r \in X_2(G \vdash M, N\colon A)$ to the reduction $x\colon \prod G \vdash r(\pi_1 x, \ldots, \pi_n x) : M' \to N' : A$.

Theorem 2. *This η is a natural transformation which is the unit of an adjunction*

$$\begin{tikzcd}
\mathcal{L}\text{-Alg} \arrow[r, bend left, "\mathcal{F}"] \arrow[r, phantom, "\perp"] & \mathsf{2CCCat}. \arrow[l, bend left, "\mathcal{V}"]
\end{tikzcd}$$

Proof. Consider any morphism $f\colon (X, h) \to \mathcal{V}(\mathcal{C})$, and let $(Y, k) = \mathcal{V}(\mathcal{F}(X, h))$ and $\mathcal{V}(\mathcal{C}) = (\mathcal{C}_0, \mathcal{C}_1, h_2\colon \mathcal{C}_2 \to \mathcal{C}_1)$. We now define a uniquely determined cartesian closed functor $f'\colon \mathcal{F}(X, h) \to \mathcal{C}$ making the triangle

$$\begin{tikzcd}
X \arrow[r, "\eta_X"] \arrow[dr, "f"'] & \mathcal{V}(\mathcal{F}(X)) \arrow[d, "\mathcal{V}(f')"] \\
& \mathcal{V}(\mathcal{C})
\end{tikzcd}$$

commute.

On objects, it is determined by induction: on sorts by f_0, and on type constructors by the requirement that f' be cartesian closed. On morphisms, it is similarly determined by f_1 and f' being cartesian closed. On 2-cells, define f' to be $f_2 \colon X_2(A \vdash M, N \colon B) \to \mathcal{C}(f'(A), f'(B))(f'(M), f'(N))$, which is also the only possible choice from f.

We thus only have to show that f' is cartesian closed, which follows by f being a morphism of $\mathcal{L}$-algebras. For example, to show that binary products of reductions are preserved, consider $r \in X_2(C \vdash M_1, M_2 \colon A)$ and $s \in X_2(C \vdash N_1, N_2 \colon B)$. Their product in $\mathcal{F}(X)$ is obtained by considering the atomic reductions $x \colon C \vdash r(x) : M_1 \to M_2 \colon A$ and $x \colon C \vdash s(x) : N_1 \to N_2 \colon B$ and taking $h(r(x), s(x))$, which is sent by f_2 to $f_2(h(r(x), s(x)))$. But, because f is a morphism of $\mathcal{L}$-algebras, this is the same as $h_2((f_2(r))(x), (f_2(s))(y))$, which is by definition (i.e., Figure 2) the product $(f_2(r), f_2(s))$ in $\mathcal{C}$. □

References

[1] H. J. Sander Bruggink. *Equivalence of reductions in higher-order rewriting*. PhD thesis, Utrecht University, 2008.

[2] Paolo Capriotti. Concurrent semantics with variable binding. Master's thesis, University of Pisa, 2009.

[3] Andrea Corradini, Fabio Gadducci, and Ugo Montanari. Relating two categorial models of term rewriting. In Jieh Hsiang, editor, *RTA*, volume 914 of *Lecture Notes in Computer Science*, pages 225–240. Springer, 1995.

[4] Thierry Despeyroux and André Hirschowitz. Principles for functional abstract syntax. Draft, 1995.

[5] Uffe Engberg and Mogens Nielsen. A calculus of communicating systems with label passing. Technical Report PB-208, Aarhus University, 1986.

[6] Marcelo P. Fiore. Second-order and dependently-sorted abstract syntax. In *LICS*, pages 57–68. IEEE Computer Society, 2008.

[7] Barney P. Hilken. Towards a proof theory of rewriting: The simply typed 2λ-calculus. *Theor. Comput. Sci.*, 170(1-2):407–444, 1996.

[8] André Hirschowitz and Marco Maggesi. Modules over monads and linearity. In *WoLLIC '07*, volume 4576 of *LNCS*. Springer, 2007.

[9] Yoshiki Kinoshita, John Power, and Makoto Takeyama. Sketches. *Journal of Pure and Applied Algebra*, 143(1-3), 1999.

[10] Jan W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, Amsterdam, 1980.

[11] Joachim Lambek. From λ-calculus to cartesian closed categories. Academic Press, 1980.

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

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I/II. *Information and Computation*, 100(1):1–77, 1992.

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

[15] Vincent van Oostrom and Femke van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In Jan Heering, Karl Meinke, Bernhard Möller, and Tobias Nipkow, editors, *HOA*, volume 816 of *Lecture Notes in Computer Science*, pages 276–304. Springer, 1993.

[16] Charles Wells. A generalization of the concept of sketch. *Theor. Comput. Sci.*, 70(1):159–178, 1990.

[17] David A. Wolfram. *The Clausal Theory of Types*. Number 21 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[18] Julianna Zsidó. *Typed Abstract Syntax*. PhD thesis, Université de Nice-Sophia Antipolis, 2010.

Appendix A. Equations on reductions

Current address: CNRS, Université de Savoie

Congruence

$$\frac{\Gamma \vdash P : M \to N \colon A}{\Gamma \vdash P \equiv P : M \to N \colon A} \qquad \frac{\Gamma \vdash P \equiv Q : M \to N \colon A}{\Gamma \vdash Q \equiv P : M \to N \colon A}$$

$$\frac{\Gamma \vdash P_1 \equiv P_2 : M \to N \colon A \qquad \Gamma \vdash P_2 \equiv P_3 : M \to N \colon A}{\Gamma \vdash P_1 \equiv P_3 : M \to N \colon A}$$

$$\frac{\Gamma \vdash P \equiv P' : M_1 \to M_2 \colon A \qquad \Gamma \vdash Q \equiv Q' : M_2 \to M_3 \colon A}{\Gamma \vdash (P \,;_{M_2} Q) \equiv (P' \,;_{M_2} Q') : M_1 \to M_3 \colon A}$$

$$\frac{\begin{array}{c}(r \in X(G \vdash M, N \colon A))\\ \Gamma \vdash P_1 \equiv Q_1 : M_1 \to N_1 \colon G_1 \qquad \ldots \qquad \Gamma \vdash P_n \equiv Q_n : M_n \to N_n \colon G_n\end{array}}{\Gamma \vdash r(P_1, \ldots, P_n) \equiv r(Q_1, \ldots, Q_n) : M[M_1, \ldots, M_n] \to N[N_1, \ldots, N_n] \colon A}$$

$$\frac{\begin{array}{c}(c \in X(G \vdash A))\\ \Gamma \vdash P_1 \equiv Q_1 : M_1 \to N_1 \colon G_1 \qquad \ldots \qquad \Gamma \vdash P_n \equiv Q_n : M_n \to N_n \colon G_n\end{array}}{\Gamma \vdash c(P_1, \ldots, P_n) \equiv c(Q_1, \ldots, Q_n) : c(M_1, \ldots, M_n) \to c(N_1, \ldots, N_n) \colon A}$$

$$\frac{\Gamma, x \colon A \vdash P \equiv Q : M \to N \colon B}{\Gamma \vdash (\lambda x \colon A.P) \equiv (\lambda x \colon A.Q) : \lambda x \colon A.M \to \lambda x \colon A.N \colon B^A}$$

$$\frac{\Gamma \vdash P \equiv P' : M \to M' \colon B^A \qquad \Gamma \vdash Q \equiv Q' : N \to N' \colon A}{\Gamma \vdash (PQ) \equiv (P'Q') : MN \to M'N' \colon B}$$

$$\frac{\Gamma \vdash P \equiv P' : M \to M' \colon A \qquad \Gamma \vdash Q \equiv Q' : N \to N' \colon B}{\Gamma \vdash (P, Q) \equiv (P', Q') : (M, N) \to (M', N') \colon A \times B}$$

$$\frac{\Gamma \vdash P \equiv Q : M \to N \colon A \times B}{\Gamma \vdash (\pi_{A,B} P) \equiv (\pi_{A,B} Q) : \pi_{A,B} M \to \pi_{A,B} N \colon A}$$

$$\frac{\Gamma \vdash P \equiv Q : M \to N \colon A \times B}{\Gamma \vdash (\pi'_{A,B} P) \equiv (\pi'_{A,B} Q) : \pi'_{A,B} M \to \pi'_{A,B} N \colon A}$$

Category

$$\frac{\Gamma \vdash P_1 : M_1 \to M_2 \colon A \qquad \Gamma \vdash P_2 : M_2 \to M_3 \colon A \qquad \Gamma \vdash P_3 : M_3 \to M_4 \colon A}{\Gamma \vdash (P_1 \,;_{M_2} (P_2 \,;_{M_3} P_3)) \equiv ((P_1 \,;_{M_2} P_2) \,;_{M_3} P_3) : M_1 \to M_4 \colon A}$$

$$\frac{\Gamma \vdash P : M \to N \colon A}{\Gamma \vdash (P \,;_N N) \equiv P : M \to N \colon A} \qquad \frac{\Gamma \vdash P : M \to N \colon A}{\Gamma \vdash (M \,;_M P) \equiv P : M \to N \colon A}$$

Figure 3. Equations on reductions (Congruence and category)

Beta and eta

$$\dfrac{\Gamma, x\colon A \vdash P : M \to M'\colon B \qquad \Gamma \vdash Q : N \to N'\colon A}{\Gamma \vdash ((\lambda x\colon A.P)Q) \equiv P[Q/x] : (\lambda x\colon A.M)N \to M'[N'/x]\colon B}$$

$$\dfrac{\Gamma \vdash P \equiv M \to N\colon B^A}{\Gamma \vdash P \equiv \lambda x\colon A.(Px) : M \to N\colon B^A}\ (x \notin \Gamma)$$

$$\dfrac{\Gamma \vdash P : M_1 \to M_2\colon A \qquad \Gamma \vdash Q : N_1 \to N_2\colon B}{\Gamma \vdash \pi(P,Q) \equiv P : \pi(M_1,N_1) \to M_2\colon A}$$

$$\dfrac{\Gamma \vdash P : M_1 \to M_2\colon A \qquad \Gamma \vdash Q : N_1 \to N_2\colon B}{\Gamma \vdash \pi'(P,Q) \equiv P : \pi'(M_1,N_1) \to N_2\colon A}$$

$$\dfrac{\Gamma \vdash P : (M_1,N_1) \to (M_2,N_2)\colon A \times B}{\Gamma \vdash P \equiv (\pi P, \pi' P) : (M_1,N_1) \to (M_2,N_2)\colon A \times B} \qquad \dfrac{\Gamma \vdash P : M \to N\colon 1}{\Gamma \vdash P \equiv () : M \to N\colon 1}$$

Lifting

$$\dfrac{(r \in X(\Gamma \vdash \langle M_1, M_2\rangle\colon A)) \qquad \Delta \vdash P : N_1 \to N_2\colon \Gamma \qquad \Delta \vdash Q : N_2 \to N_3\colon \Gamma}{\Gamma \vdash r(P \,;_{N_2} Q) \equiv M_1[P] \,;_{M_1[N_2]} r(Q) : M_1[N_1] \to M_2[N_3]\colon A}$$

$$\dfrac{(r \in X(\Gamma \vdash \langle M_1, M_2\rangle\colon A)) \qquad \Delta \vdash P : N_1 \to N_2\colon \Gamma \qquad \Delta \vdash Q : N_2 \to N_3\colon \Gamma}{\Gamma \vdash r(P \,;_{N_2} Q) \equiv r(P) \,;_{M_2[N_2]} M_2[Q] : M_1[N_1] \to M_2[N_3]\colon A}$$

$$\dfrac{\Gamma \vdash P : M_1 \to M_2\colon G \qquad \Gamma \vdash Q : M_2 \to M_3\colon G}{\Gamma \vdash (c(P \,;_{M_2} Q)) \equiv (c(P) \,;_{c(M_2)} c(Q)) : M_1 \to M_3\colon A}\ (c \in X(G \vdash A))$$

$$\dfrac{\Gamma, x\colon A \vdash P : M_1 \to M_2\colon B \qquad \Gamma, x\colon A \vdash Q : M_2 \to M_3\colon B}{\begin{array}{l}\Gamma \vdash (\lambda x\colon A.(P \,;_{M_2} Q)) \equiv ((\lambda x\colon A.P) \,;_{\lambda x\colon A.M_2} (\lambda x\colon A.Q)) \\ \hfill : \lambda x\colon A.M_1 \to \lambda x\colon A.M_3\colon B^A\end{array}}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash P : M_1 \to M_2\colon B^A \\ \Gamma \vdash P' : M_2 \to M_3\colon B^A \qquad \Gamma \vdash Q : N_1 \to N_2\colon A \qquad \Gamma \vdash Q' : N_2 \to N_3\colon A\end{array}}{\Gamma \vdash ((P \,;_{M_2} P')(Q \,;_{N_2} Q')) \equiv ((PQ) \,;_{M_2N_2} (P'Q')) : M_1N_1 \to M_3N_3\colon B}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash P : M_1 \to M_2\colon A \\ \Gamma \vdash P' : M_2 \to M_3\colon A \qquad \Gamma \vdash Q : N_1 \to N_2\colon B \qquad \Gamma \vdash Q' : N_2 \to N_3\colon B\end{array}}{\begin{array}{l}\Gamma \vdash ((P \,;_{M_2} P'), (Q \,;_{N_2} Q')) \equiv ((P,Q) \,;_{(M_2,N_2)} (P',Q')) \\ \hfill : (M_1,N_1) \to (M_3,N_3)\colon A \times B\end{array}}$$

$$\dfrac{\Gamma \vdash P : M_1 \to M_2\colon A \times B \qquad \Gamma \vdash Q : M_2 \to M_3\colon A \times B}{\Gamma \vdash (\pi_{A,B}(P \,;_{M_2} Q)) \equiv (\pi_{A,B}P \,;_{\pi_{A,B}M_2} \pi_{A,B}Q) : M_1 \to M_3\colon A}$$

$$\dfrac{\Gamma \vdash P : M_1 \to M_2\colon A \times B \qquad \Gamma \vdash Q : M_2 \to M_3\colon A \times B}{\Gamma \vdash (\pi'_{A,B}(P \,;_{M_2} Q)) \equiv (\pi'_{A,B}P \,;_{\pi'_{A,B}M_2} \pi'_{A,B}Q) : M_1 \to M_3\colon B}$$

FIGURE 4. Equations on reductions (beta-eta and lifting)

Kleene Algebra with Tests and Coq Tools for While Programs

Damien Pous

CNRS – LIP, ENS Lyon, UMR 5668

Abstract. We present a Coq library about Kleene algebra with tests, including a proof of their completeness over the appropriate notion of languages, a decision procedure for their equational theory, and tools for exploiting hypotheses of a certain kind in such a theory.
Kleene algebra with tests make it possible to represent if-then-else statements and while loops in most imperative programming languages. They were actually introduced by Kozen as an alternative to propositional Hoare logic.
We show how to exploit the corresponding Coq tools in the context of program verification by proving equivalences of while programs, correctness of some standard compiler optimisations, Hoare rules for partial correctness, and a particularly challenging equivalence of flowchart schemes.

Introduction

Kleene algebra with tests (KAT) have been introduced by Kozen [19], as an equational system for program verification. A Kleene algebra with tests is a Kleene algebra (KA) with an embedded Boolean algebra of tests. The Kleene algebra component deals with the control-flow graph of the programs—sequential composition, iteration, and branching—while the Boolean algebra component deals with the conditions appearing in if-then-else statements, while loops, or pre- and post-assertions.

This formalism is both concise and expressive, which allowed Kozen and others to give detailed paper proofs about various problems in program verification (see, e.g., [3, 19, 21, 23]). More importantly, the equational theory of KAT is decidable and complete over relational models [24], and hypotheses of a certain kind can moreover be eliminated [11, 15]. This suggests that a proof using KAT should not be done manually, but with the help of a computer. The goal of the present work is to give this possibility, inside the Coq proof assistant.

The underlying decision procedure cannot be formulated, a priori, as a simple rewriting system: it involves automata algorithms, it cannot be defined in Ltac, at the meta-level, and it does not produce a certificate which could easily be checked in Coq, a posteriori. This leaves us with only one possibility: defining a reflexive tactic [1, 8, 14]. Doing so is quite challenging: we basically have to prove completeness of KAT axioms w.r.t. the model of guarded string languages (the natural generalisation of languages for KA, to KAT), and to provide a provably correct algorithm for language equivalence of KAT expressions.

The completeness theorem is far from trivial; we actually have to formalise a lot of preliminary material: finite sums, finite sets, unique decomposition of Boolean expressions into sums of atoms, regular expression derivatives, expansion theorem for regular expressions, matrices, automata... As a consequence, we only give here a high-level overview of the involved mathematics, leaving aside standard definitions, technical details, or secondary formalisation tricks. The interested reader can consult the library, which is documented [30].

Outline. We first present KAT and its models (§1). We then sketch the completeness proof (§2), the decision procedure (§3), and the method used to eliminate hypotheses (§4). We finally illustrate the benefits of our tactics on several case-studies (§5), before discussing related works (§6), and concluding (§7).

1 Kleene Algebra with Tests

A Kleene algebra with tests consists of:

- a Kleene algebra $\langle X, \cdot, +, \cdot^\star, 1, 0\rangle$ [18], i.e., an idempotent semiring with a unary operation, called "Kleene star", satisfying an axiom: $1 + x \cdot x^\star \leq x^\star$ and two inference rules: $y \cdot x \leq x$ entails $y^\star \cdot x \leq x$ and the symmetric one. (The preorder ($\leq$) being defined by $x \leq y \triangleq x + y = y$.)
- a Boolean algebra $\langle B, \wedge, \vee, \neg, \top, \bot\rangle$;
- a homomorphism from $\langle B, \wedge, \vee, \top, \bot\rangle$ to $\langle X, \cdot, +, 1, 0\rangle$, that is, a function $[\cdot] : B \to X$ such that $[a \wedge b] = [a] \cdot [b]$, $[a \vee b] = [a] + [b]$, $[\top] = 1$, and $[\bot] = 0$.

The elements of the set B are called "tests"; we denote them by a, b. The elements of X, called "Kleene elements", are denoted by x, y, z. We usually omit the operator "$\cdot$" from expressions, writing xy for $x \cdot y$. The following (in)equations illustrate the kind of laws that hold in all Kleene algebra with tests:

$$[a \vee \neg a] = 1 \qquad [a \wedge (\neg a \vee b)] = [a][b] = [\neg(\neg a \vee \neg b)]$$

$$x^\star x^\star = x^\star \qquad (x+y)^\star = x^\star(yx^\star)^\star \qquad (x + xxy)^\star \leq (x + xy)^\star$$

$$[a]([\neg a]x)^\star = [a] \qquad [a]([a]x[\neg a] + [\neg a]y[a])^\star[a] \leq (xy)^\star$$

The laws from the first line come from the Boolean algebra structure, while the ones from the second line come from the Kleene algebra structure. The two laws from the last line are more interesting: their proof must mix both Boolean algebra and Kleene algebra reasoning. They are left to the reader as a non-trivial exercice; the tools we present in this paper allow one to prove them automatically.

1.1 The model of binary relations

Binary relations form a Kleene algebra with tests; this is the main model we are interested in, in practice. The Kleene elements are the binary relations over a given set S, the tests are the predicates over this set, and the star of a relation is its reflexive transitive closure:

$$\begin{aligned}
X &= \mathcal{P}(S \times S) & B &= \mathcal{P}(S) \\
x \cdot y &= \{(p,q) \mid \exists r, (p,r) \in x \wedge (r,q) \in y\} & a \wedge b &= a \cap b \\
x + y &= x \cup y & a \vee b &= a \cup b \\
x^\star &= \{(p_0, p_n) \mid \exists p_1 \dots p_{n-1}, \forall i < n, (p_i, p_{i+1}) \in x\} & \neg a &= S \setminus a \\
1 &= \{(p,p) \mid p \in S\} & \top &= S \\
0 &= \emptyset \qquad [a] = \{(p,p) \mid p \in a\} & \bot &= \emptyset
\end{aligned}$$

The laws of a Kleene algebra are easily proved for these operations; note however that one needs either to restrict to decidable predicates (i.e., to take `S → bool` or `{p: S → Prop | forall p, S p ∨¬S p}` for B), or to assume the law of excluded middle: B must be a Boolean algebra, so that negation has to be an involution. This choice for B is left to the user of the library.

This relational model is typically used to interpret imperative programs: such programs are state transformers, i.e., binary relations between states, and the conditions appearing in these programs are just predicates on states. These conditions are usually decidable, so that the above constraint is actually natural.

The equational theory of Kleene algebra with tests is complete over the relational model [24]: any equation $x = y$ that holds universally in this model can be proved from the axioms of KAT. We do not need to formalise this theorem, but it is quite informative in practice: by contrapositive, if an equation cannot be proved from KAT, then it cannot be universally true on binary relations, meaning that proving its validity for a particular instantiation of the variables necessarily requires one to exploit additional properties of this particular instance.

1.2 Other models

We describe two other models in the sequel: the syntactic model (§1.3) and the model of guarded string languages (§1.4); these models have to be formalised to build the reflexive tactic we aim at.

There are other important models of KAT. First of all, any Kleene algebra can be extended into a Kleene algebra with tests by embedding the two-element Boolean lattice. We also have traces models (where one keeps track of the whole execution traces of the programs rather than just their starting and ending points), matrices over a Kleene algebra with tests, but also models inherited from semirings like min-plus and max-plus algebra. The latter models have a degenerate Kleene star operation; they become useful when one constructs matrices over them, for instance to study shortest path algorithms.

Also note that like for Kleene algebra [9, 20, 29], KAT admits a natural “typed” generalisation, allowing for instance to encompass heterogeneous binary relations and rectangular matrices. Our Coq library is actually based on this generalisation, and this deeply impacts the whole infrastructure; we however omit the corresponding details and technicalities here, for the sake of clarity.

1.3 KAT expressions

Let p, q range over a set Σ of *letters* (or *actions*), and let $a_1, \dots, a_n$ be the elements of a finite set Θ of *primitive tests*. *Boolean expressions* and *KAT expressions* are defined by the following syntax:

$$a, b ::= a_i \in \Theta \mid a \wedge a \mid a \vee a \mid \neg a \mid \top \mid \bot \qquad \text{(Boolean expressions)}$$

$$x, y ::= p \in \Sigma \mid [a] \mid x \cdot y \mid x + y \mid x^\star \mid 1 \mid 0 \ . \qquad \text{(KAT expressions)}$$

Given a Kleene algebra with tests $\mathcal{K} = \langle X, B, [\cdot]\rangle$, any pair of maps $\theta : \Theta \rightarrow B$ and $\sigma : \Sigma \rightarrow X$ gives rise to a KAT homomorphism allowing to interpret expressions in $\mathcal{K}$. Given two such expressions x and y, the equation $x = y$ is a *KAT theorem*, written $\text{KAT} \vdash x = y$, when the equation holds in any Kleene algebra with tests, under any interpretation. One checks easily that KAT expressions quotiented by the latter relation form a Kleene algebra with tests; this is the free Kleene algebra with tests over Σ and Θ. (We actually use this impredicative encoding of KAT derivability in the Coq library.)

1.4 Guarded strings languages

Guarded string languages are the natural generalisation of string languages for Kleene algebra with tests. We briefly define them.

An *atom* is a function from elementary tests (Θ) to Booleans; it indicates which of these tests are satisfied. We let α, β range over atoms, the set of which is denoted by At. (Technically, we represent elementary tests as finite ordinals of a given size n ($\Theta = \texttt{ord}\ n$), and we encode atoms as ordinals ($\texttt{At} = \texttt{ord}\ 2^n$). This allows us to avoid functional extensionality problems.) We let u, v range over *guarded strings*: alternating sequences of atoms and letters, which both start and end with an atom:

$$\alpha_1, p_1, \dots, \alpha_n, p_n, \alpha_{n+1} \ .$$

The concatenation $u * v$ of two guarded strings u, v is a partial operation: it is defined only if the last atom of u is equal to the first atom of v; it consists in concatenating the two sequences and removing the shared atom in the middle.

The Kleene algebra with tests of guarded string languages is obtained by considering sets of guarded strings for X and sets of atoms for B:

$$\begin{array}{rlr}
X = \mathcal{P}\left((At \times \Sigma)^\star \times At\right) & & B = \mathcal{P}\left(At\right) \\
x \cdot y = \{u * v \mid u \in x \wedge v \in y\} & & a \wedge b = a \cap b \\
x + y = x \cup y & & a \vee b = a \cup b \\
x^\star = \{u_1 * \cdots * u_n \mid \exists u_1 \dots u_n, \forall i \leq n, u_i \in x\} & & \neg a = At \setminus a \\
1 = \{\alpha \mid \alpha \in At\} & & \top = At \\
0 = \emptyset & [a] = \{\alpha \mid \alpha \in a\} & \bot = \emptyset
\end{array}$$

Note that we slightly abuse notation by letting α denote either an atom, or a guarded string reduced to an atom. Also note that the set $B = \mathcal{P}\left(At\right)$ has to be represented by the Coq type $\texttt{At} \rightarrow \texttt{bool}$, to get a Boolean algebra on it.

2 Completeness

Let G be the unique homomorphism from KAT expressions to guarded string languages such that

$$G(a_i) = \{\alpha \mid \alpha(a_i) \text{ is true}\} \qquad\qquad G(p) = \{\alpha p \beta \mid \alpha, \beta \in At\}$$

Completeness of KAT over guarded string languages can be stated as follows.

Theorem 1. *For all KAT expressions x, y, $G(x) = G(y)$ entails* $\mathrm{KAT} \vdash x = y$.

This theorem is central to our development: it allows us to prove (in)equations in arbitrary models of KAT, by resorting to an algorithm deciding guarded string language equivalence (to be described in §3).

We closely follow Kozen and Smith' proof [24]. This proof relies on the completeness of Kleene algebra over languages, which we thus need to prove first.

2.1 Completeness of Kleene algebra axioms

Let R be the Kleene algebra homomorphism from regular expressions to (plain) string languages mapping a letter p to the language consisting of the single-letter word p. KA completeness over languages can be stated as follows [18]:

Theorem 2. *For all regular expressions x, y, $R(x) = R(y)$ entails* $\mathrm{KA} \vdash x = y$.

(Like for KAT, the judgement $\mathrm{KA} \vdash x = y$ means that $x = y$ holds in any Kleene algebra, under any interpretation.) We already presented a Coq formalisation of this theorem [9], but our development was over-complicated. We re-proved it from scratch here, following a simpler path which we now describe.

The main idea of Kozen's proof consists in replaying automata algorithms algebraically, using matrices to encode automata. The key insight that allowed us to considerably simplify the corresponding formalisation is that the algorithm used for this proof need not be the same as the one to be executed by the reflexive tactic we eventually define. Indeed, we can take the simplest possible algorithm to prove KA completeness, ignoring all complexity aspects, thus allowing us to focus on conciseness and mathematical simplicity. In contrast, the algorithm to be executed by the final reflexive tactic should be relatively efficient, but we do not need to prove it complete, nor to replay its correctness algebraically: we only need to prove its correctness w.r.t. languages, which is much easier.

A preliminary step for the proof consists in proving that matrices over a Kleene algebra form a Kleene algebra. The Kleene star for matrices is non-trivial to define and to prove correct, but this can be done with a reasonable amount of efforts once appropriate lemmas and tools for block matrices have been set up.

A finite automaton can then be represented using three matrices (u, M, v) over regular expressions, where u is a $(1, n)$-matrix, M is a (n, n)-matrix, and v is a $(n, 1)$-matrix, n being the number of states of the automaton. Such a "matricial automaton" can be evaluated into a regular expression by taking the product $u \cdot M^\star \cdot v$, which is a scalar. The various classes of automata can be recovered by imposing conditions on the coefficients of the three matrices. For instance, a non-deterministic finite automaton (NFA) is such that u and v are 01-vectors and the coefficients of M are sums of letters.

Given a regular expression x, we construct a deterministic finite automaton (DFA) (u, M, v) such that $\text{KA} \vdash x = uM^\star v$, as follows.

1. First construct a NFA with epsilon transitions (u'', M'', v''), such that $\text{KA} \vdash x = u''M''^\star v''$. This is easily done by induction on x, using Thompson construction [31] (which is compositional, unlike the construction we used in [9]).
2. Remove epsilon transitions to obtain a NFA (u', M', v') such that $\text{KA} \vdash u''M''^\star v'' = u'M'^\star v'$. We do it purely algebraically, in one line. In particular the transitive closure of epsilon transitions is computed using Kleene star on matrices. (Unlike in [9] we do not need a dedicated algorithm for this.)
3. Use the powerset construction to convert this NFA into a DFA (u, M, v) such that $\text{KA} \vdash u'M'^\star v' = uM^\star v$. Again, this is done algebraically, and we do not need to perform the standard 'accessible subsets' optimisation.

We can prove that for any DFA (u, M, v), $R(uM^\star v)$ is the language recognised by the DFA. Therefore, to obtain Theorem 2, it suffices to prove that if two DFA (u, M, v) and (s, N, t) recognise the same language, then $\text{KA} \vdash uM^\star v = sN^\star t$. For this last step, it suffices to exhibit a Boolean matrix that relates exactly those states of the two DFA that recognise the same language. We need for that an algorithm to check language equivalence of DFA states; we reduce the problem to DFA emptiness, and we perform a simple reachability analysis.

All in all, the KA completeness proof itself only requires us 124 lines of specifications, and 119 lines of proofs (according to `coqwc`).

2.2 Completeness of KAT axioms

To obtain KAT completeness (Theorem 1), Kozen and Smith [24] define a function $\widehat{\cdot}$ on KAT expressions that expands the expressions in such a way that we have $\text{KAT} \vdash x = y$ iff $\text{KA} \vdash \widehat{x} = \widehat{y}$. While this function can be thought as a reduction of KAT to KA, it cannot be used in practice: it produces expressions that are almost systematically exponentially larger than the given ones. It is however sufficient to establish completeness; as explained earlier, we defer actual computations to a completely different algorithm (§3).

More precisely, the function $\widehat{\cdot}$ is defined in such a way that we have:

$$\text{KAT} \vdash \widehat{x} = x \tag{i}$$
$$G(\widehat{x}) = R(\widehat{x}) \tag{ii}$$

We deduce KAT completeness as follows:

$$\begin{aligned}
& G(x) = G(y) && \\
\Leftrightarrow\ & G(\widehat{x}) = G(\widehat{y}) && (G \text{ is a KAT morphism, and (i)}) \\
\Leftrightarrow\ & R(\widehat{x}) = R(\widehat{y}) && (\text{by (ii)}) \\
\Rightarrow\ & \mathrm{KA} \vdash \widehat{x} = \widehat{y} && (\text{KA completeness}) \\
\Rightarrow\ & \mathrm{KAT} \vdash \widehat{x} = \widehat{y} && (\text{any KAT is a KA}) \\
\Leftrightarrow\ & \mathrm{KAT} \vdash x = y && (\text{by (i)})
\end{aligned}$$

(Note that the last equation entails the first one, so that all these statements are in fact equivalent.)

The function $\widehat{\cdot}$ is defined recursively over KAT expressions, using an intermediate datastructure: formal sums of *externally guarded terms* (i.e., either an atom, or a product of the form $\alpha x \beta$). The case of a starred expression $x^\star$ is quite involved: $\widehat{x^\star}$ is defined by an internal recursion on the length of the formal sum corresponding to $\widehat{x}$. The proof of the first equation (i) is not too difficult to formalise, using appropriate tools for finite sums (i.e., a simplified form of big operators [7], which we actually use a lot in the whole development). The second one (ii) is more cumbersome, notably because we must deal with the two implicit coercions appearing in its statement: formally, it has to be stated as follows:

$$i(G(\widehat{x})) = R(j(\widehat{x})) \ ,$$

where i takes a guarded string language and returns a finite word language on the alphabet $\Sigma \uplus \Theta \uplus \Theta$, and j takes a KAT expression and returns a regular expression over this extended alphabet, by pushing all negations to the leaves.

Apart from the properties of these coercion functions, the proof of (ii) mainly consists in rather technical arguments about regular and guarded string languages concatenation. All in all, once KA completeness has been proved, KAT completeness requires us 278 lines of specifications, and 360 lines of proofs.

3 Decision procedure

To check whether two expressions denote the same language of guarded strings, we use an algorithm based on a notion of *partial derivatives* for KAT expressions. Derivatives were introduced by Brzozowski [10] for regular expressions; they make it possible to define a deterministic automaton where the states of the automaton are the regular expressions themselves.

Derivatives can be extended to KAT expressions in a very natural way [22]: we first define a Boolean function ϵ_α, that indicates whether an expression accepts the single atom α; this function is then used to define the derivation function $\delta_{\alpha,p}$, that intuitively returns what remains of the given expression after reading the atom α and the letter p. These two functions make it possible to give a

$$\delta'_{\alpha,p}(x+y) = \delta'_{\alpha,p}(x) \cup \delta'_{\alpha,p}(y) \qquad \delta'_{\alpha,p}(q) = \begin{cases} \{1\} & \text{if } p = q \\ \emptyset & \text{otherwise} \end{cases}$$

$$\delta'_{\alpha,p}(xy) = \begin{cases} \delta'_{\alpha,p}(x)y \cup \delta'_{\alpha,p}(y) & \text{if } \epsilon_\alpha(x) \\ \delta'_{\alpha,p}(x)y & \text{otherwise} \end{cases} \qquad \delta'_{\alpha,p}([a]) = \emptyset$$

$$\delta'_{\alpha,p}(x^\star) = \delta'_{\alpha,p}(x)x^\star$$

Fig. 1. Partial derivatives for KAT expressions

coalgebraic characterisation of the function G, which underpins the correctness of the algorithm we sketch below:

$$G(x)(\alpha) = \epsilon_\alpha(x) \qquad\qquad G(x)(\alpha\, p\, u) = G(\delta_{\alpha,p}(x))(u) \ .$$

Like with standard regular expressions, the set of derivatives of a given KAT expression (i.e., the set of expressions that can be obtained by repeatedly deriving w.r.t. arbitrary atoms and letters) can be infinite. To recover finiteness, we switch to *partial* derivatives [4]. Their generalisation to KAT should be folklore; we define them in Fig. 1. We use the notation Xy to denote the set $\{xy \mid x \in X\}$ when X is a set of expressions and y is an expression. The partial derivation function $\delta'_{\alpha,p}$ returns a (finite) set of expressions rather than a single one; this corresponds to the fact that we build a non-deterministic automaton. Still abusing notations, by letting a set of expressions denote the sum of its elements, we prove that $\mathrm{KAT} \vdash \delta_{\alpha,p}(x) = \delta'_{\alpha,p}(x)$.

Now call *bisimulation* any relation R between sets of expressions such that whenever $X \mathrel{R} Y$, we have

- $\epsilon(X) = \epsilon(Y)$ and
- $\forall \alpha \in At, \forall p \in \Sigma,\ \delta'_{\alpha,p}(X) \mathrel{R} \delta'_{\alpha,p}(Y)$.

We show that if there is a bisimulation R such that $X \mathrel{R} Y$, then $G(X) = G(Y)$ (the converse also holds). This gives us an algorithm to decide language equivalence of two KAT expressions x, y: it suffices to try to construct a bisimulation that relates the singletons $\{x\}$ and $\{y\}$. This algorithm terminates because the set of partial derivatives reachable from a pair of expressions is finite (we do not need to formalise this fact since we just need the correctness of this algorithm).

There is a lot of room for optimisation in our implementation—for instance, we use unordered lists to represent binary relations. An important point in our design is that such optimisations can be introduced and proved correct independently from the completeness proof for KAT, which gives us much more flexibility than in our previous work on Kleene algebra [9].

3.1 Building a reflexive tactic

Using standard methodology [1, 8, 14], we finally pack the previous ingredients into a Coq reflexive tactic called `kat`, allowing us to close automatically any goal which belongs to the equational theory of KAT.

The tactic works on any model of KAT: those already declared in the library (relations, languages, matrices, traces), but also the ones declared by the user. The reification code is written in OCaml; it is quite complicated for at least two reasons: KAT is a two-sorted structure, and we actually deal with "typed" KAT, as explained in §1.2, which requires us to work with a dependently typed syntax.

For the sake of simplicity, the Coq algorithm we implemented for KAT does not produce a counter-example in case of failure. To be able to give such a counter-example to the user, we actually run an OCaml copy of the algorithm first (extracted from Coq, and modified by hand to produce counter-examples). This has two advantages: the tactic is faster in case of failure, and the counter-example—a guarded string—can be pretty-printed in a nicer way.

4 Eliminating hypotheses

The above `kat` tactic works for the equational theory of KAT, i.e., the (in)equations that hold in any model of KAT, under any interpretation. In particular, this tactic does not make use of any hypothesis which is specific to the model or to the interpretation. Some hypotheses can however be exploited [11,15]: those having one of the following shapes.

(i) $x = 0$;
(ii) $[a]x = x[b]$, $[a]x \leq x[b]$, or $x[b] \leq [a]x$;
(iii) $x \leq [a]x$ or $x \leq x[a]$
(iv) $a = b$ or $a \leq b$;
(v) $[a]p = [a]$ or $p[a] = [a]$, for atomic p ($p \in \Sigma$);

Equations of the first kind (i) are called "Hoare" equations, for reasons to become apparent in §5.2. They can be eliminated using the following implication:

$$\begin{cases} x + uzu = y + uzu \\ z = 0 \end{cases} \qquad \text{entails} \quad x = y \ . \tag{†}$$

This implication is valid for any term u, and the method is complete [15] when u is taken to be the universal KAT expression, $\Sigma^\star$. Intuitively, for this choice of u, uzu recognizes all guarded strings that contain a guarded string of z as a substring. Therefore, when checking that $x + uzu = y + uzu$ are language equivalent rather than $x = y$, we rule out all counter-examples to $x = y$ that contain a substring belonging to z: such counter-examples are irrelevant since z is known to be empty.

Equations of the shape (iii) and (iv) are actually special cases of those of the shape (ii), which are in turn equivalent to Hoare equations. For instance, we have $[a]x \leq x[b]$ iff $[a]x[\neg b] = 0$. Moreover, two hypotheses of shape (i) can be merged into a single one using the fact that $x = 0 \wedge y = 0$ iff $x + y = 0$. Therefore, we can aggregate all hypotheses of shape (i-iv) into a single one (of shape (i)), and use the above technique just once.

Hypotheses of shape (v) are handled differently, using the following equivalence:

$$[a]p = [a] \quad \text{iff} \quad p = [\neg a]p + [a] \ , \tag{$\ddagger$}$$

This equivalence allows us to substitute $[\neg a]p + [a]$ for p in the considered goal—whence the need for p to be atomic. Again, the method is complete [15], i.e.,

$$\text{KAT} \vdash ([a]p = [a] \Rightarrow x = y) \quad \text{iff} \quad \text{KAT} \vdash x\theta = y\theta \qquad (\theta = \{p \mapsto [\neg a]p + [a]\})$$

4.1 Automating elimination of hypotheses in Coq

The previous techniques to eliminate some hypotheses in KAT can be easily automated in Coq. We first prove once and for all the appropriate equivalences and implications (the tactic `kat` is useful for that). We then define some tactics in Ltac that collect hypotheses of shape (i-iv), put them into shape (i), and aggregate them into a single one which is finally used to update the goal according to (†). Separately, we define a tactic that rewrites in the goal using all hypotheses of shape (v), through (‡). Finally, we obtain a tactic called `hkat`, that just preprocesses the conclusion of the goal using all hypotheses of shape (i-v) and then calls the `kat` tactic. Note that the completeness of this method [15] is a meta-theorem; we do not need to formalise it.

5 Case studies

We now present some examples of Coq formalisations where one can take advantage of our library.

5.1 Bigstep semantics of 'while' programs

The bigstep semantics of 'while' programs is teached in almost any course on semantics and programming languages. Such programs can be embedded into KAT in a straightforward way [21], thus providing us with proper tools to reason about them. Let us formalise such a language in Coq.

Assume a type `state` of states, a type `loc` of memory locations, and an `update` function allowing to update the value of a memory location. Call *arithmetic expression* any function from states to natural numbers, and *Boolean expression* any function from states to Booleans (we use a partially shallow embedding). The 'while' programming language is defined by the inductive type below:

```
Variable loc, state: Set.
Variable update: loc → nat → state → state.

Definition expr := state → nat.
Definition test := state → bool.
```

```
Inductive prog :=
| skp
| aff (l: loc) (e: expr)
| seq (p q: prog)
| ite (b: test) (p q: prog)
| whl (b: test) (p: prog).
```

The bigstep semantics of such programs is given as a "state transformer", i.e., a binary relation between states. Following standard textbooks, one can define this semantics in Coq using an inductive predicate:

```
Inductive bstep: prog → rel state state :=
| s_skp: ∀ s, bstep skp s s
| s_aff: ∀ l e s, bstep (aff l e) s (update l (e s) s)
| s_seq: ∀ p q s s' s'', bstep p s s' → bstep q s' s'' → bstep (seq p q) s s''
| s_ite_ff: ∀ b p q s s', ¬ b s → bstep q s s' → bstep (ite b p q) s s'
| s_ite_tt: ∀ b p q s s', b s → bstep p s s' → bstep (ite b p q) s s'
| s_whl_ff: ∀ b p s, ¬ b s → bstep (whl b p) s s
| s_whl_tt: ∀ b p s s', b s → bstep (seq p (whl b p)) s s' → bstep (whl b p) s s'.
```

Alternatively, one can define this semantic through the relational model of KAT, by induction over the program structure:

```
Fixpoint bstep (p: prog): rel state state :=
  match p with
    | skp ⇒ 1
    | seq p q ⇒ bstep p·bstep q
    | aff l e ⇒ upd l e
    | ite b p q ⇒ [b]·bstep p + [¬b]·bstep q
    | whl b p ⇒ ([b]·bstep p)⋆·[¬b]
  end.
```

(Notations come for free since binary relations are already declared as a model of KAT in our library.) The 'skip' instruction is interpreted as the identity relation; sequential composition is interpreted by relational composition. Assignments are interpreted using an auxiliary function, defined as follows:

```
Definition upd l e: rel state state := fun s s' ⇒ s' = update l (e s) s.
```

For the 'if-then-else' statement, the Boolean expression `b` is a predicate on states, i.e., a test in our relational model of KAT; this test is used to guard both branches of the possible execution paths. Accordingly for the 'while' loop, we iterate the body of the loop guarded by the test, using Kleene star. We make sure one cannot exit the loop before the condition gets false by post-guarding the iteration with the negation of this test.

This alternative definition is easily proved equivalent to the previous one. Its relative conciseness makes it easier to read; more importantly, this definition allows us to exploit all theorems and tactics about KAT, for free. For instance, suppose that one wants to prove some program equivalences. First define program equivalence, through the bigstep semantics:

```
Notation "p ∼ q" := (bstep p == bstep q).
```

(The "==" symbol denotes equality in the considered KAT model; in this case, relational equality.) The following lemmas about unfolding loops and dead code elimination, can be proved automatically.

```
Lemma two_loops b p: whl b (whl b p) ∼ whl b p.
Proof. simpl. kat. Qed.
(* ([b]·(([b]·bstep p)⋆·[¬b]))⋆·[¬b] == ([b]·bstep p)⋆·[¬b] *)

Lemma fold_loop b p: whl b (p ; ite b p skp) ∼ whl b p.
Proof. simpl. kat. Qed.
(* ([b]·(bstep p·([b]·bstep p + [¬b]·1)))⋆·[¬b] == ([b]·bstep p)⋆·[¬b] *)
```

```
Lemma dead_code a b p q r:  whl (a ∨ b) p ; ite b q r  ∼  whl (a ∨ b) p ; r.
Proof. simpl. kat. Qed.
(* ([a ∨ b]·bstep p)⋆·[¬(a ∨ b)]·([b]·bstep q + [¬b]·bstep r)
                                          == ([a ∨ b]·bstep p)⋆·[¬(a ∨ b)]·bstep r *)
```

(The semicolon in program expressions is a notation for sequential composition; the comments below each proof show the intermediate goal where the `bstep` fixpoint has been simplified, thus revealing the underlying KAT equality.)

Of course, the `kat` tactic cannot prove arbitrary program equivalences: the theory of KAT only deals with the control-flow graph of the programs and with the Boolean expressions, not with the concrete meaning of assignments or arithmetic expressions. We can however mix automatic steps with manual ones. Consider for instance the following example, where we prove that an assignment can be delayed. Our tactics cannot solve it automatically since some reasoning about assignments is required; however, by asserting manually a simple fact (in this case, an equation of shape (ii)), the goal becomes provable by the `hkat` tactic.

```
Definition subst l e (b: test): test := fun s ⇒ b (update l (e s) s).
Lemma aff_ite l e b p q: (l←e; ite b p q)  ∼  (ite (subst l e b) (l←e; p) (l←e; q)).
Proof.
  simpl. (* upd l e·([b]·bstep p + [¬b]·bstep q) ==
         [subst l e b]·(upd l e·bstep p)·[¬subst l e b]·(upd l e·bstep q) *)
  assert (upd l e·[b]  == [subst l e b]·upd l e) by (cbv; firstorder; subst; eauto).
  hkat.
Qed.
```

5.2 Hoare logic for partial correctness

Hoare logic for partial correctness [16] is subsumed by KAT [21]. The key ingredient in Hoare logic is the notion of a "Hoare triple" $\{A\}\,p\,\{B\}$, where p is a program, and A, B are two formulas about the memory manipulated by the program, respectively called pre- and post-conditions. A Hoare triple $\{A\}\,p\,\{B\}$ is *valid* if whenever the program p starts in some state s satisfying A and terminates in a state s', then s' satisfies B. Such a statement can be translated into KAT as a simple equation:

$$[A]p[\neg B] = 0$$

Indeed, $[A]p[\neg B] = 0$ precisely means that there is no execution path along p that starts in A and ends in $\neg B$. Such equations are Hoare equations (they have the shape (i) from §4), so that they can be eliminated automatically. As a consequence, inference rules of Hoare logic can be proved automatically using the `hkat` tactic. For instance, for the 'while' rule, we get the following script:

```
Lemma rule_whl A b p: {A ∧ b} p {A} → {A} whl b p {A ∧ ¬b}.
Proof. simpl. hkat. Qed.
(* [A ∧ b]·bstep p·[¬A] == 0 → [A]·(([b]·bstep p)⋆·[¬b])·[¬(A ∧ ¬b)] == 0 *)
```

5.3 Compiler optimisations

Kozen and Patron [23] use KAT to verify a rather large range of standard compiler optimisations, by equational reasoning. Citing their abstract, they cover "*dead code elimination, common subexpression elimination, copy propagation, loop hoisting, induction variable elimination, instruction scheduling, algebraic simplification, loop unrolling, elimination of redundant instructions, array bounds check elimination, and introduction of sentinels*". They cannot use automation, so that the size of their proofs ranges from a few lines to half a page of KAT computations.

We formalised all those equational proofs using our library. Most of them can actually be solved instantaneously, by a simple call to the `hkat` tactic. For the few remaining ones, we gave three to four lines proofs, consisting of first rewriting using hypotheses that cannot be eliminated, and then a call to `hkat`.

The reason why `hkat` performs so well is that most assumptions allowing to optimise the code in these examples are of the shape (i-v). For instance, to state that an instruction p has no effect when $[a]$ is satisfied, we use an assumption $[a]p = [a]$. Similarly, to state that the execution of a program x systematically enforces $[a]$, we use an assumption $x = x[a]$. The assumptions that cannot be eliminated are typically those of the shape $pq = qp$: "the instructions p and q commute"; such assumptions have to be used manually.

5.4 Flowchart schemes

The last example we discuss here is due to Paterson, it consists in proving the equivalence of two flowchart schemes (i.e., goto programs—see Manna's book [26] for a complete description of this model). The two schemes are given in Appendix A; Manna proves their equivalence using several successive graph transormations. His proof is really high-level and informal; it is one page long, plus three additional pages to draw intermediate flowcharts schemes. Angus and Kozen [3] give a rather detailed equational proof in KAT, which is about six pages long. Using the `hkat` tactic together with some ad-hoc rewriting tools, we managed to formalise Angus and Kozen's proof in three rather sparse screens.

Like in Angus and Kozen's proof, we progressively modify the KAT expression corresponding to the first schema, to make it evolve towards the expression corresponding to the second schema. Our mechanised proof thus roughly consists in a sequence of transitivity steps closed by `hkat`, allowing us to perform some rewriting steps manually and to move to the next step. This is illustrated schematically by the code presented in Fig. 2.

Most of our transitivity steps (the y_i's) already appear in Angus and Kozen's proof; we can actually skip a lot of their steps, thanks to `hkat`. Some of these simplifications can be spectacular: for instance, they need one page to justify the passage between their expressions (24) and (27), while a simple call to `hkat` does the job; similarly for the page they need between their steps (38) and (43).

```
Lemma Paterson: x_1 == z.
Proof.
  transitivity y_1. hkat.       (* x_1 == y_1 *)
  a few rewriting steps transforming y_1 into x_2.
  transitivity y_2. hkat.       (* x_2 == y_2 *)
  a few rewriting steps transforming y_2 into x_3.
  (* ... *)
  transitivity y_19. hkat.      (* x_19 == y_19 *)
  a few rewriting steps transforming y_19 into x_20.
  hkat.                         (* x_20 == z *)
Qed.
```

Fig. 2. Squeleton for the proof of equivalence of Paterson's flowchart schems

6 Related works

Several formalisations of algorithms and results related to regular expressions and languages have been proposed since we released our Coq reflexive decision procedure for Kleene algebra [9]: partial derivatives for regular expressions [2], regular expression equivalence [6, 12, 25, 27], regular expression matching [17]. None of these works contains a formalised proof of completeness for Kleene algebra, so that they cannot be used to obtain a general tactic for KA (note however that Krauss and Nipkow [25] obtain an Isabelle/HOL tactic for binary relations using a nice trick to sidestep the completeness proof—but they cannot deal with other models of KA).

On the more algebraic side, Struth et al. [5, 13] showed how to formalise and use relation algebra and Kleene algebra in Isabelle/HOL; they exploit the automation tools provided by this assistant, but they do not try to define decision procedures specific to Kleene algebra, and they do not prove completeness.

To the best our knowledge, the only formalisation of KAT prior to the present work is due to Pereira and Moreira [28], in Coq. They state all axioms of KAT, derive some simple consequences of these axioms (e.g., Boolean disjunction distribute over conjunction, Kleene star is monotone), and use them to manually prove the inference rules of Hoare logic, as we did automatically in §5.2. They do not provide models, automation tools, or completeness proof.

7 Conclusion

We presented a rather exhaustive Coq formalisation of Kleene algebra with tests: axiomatisation, models, completeness proof, decision procedure, elimination of hypotheses. We then showed several use-cases for the corresponding library: proofs about while programs and Hoare logic, certification of standard compiler optimisations, and equivalence of flowchart schemes.

Most of the theoretical material is due to Kozen et al. [3, 15, 18–24], so that our contribution mostly lies in the Coq mechanisation of these ideas. The completeness proof was particularly challenging to formalise, and lots of aspects of this work could not be explained in this extended abstract: how to encode the algebraic hierarchy, how to work efficiently with finite sets and finite sums, how to exploit symmetry arguments, reflexive normalisation tactics, tactics about lattices, finite ordinals and encodings of set-theoretic constructs in ordinals. . .

The Coq library is available online [30]; it is documented and axiom-free; its overall structure is given in Appendix B. This library actually has a larger scope than what we presented here: our long-term goal is to formalise and automate other fragments of relation algebra (residuated structures, Kleene algebra with converse, allegories. . .), so that the library is designed to allow for such extensions. For instance normalisation tactics and an ad-hoc semi-decision procedures are already defined for algebraic structures beyond Kleene algebra and KAT.

According to `coqwc`, the library consists of 4377 lines of specifications and 3020 lines of proofs, that distribute as follows. Overall, this is slightly less than our previous library for KA [9] (5105+4315 lines), and we do much more: not only we handle KAT, but we also lay the ground for the mechanisation of other fragments of relation algebra, as explained above.

	specifications	proofs	comments
ordinals, comparisons, finite sets. . .	674	323	225
algebraic hierarchy	490	374	216
models (languages, relations, expressions. . .)	1279	461	404
linear algebra, matrices	534	418	163
completeness, decisions procedure, tactics	1400	1444	740

The resulting theorems and tactics allowed us to shorten significantly a number of paper proofs—those about Hoare logic, compiler optimisations, and flowchart schemes. Getting a way to guarantee that such proofs are correct is important: although mathematically simple, they tend to be hard to proofread (we invite the skeptical reader to check Angus and Kozen's paper proof of Paterson example [3]). Moreover, automation greatly helps when searching for such proofs: being able to get either a proof or a counter-example for any proposed equation is a big plus: it makes it much easier to progress in the overall proof.

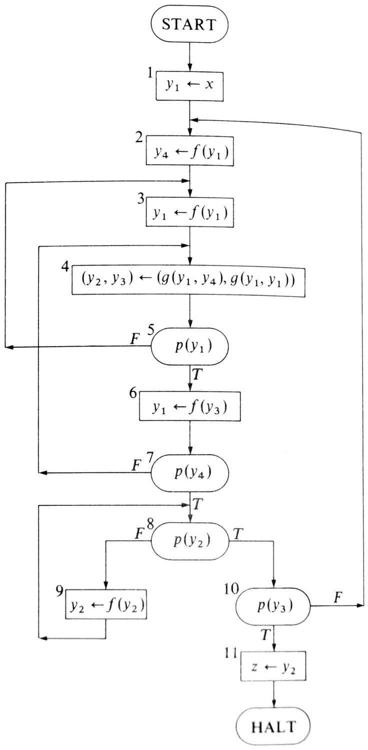
References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *Proc. LICS*, pages 95–105. IEEE Computer Society, 1990.
2. J. B. Almeida, N. Moreira, D. Pereira, and S. M. de Sousa. Partial derivative automata formalized in Coq. In *Proc. CIAA*, volume 6482 of *LNCS*, pages 59–68. Springer, 2010.
3. A. Angus and D. Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, CS Dpt, Cornell University, July 2001.
4. V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *TCS*, 155(2):291–319, 1996.
5. A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In *Proc. RAMiCS*, volume 7560 of *LNCS*, pages 66–81. Springer, 2012.

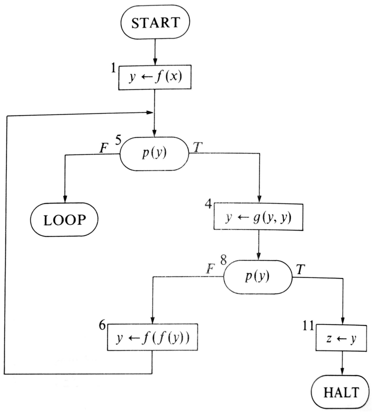
6. A. Asperti. A compact proof of decidability for regular expression equivalence. In *Proc. ITP*, volume 7406 of *LNCS*, pages 283–298. Springer, 2012.
7. Y. Bertot, G. Gonthier, S. O. Biha, and I. Pasca. Canonical big operators. In *TPHOLs*, volume 5170 of *LNCS*, pages 86–101. Springer, 2008.
8. R. Boyer and J. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*. NY: Academic Press, 1981.
9. T. Braibant and D. Pous. An efficient Coq tactic for deciding Kleene algebras. In *Proc. 1st ITP*, volume 6172 of *LNCS*, pages 163–178. Springer, 2010.
10. J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
11. E. Cohen. Hypotheses in Kleene algebra. Technical report, Bellcore, Morristown, N.J., 1994.
12. T. Coquand and V. Siles. A decision procedure for regular expression equivalence in type theory. In *Proc. CPP*, volume 7086 of *LNCS*. Springer, 2011.
13. S. Foster and G. Struth. Automated analysis of regular algebra. In *Proc. IJCAR*, volume 7364 of *LNCS*, pages 271–285. Springer, 2012.
14. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Proc. TPHOL*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.
15. C. Hardin and D. Kozen. On the elimination of hypotheses in Kleene algebra with tests. Technical Report TR2002-1879, CS Dpt, Cornell University, October 2002.
16. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
17. V. Komendantsky. Reflexive toolbox for regular expression matching: verification of functional programs in Coq+ssreflect. In *Proc. PLPV*, pages 61–70. ACM, 2012.
18. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. and Comp.*, 110(2):366–390, 1994.
19. D. Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.
20. D. Kozen. Typed Kleene algebra, 1998. TR98-1669, CS Dpt. Cornell University.
21. D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Log.*, 1(1):60–76, 2000.
22. D. Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report `http://hdl.handle.net/1813/10173`, CIS, Cornell University, March 2008.
23. D. Kozen and M.-C. Patron. Certification of compiler optimizations using Kleene algebra with tests. In *Proc. CL2000*, volume 1861 of *LNAI*, pages 568–582. Springer, 2000.
24. D. Kozen and F. Smith. Kleene algebra with tests: Completeness and decidability. In *Proc. CSL*, volume 1258 of *LNCS*, pages 244–259. Springer, September 1996.
25. A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *JAR*, 49(1):95–106, 2012.
26. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
27. N. Moreira, D. Pereira, and S. M. de Sousa. Deciding regular expressions (in-)equivalence in Coq. In *Proc. RAMiCS*, volume 7560 of *LNCS*, pages 98–113. Springer, 2012.
28. D. Pereira and N. Moreira. KAT and PHL in Coq. *Comput. Sci. Inf. Syst.*, 5(2):137–160, 2008.
29. 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, 2010.
30. D. Pous. RelationAlgebra: Coq library containing all material presented in this paper. `http://perso.ens-lyon.fr/damien.pous/ra`, December 2012.
31. K. Thompson. Regular expression search algorithm. *C. ACM*, 11:419–422, 1968.

A Paterson's flowchart schemes

Here are the two flowchart schemes we proved equivalent (§5.4), they appear in [26, pages 254 and 258].



Schema S_{6A}



Schema S_{6E}

Following Angus and Kozen's notations [3], these two schemes can be converted into the following KAT expressions:

$$
\begin{aligned}
S_{6A} = {} & x_1 p_{41} p_{11} q_{214} q_{311} \left([\neg a_1] p_{11} q_{214} q_{311}\right)^\star [a_1] p_{13} \\
& \big(([\neg a_4] + [a_4]([\neg a_2] p_{22})^\star [a_2 \wedge \neg a_3] p_{41} p_{11})\, q_{214} q_{311} \left([\neg a_1] p_{11} q_{214} q_{311}\right)^\star [a_1] p_{13}\big)^\star \\
& [a_4] \left([\neg a_2] p_{22}\right)^\star [a_2 \wedge a_3] z_2
\end{aligned}
$$

$$S_{6E} = s_1 [a_1] q_1 \left([\neg a_1] r_1 [a_1] q_1\right)^\star [a_1] z_1 \ ,$$

where the tests and actions are interpreted as follows:

$$
\begin{array}{lll}
x_i \triangleq y_i \leftarrow x & z_i \triangleq z \leftarrow y_i & a_i \triangleq P(y_i) \\
p_{ij} \triangleq y_i \leftarrow f(y_j) & q_{ijk} \triangleq y_i \leftarrow g(y_j, y_k) &
\end{array}
$$

(Note that we actually renamed the local variable y from schema S_{6E} into y_1, for the sake of uniformity.)

B Overall structure of the library

Here is a succinct description of each module from the library:

Utilities

- `common`: basic tactics and definitions used throughout the library
- `comparisons`: types with decidable equality and ternary comparison function
- `positives`: simple facts about binary positive numbers
- `ordinal`: finite ordinals, finite sets of finite ordinals
- `pair`: encoding pairs of ordinals as ordinals
- `powerfix`: simple pseudo-fixpoint iterator
- `lset`: sup-semilattice of finite sets represented as lists

Algebraic hierarchy

- `level`: bitmasks allowing us to refer to an arbitrary point in the hierarchy
- `lattice`: “flat” structures, from preorders to Boolean lattices
- `monoid`: typed structures, from po-monoids to residuated Kleene lattices
- `kat`: Kleene algebra with tests
- `kleene`: Basic facts about Kleene algebra
- `normalisation`: normalisation and semi-decision tactics for relation algebra

Models

- `prop`: distributive lattice of propositions
- `boolean`: Boolean trivial lattice, extended to a monoid.
- `rel`: heterogeneous binary relations
- `lang`: word languages
- `traces`: trace languages
- `atoms`: atoms of the free Boolean lattice over a finite set
- `glang`: guarded string languages
- `lsyntax`: free lattice (Boolean expressions)
- `syntax`: free relation algebra
- `regex`: regular expressions
- `gregex`: KAT expressions (typed—for KAT completeness)
- `ugregex`: untyped KAT expressions (untyped—for KAT decision procedure)

Untyping theorems

- `untyping`: untyping theorem for structures below KA with converse
- `kat_untyping`: untyping theorem for guarded string languages

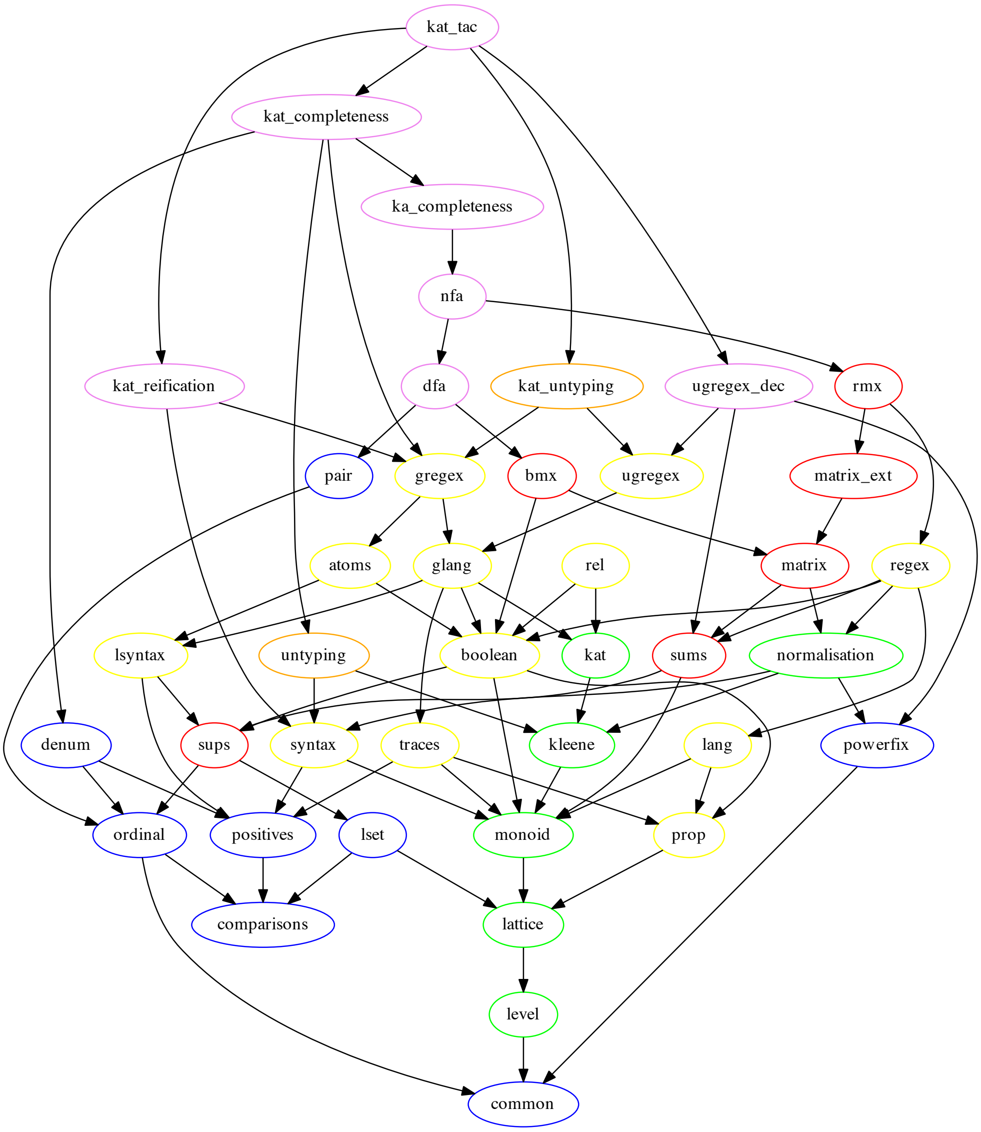
Linear algebra

- `sups`: finite suprema/infima (a la bigop, from ssreflect)
- `sums`: finite sums
- `matrix`: matrices over all structures supporting this construction
- `matrix_ext`: additional operations and properties about matrices
- `rmx`: matrices of regular expressions
- `bmx`: matrices of Booleans

Automata, completeness

- `dfa`: deterministic finite state automata, decidability of language inclusion
- `nfa`: matricial non-deterministic finite state automata
- `ugregex_dec`: decision of language equivalence for KAT expressions
- `ka_completeness`: (untyped) completeness of Kleene algebra
- `kat_completeness`: (typed) completeness of Kleene algebra with tests
- `kat_reification`: tools and definitions for KAT reification
- `kat_tac`: decision tactics for KA and KAT, elimination of hypotheses

Here are the dependencies between these modules:



Brzozowski's and up-to algorithms for must testing[⋆]

Filippo Bonchi[1], Georgiana Caltais[2], Damien Pous[1], Alexandra Silva[3⋆⋆]

[1] ENS Lyon, U. de Lyon, CNRS, INRIA, UCBL
[2] Reykjavik University
[3] Radboud University Nijmegen

Abstract. Checking language equivalence (or inclusion) of finite automata is a classical problem in Computer Science, which has recently received a renewed interest and found novel and more effective solutions, such as approaches based on antichains or bisimulations up-to. Several notions of equivalence (or preorder) have been proposed for the analysis of concurrent systems. Usually, the problem of checking these equivalences is reduced to checking bisimilarity. In this paper, we take a different approach and propose to adapt algorithms for language equivalence to check one prime equivalence in concurrency theory, must testing semantics. To achieve this transfer of technology from language to must semantics, we take a coalgebraic outlook at the problem.

1 Introduction

Determining whether two systems exhibit the same behavior under a given notion of equivalence is a recurring problem in different areas from Computer Science, from compiler analysis, to program verification, to concurrency theory. A widely accepted notion of equivalence is that two systems are equivalent if they behave the same when placed in the same context.

We will focus on the equivalence problem in the context of concurrency theory and process calculi. Systems are processes and contexts will be given by sets of tests a process should obey. This leads us to consider standard behavioural equivalences and preorders for process calculi, in particular *must testing* [14]: two systems are equivalent if they pass exactly the same tests, in all their executions.

The problem of automatically checking such testing equivalences is usually reduced to the problem of checking bisimilarity, as proposed in [12] and implemented in several tools [13, 10]. In a nutshell, equivalence is checked as follows. Two processes are considered, given by their labeled transition systems (LTS's). Then, the given LTS's are first transformed into "acceptance graphs", using a construction which is reminiscent of the *determinization* of non-deterministic automata (NDA). Finally, bisimilarity is checked via the *partition refinement*

[⋆] Appeared in Proc. APLAS, vol. 8301 of LNCS, pages 1-16, Springer, 2013

[⋆⋆] Also affiliated to Centrum Wiskunde & Informatica (Amsterdam, The Netherlands) and HASLab / INESC TEC, Universidade do Minho (Braga, Portugal).

algorithm [17, 22]. And one can answer the question of testing equivalence because gladly bisimilarity in acceptance graphs coincides with testing equivalence in the original LTS's.

The partition refinement algorithm, which is the best-known for minimizing LTS's w.r.t. bisimilarity, is analogous to Hopcroft's algorithm [16] for minimizing deterministic automatata (DA) w.r.t. language equivalence. In both cases, a partition of the state space is iteratively refined until a fixpoint is reached. Thus, the above procedure for checking testing semantics [12] is in essence the same as the classical procedure for checking language equivalence of NDA: first determinize and then compute a (largest) fixpoint.

In this work, we propose to transfer other algorithms for language equivalence, which are not available for bisimilarity, to the world of testing semantics. In order to achieve this, we take a coalgebraic perspective at the problem in hand, which allows us to study the constructions and the semantics in a uniform fashion. The abstract framework of *coalgebras* makes it possible to study different kinds of state based systems in a uniform way [26]. In particular, both the determinization of NDA's and the construction of acceptance graphs in [12] are instances of the generalized powerset construction [28, 20, 11]. This is the key observation of this work, which enables us to devise the presented algorithms.

First, we consider *Brzozowski's algorithm* [9] which transforms an NDA into the minimal deterministic automaton accepting the same language in a rather magical way: the input automaton is reversed (by swapping final and initial states and reversing its transitions), determinized, reversed and determinized once more. This somewhat intriguing algorithm can be explained in terms of duality and coalgebras [4, 2]. The coalgebraic outlook in [4] has several generalization of Brzozowski's algorithm to other types of transition systems, including Moore machines. This paves the way to adapt Brzozowski's algorithm for checking must semantics, which we will do in this paper.

Next, we consider several more efficient algorithms that have been recently introduced in a series of papers [32, 1, 7]. These algorithms rely on different kinds of *(bi)simulations up-to*, which are proof techniques originally proposed for process calculi [21, 27]. From these algorithms, we choose the one in [7] (HKC) which has been introduced by a subset of the authors and which, as we will show, can be adapted to must testing using a coalgebraic characterization of must equivalence, which we will also introduce.

Comparing these three families of algorithms (partition refinement [12], Brzozowski and bisimulations up-to) is not a simple task: both the problems of checking language and must equivalence are PSPACE-complete [17] but, in both cases, the theoretical complexity appears not to be problematic in practice, so that an empirical evaluation is more desirable. In [31, 29], experiments have shown that Brzozowski's algorithm performs better than Hopcroft for "high-density" NDA's, while Hopcroft is more efficient for generic NDA's. Both algorithms appear to be rather inefficient compared to those of the new generation [32, 1, 7]. It is out of the scope of this paper to present an experimental comparison of these algorithms and we confine our work to showing concrete examples where

HKC and Brzozowski's algorithm are exponentially more efficient than the other approaches.

Contributions. The main contributions of this work are:

- The coalgebraic treatment of must semantics (preorder and equivalence).
- The adaptation of HKC and Brzozowski's algorithm for must semantics. For the latter, this includes an optimization which avoids an expensive determinization step.
- The evidence that the coalgebraic analysis of systems yields not only a good mathematical theory of their semantics but also a rich playground to devise algorithms.
- An interactive applet allowing one to experiment with these algorithms [6].

The full version of this paper [5] contains further optimizations for the algorithms, their proofs of correctness, the formal connections with the work in [12] and the results of experiments checking the equivalence of an ideal and a distributed multiway synchronisation protocol [23].

Related Work. Another coalgebraic outlook on must is presented in [8] which introduces a fully abstract semantics for CSP. The main difference with our work consists in the fact that [8] builds a coalgebra from the syntactic terms of CSP, while here we build a coalgebra starting from LTS's via the generalized powerset construction [28]. Our approach puts in evidence the underlying semilattice structure which is needed for defining bisimulations up-to and HKC. As a further coalgebraic approach to testing, it is worth mentioning test-suites [18], which however do not tackle must testing. A coalgebraic characterization of other semantics of the linear time/branching time spectrum is given in [3].

Notation. We denote sets by capital letters $X, Y, S, T \dots$ and functions by lower case letters $f, g, \dots$ Given sets X and Y, $X \times Y$ is the Cartesian product of X and Y, $X + Y$ is the disjoint union and X^Y is the set of functions $f\colon Y \to X$. The collection of *finite* subsets of X is denoted by $\mathcal{P}(X)$ (or just $\mathcal{P}X$). These operations, defined on sets, can analogously be defined on functions [26], yielding (bi-)functors on **Set**, the category of sets and functions. For a set of symbols A, A^* denotes the set of all finite words over A; ε the empty word; and $w_1 \cdot w_2$ (or $w_1 w_2$) the concatenation of words $w_1, w_2 \in A^*$. We use 2 to denote the set $\{0, 1\}$ and 2^{A^*} to denote the set of all formal languages over A. A *semilattice with bottom* $(X, \sqcup, 0)$ consists of a set X and a binary operation $\sqcup\colon X \times X \to X$ that is associative, commutative, idempotent (ACI) and has $0 \in X$ (the bottom) as identity. A *homomorphism* (of semilattices with bottom) is a function preserving $\sqcup$ and 0. Every semilattice induces a *partial order* defined as $x \sqsubseteq y$ iff $x \sqcup y = y$. The set 2 is a semilattice when taking $\sqcup$ to be the ordinary Boolean disjunction. Also the set of all languages 2^{A^*} carries a semilattice structure where $\sqcup$ is the union of languages and 0 is the empty language. More generally, for any set S, $\mathcal{P}(S)$ is a semilattice where $\sqcup$ is the union of sets and 0 is the empty set. In the rest of the paper we will indiscriminately use 0 to denote the element $0 \in 2$, the empty language in 2^{A^*} and the empty set in $\mathcal{P}(S)$. Analogously, $\sqcup$ will denote the "Boolean or" in 2, the union of languages in 2^{A^*} and the union of sets in $\mathcal{P}(S)$.

Acknowledgments. This work was supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program Investissements dAvenir (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR). In addition, Filippo Bonchi was partially supported by the projects PEPS-CNRS CoGIP, ANR-09-BLAN-0169-01, and ANR 12IS02001 PACE. Georgiana Caltais has been partially supported by the project 'Meta-theory of Algebraic Process Theories' (nr. 100014021) of the Icelandic Research Fund. Damien Pous was partially supported by the PiCoq project, ANR-10-BLAN-0305. Alexandra Silva was partially supported by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project ref. `PTDC/EIA-CCO/122240/2010` and `SFRH/BPD/71956/2010`.

2 Background

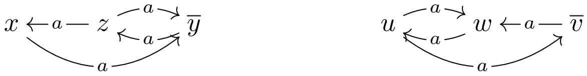
The core of this paper is about the problem of checking whether two states in a transition system are testing equivalent by reducing it to the classical problem of checking language equivalence. We will consider different types of transition systems, deterministic and non-deterministic, which we will formally describe next, together with their language semantics.

A *deterministic automaton* (DA) over the alphabet A is a pair $(S, \langle o, t\rangle)$, where S is a set of states and $\langle o, t\rangle \colon S \to 2 \times S^A$ is a function with two components: o, the output function, determines whether a state x is final ($o(x) = 1$) or not ($o(x) = 0$); and t, the transition function, returns for each state and each input letter, the next state. From any DA, there exists a function $[\![-]\!] \colon S \to 2^{A^*}$ mapping states to languages, defined for all $x \in S$ as follows:

$$[\![x]\!](\varepsilon) = o(x) \qquad\qquad [\![x]\!](a \cdot w) = [\![t(x)(a)]\!](w) \tag{1}$$

The language $[\![x]\!]$ is called the language accepted by x. Given an automaton $(S, \langle o, t\rangle)$, the states $x, y \in S$ are said to be *language equivalent* iff they accept they same language.

A *non-deterministic automaton* (NDA) is similar to a DA but the transition function returns a set of next-states instead of a single state. Thus, an NDA over the input alphabet A is a pair $(S, \langle o, t\rangle)$, where S is a set of states and $\langle o, t\rangle \colon S \to 2 \times (\mathcal{P}(S))^A$. An example is depicted below (final states are overlined, labeled edges represent transitions).



(2)

Classically, in order to recover language semantics of NDA, one uses the *subset (or powerset) construction*, transforming every NDA $(S, \langle o, t\rangle)$ into the

DA $(\mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$ where $o^\sharp \colon \mathcal{P}(S) \to 2$ and $t^\sharp \colon \mathcal{P}(S) \to \mathcal{P}(S)^A$ are defined for all $X \in \mathcal{P}(S)$ as

$$o^\sharp(X) = \bigsqcup_{x \in X} o(x) \qquad\qquad t^\sharp(X)(a) = \bigsqcup_{x \in X} t(x)(a) \ .$$

For instance with the NDA from (2), $o^\sharp(\{x,y\}) = 0 \sqcup 1 = 1$ (*i.e.*, the state $\{x,y\}$ is final) and $t^\sharp(\{x,y\})(a) = \{y\} \sqcup \{z\} = \{y,z\}$ (*i.e.*, $\{x,y\} \xrightarrow{a} \{y,z\}$).

Since $(\mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$ is a deterministic automaton, we can now apply (1), yielding a function $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ mapping *sets* of states to languages. Given two states x and y, we say that they are language equivalent iff $[\![\{x\}]\!] = [\![\{y\}]\!]$. More generally, for two sets of states $X, Y \subseteq S$, we say that X and Y are language equivalent iff $[\![X]\!] = [\![Y]\!]$.

In order to introduce the algorithms in full generality, it is important to remark here that the sets 2, $\mathcal{P}(S)$, $\mathcal{P}(S)^A$, $2 \times \mathcal{P}(S)^A$ and 2^{A^*} carry semilattices with bottom and that the functions $\langle o^\sharp, t^\sharp \rangle \colon \mathcal{P}(S) \to 2 \times \mathcal{P}(S)^A$ and $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ are homomorphisms.

2.1 Checking language equivalence via bisimulation up-to

We recall the algorithm HKC from [7]. We first define a notion of bisimulation on sets of states. We make explicit the underlying notion of progression.

Definition 1 (Progression, Bisimulation). *Let $(S, \langle o, t \rangle)$ be an NDA. Given two relations $R, R' \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$, R* progresses *to R', denoted $R \rightarrowtail R'$, if whenever $X \mathrel{R} Y$ then*

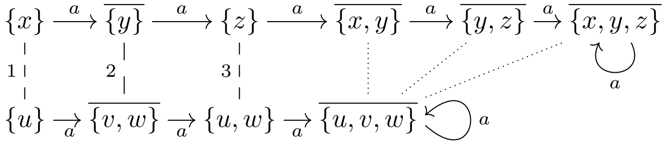
1. $o^\sharp(X) = o^\sharp(Y)$ and 2. for all $a \in A$, $t^\sharp(X)(a) \mathrel{R'} t^\sharp(Y)(a)$.

A bisimulation *is a relation R such that $R \rightarrowtail R$.*

This definition considers the states, the transitions and the outputs of the *determinized* NDA. For this reason, the bisimulation proof technique is sound and complete for language equivalence rather than for the standard notion of bisimilarity by Milner and Park [21].

Proposition 1 (Coinduction [7]). *For all $X, Y \in \mathcal{P}(S)$, $[\![X]\!] = [\![Y]\!]$ iff there exists a bisimulation that relates X and Y.*

For an example, we want to check the equivalence of $\{x\}$ and $\{u\}$ of the NDA in (2). The part of the determinized NDA that is reachable from $\{x\}$ and $\{u\}$ is depicted below. The relation consisting of dashed and dotted lines is a bisimulation which proves that $[\![\{x\}]\!] = [\![\{u\}]\!]$.



(3)

The dashed lines (numbered by 1, 2, 3) form a smaller relation which is not a bisimulation, but a *bisimulation up-to congruence*: the equivalence of $\{x, y\}$ and $\{u, v, w\}$ can be immediately deduced from the fact that $\{x\}$ is related to $\{u\}$ and $\{y\}$ to $\{v, w\}$. In order to formally introduce bisimulations up-to congruence, we need to define first the *congruence closure* $c(R)$ of a relation $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$. This is done inductively, by the following rules:

$$\begin{array}{ccc} \dfrac{X\ R\ Y}{X\ c(R)\ Y} & \dfrac{}{X\ c(R)\ X} & \dfrac{X\ c(R)\ Y}{Y\ c(R)\ X} \\[2ex] \dfrac{X\ c(R)\ Y\ \ Y\ c(R)\ Z}{X\ c(R)\ Z} & & \dfrac{X_1\ c(R)\ Y_1\ \ X_2\ c(R)\ Y_2}{X_1 \sqcup X_2\ c(R)\ Y_1 \sqcup Y_2} \end{array} \tag{4}$$

Note that the term "congruence" here is intended w.r.t. the semilattice structure carried by the state space $\mathcal{P}(S)$ of the determinized automaton. Intuitively, $c(R)$ is the smallest equivalence relation containing R and which is closed w.r.t $\sqcup$.

Definition 2 (Bisimulation up-to congruence). *A relation* $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ *is a* bisimulation up-to c *if* $R \rightarrowtail c(R)$, i.e.*, whenever* $X\ R\ Y$ *then*

1. $o^\sharp(X) = o^\sharp(Y)$ *and* *2. for all* $a \in A$, $t^\sharp(X)(a)\ c(R)\ t^\sharp(Y)(a)$.

Theorem 1 ([7]). *Any bisimulation up-to c is contained in a bisimulation.*

The corresponding algorithm (`HKC`) is given in Figure 1 (left). Starting from an NDA $(S, \langle o, t\rangle)$ and considering the determinized automaton $(\mathcal{P}(S), \langle o^\sharp, t^\sharp\rangle)$, it can be used to check language equivalence of two sets of states X and Y. Starting from the pair (X, Y), the algorithm builds a relation R that, in case of success, is a bisimulation up-to congruence. In order to do that, it employs the set *todo* which, intuitively, at any step of the execution, contains the pairs (X', Y') that must be checked: if (X', Y') already belongs to $c(R \cup \mathit{todo})$, then it does not need to be checked. Otherwise, the algorithm checks if X' and Y' have the same outputs. If $o^\sharp(X') \neq o^\sharp(Y')$ then X and Y are different, otherwise the algorithm inserts (X', Y') in R and, for all $a \in A$, the pairs $(t^\sharp(X')(a), t^\sharp(Y')(a))$ in *todo*. The check $(X', Y') \in c(R \cup \mathit{todo})$ at step `2.2` is done with the rewriting algorithm of [7, Section 3.4].

Proposition 2. *For all* $X, Y \in \mathcal{P}(S)$, $[\![X]\!] = [\![Y]\!]$ *iff* `HKC`(X, Y).

The iterations corresponding to the execution of `HKC`$(\{x\}, \{u\})$ on the NDA in (2) are concisely described by the numbered dashed lines in (3). Observe that only a small portion of the determinized automaton is explored; this fact usually makes `HKC` more efficient than the algorithms based on minimization, that need to build the whole reachable part of the determinized automaton.

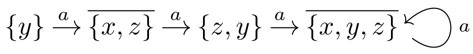
2.2 Checking language equivalence via Brzozowski's algorithm

The problem of checking language equivalence of two sets of states X and Y of a non-deterministic finite automaton can be reduced to that of building the minimal DA for $[\![X]\!]$ and $[\![Y]\!]$ and checking whether they are the same (up to isomorphism). The most well-known procedure consists in first determinizing the NDA and then minimizing it with the Hopcroft algorithm [16]. Another interesting solution is Brzozowski's algorithm [9].

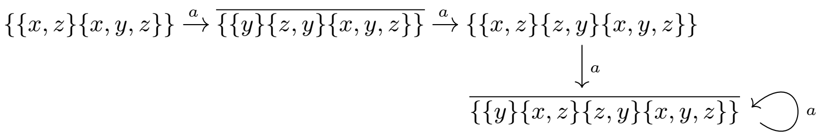
To explain the latter, it is convenient to consider a set of *initial states* I. Given an NDA $(S, \langle o, t\rangle)$ and a set of states I, Brzozowski's algorithm computes the minimal automaton for the language $[\![I]\!]$ by performing the 4 steps in Figure 1 (right).

The operation `reverse and determinize` takes as input an NDA $(S, \langle o, t\rangle)$ and returns a DA $(\mathcal{P}(S), \langle \overline{o}_R, \overline{t}_R\rangle)$ where the functions $\overline{o}_R \colon \mathcal{P}(S) \to 2$ and $\overline{t}_R \colon \mathcal{P}(S) \to \mathcal{P}(S)^A$ are defined for all $X \in \mathcal{P}(S)$ as $\overline{o}_R(X) = 1$ iff $X \cap I \neq 0$ and $\overline{t}_R(X)(a) = \{x \in S \mid t(x)(a) \cap X \neq 0\}$. The new initial state is the set of accepting states of the original NDA: $\overline{I}_R = \{x \mid o(x) = 1\}$. The second step consists in taking the part of $(\mathcal{P}(S), \langle \overline{o}_R, \overline{t}_R\rangle)$ which is reachable from $\overline{I}_R$. The third and the fourth steps perform this procedure once more.

As an example, consider the NDA in (2) with the set of initial states $I = \{x\}$. Brzozowski's algorithm builds the minimal DA accepting $[\![\{x\}]\!]$ as follows. After the first two steps, it returns the following DA where the initial state is $\{y\}$.



After steps 3 and 4, it returns the DA below with initial state $\{\{x,z\}\{x,y,z\}\}$.


$\{\{x,z\}\{x,y,z\}\}$
a
$\overline{\{\{y\}\{z,y\}\{x,y,z\}\}}$
a
$\{\{x,z\}\{z,y\}\{x,y,z\}\}$
a
$\overline{\{\{y\}\{x,z\}\{z,y\}\{x,y,z\}\}}$
a

Computing the minimal NDA in (2) with the set of initial states $I = \{u\}$ results in an isomorphic automaton, showing the equivalence of x and u.

2.3 Generalized Powerset Construction

The notions introduced above can be easily described using *coalgebras*. Given a functor $F \colon \mathbf{Set} \to \mathbf{Set}$, an F*-coalgebra* is a pair (S, f) where S is a set of states and $f \colon S \to F(S)$ is its *transition structure*. F intuitively determines the "type" of the transitions. An F*-homomorphism* from an F-coalgebra (S, f) to an F-coalgebra (T, g) is a function $h \colon S \to T$ preserving the transition structure, *i.e.*, $g \circ h = F(h) \circ f$. An F-coalgebra (Ω, ω) is said to be *final* if for any F-coalgebra (S, f) there exists a unique F-homomorphism $[\![-]\!] \colon S \to \Omega$. Intuitively, Ω represents the universe of "F-behaviours" and $[\![-]\!]$ represents the semantic map associating states to their behaviours. Two states $x, y \in X$ are said F*-behaviourally equivalent* iff $[\![x]\!] = [\![y]\!]$. Such equivalence can be proved using F*-bisimulations* [26]. For lack of space, we refer the reader to [25] for their categorical definitions. Given a behaviour $b \in \Omega$, the *minimal coalgebra* realizing b is the part of (Ω, ω) that is reachable from b.

Let us exemplify for DA's how these abstract notions yield the expected concrete notions. DA's are coalgebras for the functor $F(S) = 2 \times S^A$. The final coalgebra of this functor is the set 2^{A^*} of formal languages over A, or more precisely, the pair $(2^{A^*}, \langle \epsilon, (-)_a \rangle)$ where $\langle \epsilon, (-)_a \rangle$, given a language L, determines whether or not the empty word is in the language ($\epsilon(L) = 1$ or $\epsilon(L) = 0$, resp.) and, for each input letter a, returns the *a-derivative* of L: $L_a = \{w \in A^* \mid aw \in L\}$. The unique map $[\![-]\!]$ into the final coalgebra 2^{A^*} is precisely the map which assigns to each state the language that it recognizes. For any language $L \in 2^{A^*}$, the minimal automaton for L is the part of $(2^{A^*}, \langle \epsilon, (-)_a \rangle)$ that is reachable from L.

In Section 3, we will use *Moore machines* which are coalgebras for the functor $F(S) = B \times S^A$. These are like DA's, but with outputs in a fixed set B. The unique F-homomorphism to the final coalgebra $[\![-]\!] \colon S \to B^{A^*}$ is defined exactly as for DA's by the equations in (1). Note that the behaviours of Moore machines are functions $\varphi \colon A^* \to B$, rather than subsets of A^*. For each behaviour $\varphi \in B^{A^*}$, there exists a minimal Moore machine realizing it.

Recall that an NDA is a pair $(S, \langle o, t \rangle)$, where $\langle o, t \rangle \colon S \to 2 \times (\mathcal{P}(S))^A$. As explained above, to recover language semantics one needs to use the subset construction, which transforms an NDA into a DA. More abstractly, this can be captured by observing that the type functor of NDA's – $2 \times \mathcal{P}(-)^A$ – is a composition of the functor $F(S) = 2 \times S^A$ (that is the functor for DA's) and the monad $T(S) = \mathcal{P}(S)$. $\mathcal{P}$-algebras are exactly semilattices with bottom and $\mathcal{P}$-algebra morphisms are the ones of semilattices with bottom. Now note that (a) the F-coalgebra $(\mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$ resulting of the powerset construction is a morphism of semilattices, (b) 2^{A^*} carries a semilattice structure and (c) $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ is a morphism of semilattices. This is summarized by the following commuting diagram:

$$\begin{tikzcd}
S \arrow[r, "\{-\}"] \arrow[d, "\langle o,t \rangle"'] & \mathcal{P}(S) \arrow[r, dashed, "{[\![-]\!]}"] \arrow[dl, "\langle o^\sharp, t^\sharp \rangle"] & 2^{A^*} \arrow[d, "\langle \epsilon, (-)_a \rangle"] \\
2 \times \mathcal{P}(S)^A \arrow[rr, dashed, "id_2 \times {[\![-]\!]}^A"] & & 2 \times (2^{A^*})^A
\end{tikzcd}$$

In the diagram above, one can replace $2 \times -^A$ and $\mathcal{P}$ by arbitrary F and T as long as $FT(S)$ *carries a* T*-algebra structure*. In fact, given an FT*-coalgebra*, that is $(S, f \colon S \to FT(S))$, if $FT(S)$ carries a T-algebra structure h, then (a) one can define an F-coalgebra $(T(S), f^\sharp = h \circ Tf)$ where $f^\sharp \colon T(S) \to FT(S)$ is a T-algebra morphism (b) the final F-coalgebra (Ω, ω) carries a T-algebra and (c) the F-homomorphism $[\![-]\!] \colon T(S) \to \Omega$ is a T-algebra morphism.

The F-coalgebra $(T(S), f^\sharp)$ is (together with the multiplication $\mu \colon TT(S) \to T(S)$) a *bialgebra* for some distributive law $\lambda \colon FT \Rightarrow TF$ (we refer the reader to [19] for a nice introduction on this topic). The behavioural equivalence of bialgebras can be proved either via bisimulation, or, like in Section 2.1, via *bisimulation up-to congruence* [20, 25]: the result that justifies HKC (Theorem 1)

```
HKC($X,Y$):
(1) $R$ is empty; $todo$ is $\{(X',Y')\}$;
(2) while $todo$ is not empty, do
  (2.1) extract $(X',Y')$ from $todo$;
  (2.2) if $(X',Y') \in c(R \cup todo)$ then continue;
  (2.3) if $o^\sharp(X') \neq o^\sharp(Y')$ then return $false$;
  (2.4) for all $a \in A$,
          insert $(t^\sharp(X')(a), t^\sharp(Y')(a))$ in $todo$;
  (2.5) insert $(X',Y')$ in $R$;
(3) return $true$;
```

```
Brzozowski :
(1) reverse and determinize;
(2) take the reachable part;
(3) reverse and determinize;
(4) take the reachable part.
```

Fig. 1: Left: Generic HKC algorithm, parametric on $o^\sharp$, $t^\sharp$ and c; Right: Generic Brzozowski's algorithm, parametric on **reverse and determinize**. Instantiations to language and must equivalence in Sections 2 and 3.

generalises to this setting – the congruence being taken w.r.t. the algebraic structure μ. This is what allows us to move to must semantics.

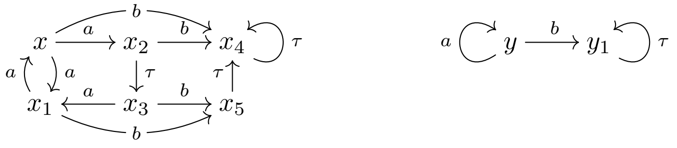
3 Must semantics

The operational semantics of concurrent systems is usually given by *labelled transition systems* (LTS's), labelled by actions that are either visible to an external observer or internal actions (usually denoted by a special symbol τ). Different kinds of semantics can be defined on these structures (*e.g.*, linear or branching time, strong or weak semantics). In this paper we consider *must semantics* [14] which, intuitively, equates those systems that pass exactly the same tests, in all their executions.

Before formally introducing *must semantics* as in [12], we fix some notations: $\overset{\varepsilon}{\Rightarrow}$ denotes $\xrightarrow{\tau}^*$ the reflexive and transitive closure of $\xrightarrow{\tau}$ and, for $a \in A$, $\overset{a}{\Rightarrow}$ denotes $\xrightarrow{\tau}^* \xrightarrow{a} \xrightarrow{\tau}^*$. For $w \in A^*$, $\overset{w}{\Longrightarrow}$ is defined inductively, in the obvious way. The *acceptance set of x after w* is $A(x,w) = \{\{a \in A \mid x' \xrightarrow{a}\} \mid x \overset{w}{\Longrightarrow} x' \wedge x' \overset{\tau}{\not\rightarrow}\}$. Intuitively, it represents the set of actions that can be fired after "maximal" executions of w from x, those that cannot be extended by some τ-labelled transitions. The possibility of executing τ-actions forever is referred to as *divergence*. We write $x \not\downarrow$ whenever x diverges. Dually, the convergence relation $x \downarrow w$ for a state x and a word $w \in A^*$ is inductively defined as follows: $x \downarrow \varepsilon$ iff x does not diverge and $x \downarrow aw'$ iff (a) $x \downarrow \varepsilon$ and (b) if $x \overset{a}{\Rightarrow} x'$, then $x' \downarrow w'$. Given two sets $B, C \in \mathcal{P}\mathcal{P}(A)$, we write $B \subset\subset C$ iff for all $B_i \in B$, there exists $C_i \in C$ such that $C_i \subseteq B_i$. With these ingredients, it is possible to introduce must preorder and equivalence.

Definition 3 (Must semantics [12]). *Let x and y be two states of an LTS. We write $x \sqsubseteq_{mst} y$ iff for all words $w \in A^*$, if $x \downarrow w$ then $y \downarrow w$ and $A(y,w) \subset\subset A(x,w)$. We say that x and y are must-equivalent ($x \sim_{mst} y$) iff $x \sqsubseteq_{mst} y$ and $y \sqsubseteq_{mst} x$.*

As an example, consider the LTS depicted below. States x_4, x_5 and y_1 are divergent. All the other states diverge for words containing the letter b and converge for words on a^*. For these words and states x, x_1, x_2, x_3 and y, the corresponding acceptance sets are $\{\{a, b\}\}$. In particular, note that $A(x_2, \varepsilon)$ is $\{\{a, b\}\}$ and not $\{\{b\}, \{a, b\}\}$. It is therefore easy to conclude that x, x_1, x_2, x_3 and y are all must equivalent.



(5)

3.1 A coalgebraic characterization of Must semantics

In what follows we show how $\sqsubseteq_{mst}$ can be captured in terms of coalgebras. This will further allow adapting the algorithms introduced in Section 2 for checking $\sim_{mst}$ and $\sqsubseteq_{mst}$.

First, we model LTS's in terms of coalgebras $(S, t \colon S \to (1 + \mathcal{P}(S))^A)$, where $1 = \{\top\}$ is the singleton set, and for $x \in S$,

$$t(x)(a) = \top, \text{ if } x \not\downarrow a \qquad t(x)(a) = \{y \mid x \overset{a}{\Rightarrow} y\}, \text{ otherwise.}$$

Intuitively, a state $x \in S$ that displays divergent behaviour with respect to an action $a \in A$ is mapped to $\top$. Otherwise t computes the set of states that can be reached from x through a (by possibly performing a finite number of τ-transitions). At this point we need some additional definitions: for a function $\varphi \colon A \to \mathcal{P}(S)$, $I(\varphi)$ denotes the set of all labels "enabled" by φ, given by $I(\varphi) = \{a \in A \mid \varphi(a) \neq 0\}$, while $\mathit{Fail}(\varphi)$ denotes the set $\{Z \subseteq A \mid Z \cap I(\varphi) = 0\}$. With these definitions, we decorate the states of an LTS by means of an output function $o \colon S \to 1 + \mathcal{P}(\mathcal{P}(A))$ defined as follows:

$$o(x) = \top, \text{ if } x \not\downarrow \qquad o(x) = \bigcup_{x \xrightarrow{\tau} x'} o(x') \text{ if } x \xrightarrow{\tau}, \qquad o(x) = \mathit{Fail}(t(x)), \text{ otherwise.}$$

Note that $(S, \langle o, t \rangle)$ is an FT-coalgebra for the functor $F(S) = (1 + \mathcal{P}\mathcal{P}A) \times S^A$ and the monad $T(S) = 1 + \mathcal{P}(S)$. Algebras for such monad T are semilattices with bottom and an extra element $\top$ acting as *top* (*i.e.*, such that $x \sqcup \top = \top$ for all x). For any set U, $1 + \mathcal{P}(U)$ carries a semilattice with bottom and top: bottom is the empty set; top is the element $\top \in 1$; $X \sqcup Y$ is defined as the union for arbitrary subsets $X, Y \in \mathcal{P}(U)$ and as $\top$ otherwise. Consequently, $1 + \mathcal{P}(\mathcal{P}A)$, $1 + \mathcal{P}(S)$, $(1 + \mathcal{P}(S))^A$ and $FT(S)$ carry a T-algebra structure as well. This enables the application of the generalized powerset construction (Section 2.3) associating to each FT-coalgebra $(S, \langle o, t \rangle)$ the F-coalgebra $(1 + \mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$ defined for all $X \in 1 + \mathcal{P}(S)$ as expected:

$$o^\sharp(X) = \begin{cases} \top & \text{if } X = \top \\ \bigsqcup_{x \in X} o(x) & \text{if } X \in \mathcal{P}(S) \end{cases} \quad t^\sharp(X)(a) = \begin{cases} \top & \text{if } X = \top \\ \bigsqcup_{x \in X} t(x)(a) & \text{if } X \in \mathcal{P}(S) \end{cases}$$

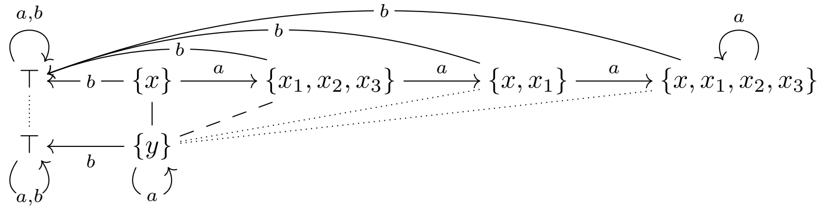
Note that in the above definitions, $\sqcup$ is not simply the union of subsets, but it is the join operation in $1 + \mathcal{P}\mathcal{P}A$ and $1 + \mathcal{P}(S)$. Moreover, $(1 + \mathcal{P}S, \langle o^\sharp, t^\sharp \rangle)$ is a Moore machine with output in $1 + \mathcal{P}\mathcal{P}A$ and, therefore, the equations in (1) induce a function $[\![-]\!] \colon (1 + \mathcal{P}(S)) \to (1 + \mathcal{P}\mathcal{P}A)^{A^*}$. The semilattice structure of $1 + \mathcal{P}\mathcal{P}A$ can be easily lifted to $(1 + \mathcal{P}\mathcal{P}A)^{A^*}$: bottom, top and $\sqcup$ are defined pointwise on A^*. If $\sqsubseteq_\mathcal{M}$ represents the preorder on $(1 + \mathcal{P}\mathcal{P}A)^{A^*}$ induced by this semilattice, then the following theorem holds.

Theorem 2. *$x \sqsubseteq_{mst} y$ iff $[\![\{y\}]\!] \sqsubseteq_\mathcal{M} [\![\{x\}]\!]$ and $x \sim_{mst} y$ iff $[\![\{x\}]\!] = [\![\{y\}]\!]$.*

Note that according to the definition of $\sqsubseteq_\mathcal{M}$, $[\![\{y\}]\!] \sqsubseteq_\mathcal{M} [\![\{x\}]\!]$ iff $[\![\{y\}]\!] \sqcup [\![\{x\}]\!] = [\![\{x\}]\!]$, and since $[\![-]\!]$ is a T-homomorphism (namely it preserves bottom, top and $\sqcup$), the latter equality holds iff $[\![\{y, x\}]\!] = [\![\{x\}]\!]$. Summarizing,

$$x \sqsubseteq_{mst} y \text{ iff } [\![\{x, y\}]\!] = [\![\{x\}]\!].$$

Consider, once more, the LTS in (5). The part of the Moore machine $(1 + \mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$ which is reachable from $\{x\}$ and $\{y\}$ is depicted below (the output function $o^\sharp$ maps $\top$ to $\top$ and the other states to $\{0\}$). The relation consisting of dashed and dotted lines is a bisimulation proving that $[\![\{x\}]\!] = [\![\{y\}]\!]$, *i.e.*, that $x \sim_{mst} y$.



(6)

Our construction is closely related to the one in [12], that transforms LTS's into (deterministic) acceptance graphs. We refer the interested reader to a detailed comparison provided in the full version of this paper [5]. There we also show an optimization for representing outputs by means of $I(t(x))$ rather $Fail(t(x))$.

3.2 `HKC` for must semantics

The coalgebraic characterization discussed in the previous section guarantees soundness and completeness of bisimulation up-to congruence for must equivalence. Bisimulations are now relations $R \subseteq (1 + \mathcal{P}(S)) \times (1 + \mathcal{P}(S))$ on the state space $1 + \mathcal{P}(S)$ where $o^\sharp$ and $t^\sharp$ are defined as in Section 3.1. Now, the congruence closure $c(R)$ of a relation $R \subseteq (1 + \mathcal{P}(S)) \times (1 + \mathcal{P}(S))$ is defined by the rules in (4) where $\sqcup$ is the join in $(1 + \mathcal{P}(S))$ (rather than the union in $\mathcal{P}(S)$). By simply redefining $o^\sharp$, $t^\sharp$ and $c(R)$, the algorithm in Figure 1 can be used to check must equivalence and preorder (the detailed proof can be found in the full version of the paper [5]). In particular, note that the check at step `2.1` can be done with the same algorithm as in [7, Section 3.4].

Suppose, for example, that we want to check whether the states x and y of the LTS in (5) are must equivalent. The relation $R = \{(\{x\}, \{y\}), (\{x_1, x_2, x_3\}, \{y\})\}$

depicted by the dashed lines in (6) is not a bisimulation, but a bisimulation up-to congruence, since both $(\top, \top) \in c(R)$ and $(\{x, x_1\}, \{y\}) \in c(R)$. For the latter, observe that

$$\{x, x_1\}\ c(R)\ \{y, x_1\}\ c(R)\ \{x_1, x_2, x_3\}\ c(R)\ \{y\}.$$

It is important to remark here that `HKC` computes this relation without the need of exploring all the reachable part of the Moore machine $(1 + \mathcal{P}(S), \langle o^\sharp, t^\sharp\rangle)$. So, amongst all the states in (6), `HKC` only explores $\{x\}$, $\{y\}$ and $\{x_1, x_2, x_3\}$.

3.3 Brzozowski's algorithm for must semantics

A variation of the Brzozowski algorithm for Moore machines is given in [4]. We could apply such algorithm to the Moore machine $(1 + \mathcal{P}(S), \langle o^\sharp, t^\sharp\rangle)$ which is induced by the coalgebra $(S, \langle o, t\rangle)$ introduced in Section 3.1. Here, we propose a more efficient variation that skips the first determinization from $(S, \langle o, t\rangle)$ to $(1 + \mathcal{P}(S), \langle o^\sharp, t^\sharp\rangle)$.

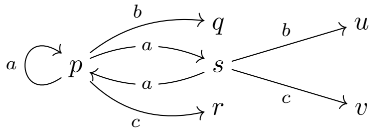
The novel algorithm consists of the four steps described in Section 2.2, where the procedure `reverse and determinize` is modified as follows: $(S, \langle o, t\rangle)$ with the set of initial states I is transformed into $((1 + \mathcal{P}\mathcal{P}(A))^S, \langle \overline{o}_R, \overline{t}_R\rangle)$ where $\overline{o}_R\colon (1+\mathcal{P}\mathcal{P}A)^S \to 1+\mathcal{P}\mathcal{P}A$ and $\overline{t}_R\colon (1+\mathcal{P}\mathcal{P}A)^S \to ((1+\mathcal{P}\mathcal{P}A)^S)^A$ are defined for all functions $\psi \in (1 + \mathcal{P}\mathcal{P}A)^S$ as

$$\overline{o}_R(\psi) = \bigsqcup_{x \in I} \psi(x) \qquad \overline{t}_R(\psi)(a)(x) = \begin{cases} \top & \text{if } t(x)(a) = \top \\ \bigsqcup_{y \in t(x)(a)} \psi(y) & \text{otherwise} \end{cases}$$

and the new initial state is $\overline{I}_R = o$.

Note that the result of this procedure is a Moore machine. Brzozowski's algorithm in Section 2.2 transforms an NDA $(S, \langle o, t\rangle)$ with initial state I into the minimal DA for $[\![I]\!]$. Analogously, our algorithm transforms an LTS into the minimal Moore machine for $[\![I]\!]$.

Let us illustrate the minimization procedure by means of an example. Take the alphabet $A = \{a, b, c\}$ and the LTS depicted below on the left.

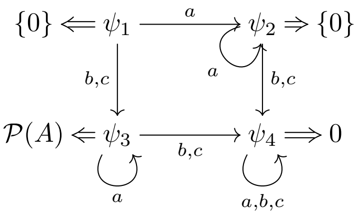


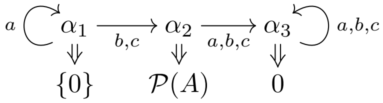
$$\begin{array}{rr} o(p) = \{0\} & o(s) = \{0\} \\ o(q) = \mathcal{P}(A) & o(u) = \mathcal{P}(A) \\ o(r) = \mathcal{P}(A) & o(v) = \mathcal{P}(A) \end{array}$$

Since there are no τ transitions, the function $t\colon S \to (1 + \mathcal{P}(S))^A$ is defined as on the left, and the function $o\colon S \to (1 + \mathcal{P}\mathcal{P}A)$ (given on the right) assigns to each state x the set $\mathit{Fail}(t(x))$. Suppose we want to build the minimal Moore machine for the behaviour $[\![\{p\}]\!]\colon A^* \to 1 + \mathcal{P}\mathcal{P}A$, which is the function

$$[\![\{p\}]\!]\colon a^* \mapsto \{0\},\ a^*b \mapsto \mathcal{P}(A),\ a^*c \mapsto \mathcal{P}(A),\ \_ \mapsto 0$$

where _ denotes all the words different from a^*, a^*b and a^*c. By applying our algorithm to the coalgebra $(S, \langle o, t\rangle)$, we first obtain the intermediate Moore machine on the left below, where a double arrow $\psi \Rightarrow Z$ means that the output of ψ is the set Z. The initial state is $\psi_1 \colon S \to 1 + \mathcal{P}\mathcal{P}A$ which, by definition, is the output function o above. The explicit definitions of the other functions ψ_i can be computed according to the definition of $\bar{t}_R$.





Observe that $[\![\psi_1]\!]$ is the "reversed" of $[\![\{p\}]\!]$. For instance, triggering ba^* from ψ_1 leads to ψ_3 with output $\mathcal{P}(A)$; this is the same output we get by executing a^*b from p, according to $[\![\{p\}]\!]$. Executing `reverse and determinize` once more (step 3) and taking the reachable part (step 4), we obtain the minimal Moore machine on the right, with initial state α_1.

We have proved the correctness of this algorithm in the full version of this paper [5]; it builds on the coalgebraic perspective on Brzozowski's algorithm given in [4].

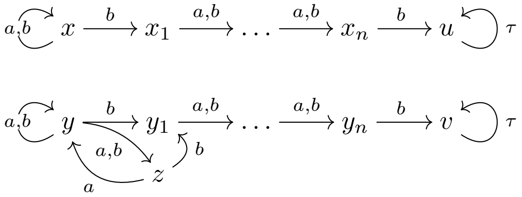
4 A family of examples

As discussed in the introduction, the problem of checking must equivalence is PSPACE-complete [17]. Hence, a theoretical comparison of HKC, Brzozowski (BRZ) and the partition refinement (PR) of [12] will be less informative than a thorough experimental analysis. Designing adequate experiments is out of the scope of this paper. We will instead just show the reader some concrete examples. It is possible to show some concrete cases where (a) HKC takes polynomial time while BRZ and PR exponential time and (b) (BRZ) polynomial time while HKC and PR exponential time. There are also examples where (c) PR is polynomial and BRZ is exponential, but it is impossible to have PR polynomial and HKC exponential. Indeed, cycle 2 of HKC is repeated at most $1 + |A|\cdot|R|$ times where $|A|$ is the size of the alphabet and $|R|$ is the size of the produced relation R. Such relation always contains at most n pairs of states, for n being the size of the reachable part of the determinised system. Therefore, if HKC takes exponential time, then also PR takes exponential time since it always needs to build the reachable part of the determinised LTS.

In this section we show an example for (a). Examples for (b) and (c) can be found in the full version of this paper [5].

Consider the following LTS, where n is an arbitrary natural number. After the determinization, $\{x\}$ can reach all the states of the shape $\{x\} \cup X_N$, where

$X_N = \{x_i \mid i \in N\}$ for any $N \subseteq \{1,\dots,n\}$. For instance for $n=2$, $\{x\} \overset{aa}{\to} \{x\}$, $\{x\} \overset{ab}{\to} \{x, x_1\}$, $\{x\} \overset{ba}{\to} \{x, x_2\}$ and $\{x\} \overset{bb}{\to} \{x, x_1, x_2\}$. All those states are distinguished by must and, therefore, the minimal Moore machine for $[\![\{x\}]\!]$ has at least 2^n states.



One can prove that x and y are must equivalent by showing that relation

$$\begin{aligned} R = \{&(\{x\},\{y\}), (\{x\},\{y,z\}), (\top,\top)\} \\ \cup\, \{&(\{x\} \cup X_N,\ \{y,z\} \cup Y_N) \mid N \subseteq \{1,\dots,n\}\} \end{aligned}$$

is a bisimulation (here $Y_N = \{y_i \mid i \in N\}$). Note that R contains $2^n + 2$ pairs.

In order to check $[\![\{x\}]\!]{=}[\![\{y\}]\!]$, `HKC` builds the following relation,

$$R' = \{(\{x\},\{y\}), (\{x\},\{y,z\})\} \cup \{(\{x, x_i\},\ \{y, z, y_i\}) \mid i \in \{1,\dots,n\}\}$$

which is a bisimulation up-to and which contains only $n+2$ pairs. It is worth to observe that R' is like a "basis" of R: all the pairs $(X,Y) \in R$ can be generated by those in R' by iteratively applying the rules in (4). Therefore, `HKC` proves $[\![\{x\}]\!]{=}[\![\{y\}]\!]$ in polynomial time, while minimization-based algorithms (such as [12] or Brzozowski's algorithm) require exponential time.

5 Conclusions and Future Work

We have introduced a coalgebraic characterization of must testing semantics by means of the *generalized powerset construction* [28]. This allowed us to adapt proof techniques and algorithms that have been developed for language equivalence to must semantics. In particular, we showed that *bisimulations up-to congruence* (that was recently introduced in [7] for NDA's) are sound also for must semantics. This fact guarantees the correctness of a generalization of `HKC` [7] for checking must equivalence and preorder and suggests that the *antichains*-based algorithms [32, 1] can be adapted in a similar way. We have also proposed a variation of Brzozowski's algorithm [9] to check must semantics, by exploiting the abstract theory in [4]. Our contribution is not a simple instantiation of [4], but developing our algorithm has required some ingenuity to avoid the preliminary determinization that would be needed to directly apply [4]. We implemented these algorithms together with an interactive applet available online [6].

We focused on must testing semantics because it is challenging to compute, but our considerations hold also for may testing and for several decorated trace semantics of the *linear time/branching time spectrum* [30] (namely, those that have been studied in [3]). Adapting these algorithms to check *fair testing* [24] seems to be more complicated: while it is possible to coalgebraically capture failure trees, we do not know how to model fair testing equivalence. We believe that this is a challenging topic to investigate in the future. Moreover, since coalgebras can easily model probabilistic systems, we think promising to investigate whether our approach can be extended to the testing semantics of probabilistic and non-deterministic processes (e.g. [15]).

References

1. P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *Proc. TACAS*, volume 6015 of *LNCS*, pages 158–174. Springer, 2010.
2. N. Bezhanishvili, P. Panangaden, and C. Kupke. Minimization via duality. In *WoLLIC*, volume 7456 of *LNCS*, pages 191–205. Springer, 2012.
3. F. Bonchi, M. Bonsangue, G. Caltais, J. Rutten, and A. Silva. Final semantics for decorated traces. *Elect. Not. in Theor. Comput. Sci.*, 286:73–86, 2012.
4. F. Bonchi, M. Bonsangue, J. Rutten, and A. Silva. Brzozowski's algorithm (co)algebraically. In *Logic and Program Semantics*, volume 7230 of *LNCS*, pages 12–23, 2012.
5. F. Bonchi, G. Caltais, D. Pous, and A. Silva. Brzozowski's and up-to algorithms for must testing (full version). Available from http://www.alexandrasilva.org/files/brz-hkc-must-full.pdf.
6. F. Bonchi, G. Caltais, D. Pous, and A. Silva. Web appendix of this paper, with implementation of the algorithms. http://perso.ens-lyon.fr/damien.pous/brz, July 2013.
7. F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In *POPL*, pages 457–468. ACM, 2013.
8. M. Boreale and F. Gadducci. Processes as formal power series: a coinductive approach to denotational semantics. *TCS*, 360(1):440–458, Aug. 2006.
9. J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical Theory of Automata*, volume 12(6), pages 529–561. Polytechnic Press, NY, 1962.
10. F. Calzolai, R. De Nicola, M. Loreti, and F. Tiezzi. TAPAs: A tool for the analysis of process algebras. *In Petri Nets and Other Models of Concurrency*, 5100:54–70, 2008.
11. D. Cancila, F. Honsell, and M. Lenisa. Generalized coiteration schemata. *Elect. Not. in Theor. Comput. Sci.*, 82(1), 2003.
12. R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. In J. Sifakis, editor, *In Proc. AVMFSS'89*, volume 407 of *LNCS*, pages 11–23. Springer, 1989.
13. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *TOPLAS*, 15(1):36–72, 1993.
14. R. De Nicola and M. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1984.
15. Y. Deng, R. J. van Glabbeek, M. Hennessy, and C. Morgan. Real-reward testing for probabilistic processes. In *QAPL*, volume 57 of *EPTCS*, pages 61–73, 2011.

16. J. E. Hopcroft. An n log n algorithm for minimizing in a finite automaton. In *Proc. Int. Symp. of Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.
17. P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. PODC '83, pages 228–240, New York, NY, USA, 1983. ACM.
18. B. Klin. A coalgebraic approach to process equivalence and a coinduction principle for traces. *Elect. Not. in Theor. Comput. Sci.*, 106:201–218, 2004.
19. B. Klin. Bialgebras for structural operational semantics: An introduction. *TCS*, 412(38):5043–5069, 2011.
20. M. Lenisa. From set-theoretic coinduction to coalgebraic coinduction: some results, some problems. *Elect. Not. in Theor. Comput. Sci.*, 19:2–22, 1999.
21. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
22. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.
23. J. Parrow and P. Sjödin. Designing a multiway synchronization protocol. *Computer Communications*, 19(14):1151–1160, 1996.
24. A. Rensink and W. Vogler. Fair testing. *Inf. Comput.*, 205(2):125–198, Feb. 2007.
25. J. Rot, M. Bonsangue, and J. Rutten. Coalgebraic bisimulation-up-to. In *SOFSEM*, pages 369–381, 2013.
26. J. Rutten. Universal coalgebra: a theory of systems. *TCS*, 249(1):3–80, 2000.
27. D. Sangiorgi. On the bisimulation proof method. *Math. Struc. in CS*, 8:447–479, 1998.
28. A. Silva, F. Bonchi, M. Bonsangue, and J. Rutten. Generalizing the powerset construction, coalgebraically. In *Proc. FSTTCS*, volume 8 of *LIPIcs*, pages 272–283, 2010.
29. D. Tabakov and M. Vardi. Experimental evaluation of classical automata constructions. In *Proc. LPAR*, volume 3835 of *LNCS*, pages 396–411. Springer, 2005.
30. R. van Glabbeek. The linear time - branching time spectrum I. The semantics of concrete, sequential processes. In *Handbook of Process Algebra*, pages 3–99. Elsevier, 2001.
31. B. W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Eindhoven University of Technology, the Netherlands, 1995.
32. M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. CAV*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.