ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

On sequentiality and well-bracketing in the π-calculus

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Hirschkoff D., Prebet E., Sangiorgi D. (2021). On sequentiality and well-bracketing in the π-calculus. N.Y. : Institute of Electrical and Electronics Engineers Inc. [10.1109/LICS52264.2021.9470559].

Availability:
This version is available at: https://hdl.handle.net/11585/851822 since: 2022-11-10

Published:
DOI: http://doi.org/10.1109/LICS52264.2021.9470559

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2024

This is the final peer-reviewed accepted manuscript of:

On sequentiality and well-bracketing in the π-calculus

The final published version is available online at:
https://doi.org/10.1109/LICS52264.2021.9470559

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)

When citing, please refer to the published version.

On sequentiality and well-bracketing in the π-calculus

Daniel Hirschkoff
ENS de Lyon

Enguerrand Prebet
ENS de Lyon

Davide Sangiorgi
Università di Bologna and INRIA

***Abstract*—The π-calculus is used as a model for programming languages. Its contexts exhibit arbitrary concurrency, making them very discriminating. This may prevent validating desirable behavioural equivalences in cases when more disciplined contexts are expected.**

In this paper we focus on two such common disciplines: sequentiality, meaning that at any time there is a single thread of computation, and well-bracketing, meaning that calls to external services obey a stack-like discipline. We formalise the disciplines by means of type systems. The main focus of the paper is on studying the consequence of the disciplines on behavioural equivalence. We define and study labelled bisimilarities for sequentiality and well-bracketing. These relations are coarser than ordinary bisimilarity. We prove that they are sound for the respective (contextual) barbed equivalence, and also complete under a certain technical condition.

We show the usefulness of our techniques on a number of examples, that have mainly to do with the representation of functions and store.

I. Introduction

The π-calculus has been advocated as a model to give semantics to, and reason about, various forms of programming languages, including those with higher-order features. Strengths of the π-calculus are its rich algebraic theory and its wide spectrum of proof techniques. Concurrency is at the heart of the π-calculus: computation is interaction between concurrent processes. The operators of the calculus are simple (parallelism, input, output, restriction being the main ones) and unconstrained. This yields an amazing expressive power — the calculus can model a variety of programming idioms [1]. However, this also makes the contexts of the calculus very discriminating; as a consequence, behavioural equivalences, which are supposed to be preserved by all the contexts of the calculus, are rather demanding relations.

Higher-level languages may be syntactically quite different from a language for pure concurrency such as the π-calculus. For instance, the paradigmatic higher-order programming language, the λ-calculus, is a pure calculus of functions and, in both its call-by-name and call-by-value variants, is sequential — it is even deterministic. A variety of extensions of it have been considered; examples of additional features are references, control operators, non-determinism, (constrained) forms of concurrency. The specific set of syntactic features chosen for the language determines the ways in which the contexts of the language may interact with the terms. In any case, the patterns of interaction are usually more disciplined than those that arise in π-calculus representations of those terms. Thus there are λ-terms that are indistinguishable within the (pure) λ-calculus whose π-calculus images can be separated by appropriate π-contexts.

A well-known way of imposing a discipline to the π-calculus is to equip it with a type system. Such systems are intended to capture communication patterns that occur frequently when programming in the π-calculus. A number of type systems have been considered: e.g., capability types (formalising the intended I/O usage of names that are exchanged among processes), linearity (formalising the property that certain names may be used at most once), session types (formalising the communication protocols in the dialogues between two or more processes), and so on [2], [3], [4], [5]. Type systems have also been designed to capture specific properties of processes, such as termination, deadlock-freedom, lock-freedom [6], [7], [8], [9], [10]. Types impose constraints on the set of legal contexts in which well-typed terms are supposed to be used; this can make behavioural equivalences usefully coarser.

A further step is then to tune the proof techniques of the π-calculus to such type systems, so to be able to actually prove the behavioural equalities that only hold in presence of types. Typically this is investigated in the coinductive setting of bisimilarity, and achieved by refining and/or modifying the standard bisimilarity clauses so to take the usage of types into account. The resulting bisimilarity should be sound with respect to contextually-defined forms of bisimilarity such as *barbed equivalence* (or *congruence*); ideally, it should also be complete.

In barbed equivalence, the bisimulation game is played only on internal actions, and certain success signals, the barbs, are used to monitor the computation. In the standard barbed equivalence, an arbitrary context may be added, once (at the beginning), on top of the tested processes. In *reduction-closed* barbed equivalence [11], [1], the context may be dynamically updated, by adding further components during the computation. Reduction-closed barbed equivalence usually allows simpler proofs of completeness, and does not require any hypothesis of image-finiteness on the state space of the tested processes. In contrast, standard barbed equivalence is more robust — reduction-closed barbed equivalence may sometimes be over-discriminating [12].

In this paper we focus on the π-calculus representation of *sequentiality* and *well-bracketing*. 'Sequentiality' intuitively

978-1-6654-4895-6/21/$31.00 ©2021 IEEE

indicates the existence of a single thread of computation. 'Well-bracketing' is a terminology borrowed from game semantics, and used to refer to a language without control operators, in which the call-return interaction behaviour between a term and its context follows a stack discipline. Our main objectives are to define bisimilarity-based proof techniques for type systems in the π-calculus that formalise the sequentiality and well-bracketing notions. We actually work with the *asynchronous* π-calculus, Aπ, as this is the calculus that is usually adopted in the literature for modelling higher-order languages.

In Aπ, sequentiality is the property that, at any time, at most one process *is active*, or *carries the thread*; that is, the process has the control on the computation and decides what the next computation step can be. In other words, we never find two sub-components of a system both of which contain an *interaction redex* (a pair of an input and an output processes at the same name).

In the (standard) encodings of the λ-calculus [13], [14], a process is active, i.e., it carries the thread, when it contains an unguarded output particle. Indeed, the π-calculus terms obtained from the encodings give rise to computations in which, syntactically, at any time there is at most one unguarded output particle. An input process that consumes that output will in turn become active.

Our type system is more general, in that we allow also input processes to carry the thread. The type system specifies whether a name may carry the thread in output or in input; we call these names *output-controlled* and *input-controlled*. While the output-controlled are the most important ones (for instance, they play a central role in the modelling of functions), input-controlled names may be useful too, for instance, in the representation of references or locks. A reference ℓ that contains the value n is represented in Aπ by an output particle $\overline{\ell}\langle n\rangle$; and a process accessing the reference will do so by performing an input at ℓ. Thus an input at ℓ indicates ownership of the current computation thread.

As remarked above, sequentiality implies absence of parallel computation threads. Sequentiality however does not exclude non-determinism. An output particle $\overline{a}\langle b\rangle$ that owns the thread may have the possibility of interacting with different input processes at a (and symmetrically for input processes owning the thread). Indeed we also admit internal non-determinism (i.e, processes such as $\tau.\,P + \tau.\,Q$ that may chose to reduce either to P or to Q without interactions with the environment), both in active and in inactive processes.

The type system for well-bracketing is a refinement of that for sequentiality, in which a stack of *continuation names* keeps track of the structure of calls and returns among the processes. These stacks are similar to those used in the implementation of compilers for languages (or fragments of languages) adopting well-bracketing, or used in well-bracketed forms of game semantics.

Finding proof techniques to reason about sequentiality and well-bracketing presents a number of caveats, that have mainly to do with the soundness and completeness of the resulting bisimilarity with respect to barbed equivalence. We briefly discuss below a couple of issues concerning completeness.

In the proof of completeness one has to show that the contexts of the language are at least as discriminating as the labelled bisimilarity. In standard proofs, one defines special contexts that interact with the tested processes and, at the same time, emit certain signals to the outside so to provide information on the kind of interactions that have occurred with the processes. Such behaviour of the testing contexts is however inherently concurrent — the context has to interact with the tested processes and, at the same time, emit signals to the outside — and is therefore liable to break the typing discipline for sequentiality (and hence also well-bracketing).

Further problems arise in proofs about reduction-closed barbed equivalence. The reason why completeness proofs for reduction-closed barbed equivalence may be simpler than with standard barbed equivalence is that the testing context may be incrementally adjusted, after every interaction step with the tested processes. This however requires the existence of special components in the contexts to handle the fresh names generated by the tested processes. Specifically, the task of these components is to ensure that new pieces of contexts, added later, will be able to access such fresh names. Again, these components represent parallel threads, and break the sequentiality and well-bracketing disciplines. For this reason in the paper we cannot appeal to reduction-closed forms of barbed equivalence, remaining within the standard notions and therefore requiring an image-finiteness condition.

In the case of well-bracketing the problems above are enhanced by the presence of continuation names. These names are *linear* [3] (they may only be used once), *input receptive* [15] (the input-end of the name should always be available), and output-controlled. This places further constraints on the use of such names within contexts that test the processes.

For the above reasons, the completeness proofs for sequentiality and well-bracketing present significant technical differences, both between them and from completeness proofs in the literature.

In the paper we propose labelled bisimilarities that allow us to reason about processes following the sequentiality or well-bracketing disciplines. We prove that the bisimilarities are sound with respect to barbed equivalence. We also establish completeness, on processes with only output-controlled names. We do not know whether completeness holds in the general case, with also input-controlled names. We also study some refinements of the bisimilarities: one is obtained by injecting ideas from bisimilarities for calculi with references [16]; other refinements are forms of 'up-to techniques'. We illustrate applications of our techniques on a number of examples, most of which have to do with the representation of functions and references. Usually the examples are about equalities that only hold under the sequentiality or well-bracketing disciplines; other examples show that sequentiality and well-bracketing may make equalities simpler to prove because there are fewer observables to take into account.

Paper outline. We introduce some background in Section II.

We study sequentiality in Section III, and well-bracketing in Section IV: in each case, we present our type system, define an appropriate notion of bisimilarity, and show some examples or laws that we can derive. Related and future works are discussed in Section V. For lack of space, some technical definitions and proofs are given in [17].

II. Background: the (asynchronous) π-calculus

We recall here the standard syntax of the asynchronous π-calculus, Aπ, from [18]:

$$\begin{array}{rcl} P, Q & ::= & \overline{a}\langle\widetilde{b}\rangle \mid !a(\widetilde{b}).\,P \mid P \mid Q \mid (\boldsymbol{\nu} a)P \mid G \\ G, G' & ::= & \mathbf{0} \mid a(\widetilde{b}).\,P \mid \tau.\,P \mid [a=b]G \mid G + G' \end{array}$$

Names are ranged over by $a, b, \ldots$. In prefixes $\overline{a}\langle\widetilde{b}\rangle$ and $a(\widetilde{b}).\,P$, name a is the *subject* and $\widetilde{b}$ are the *objects*. We use a tilde, like in $\widetilde{b}$, for (possibly empty) tuples of names; similarly $(\boldsymbol{\nu}\widetilde{a})P$ stands for a sequence of restrictions. As usual, we write $a.\,P$ and $\overline{a}$ when the object of a prefix is the empty tuple. We use $\sum_{i\in I} G_i$ (resp. $\prod_{i\in I} P_i$) for $G_{i_1} + \cdots + G_{i_n}$ (resp. $P_{i_1} \mid \ldots \mid P_{i_n}$) where $I = \{i_1, \ldots, i_n\}$. We write $P\{a/b\}$ for the result of replacing name b with a in P in a capture-avoiding way. Contexts, C, are processes containing a single occurrence of a special constant, the hole (written $[\cdot]$). The *static* contexts, ranged over by E, have the form $\boldsymbol{\nu}\widetilde{a}\,(P \mid [\cdot])$. In examples, for readability we sometimes use basic data values such as integers and booleans. The definition of structural congruence, written $\equiv$, and of the strong and weak labelled transitions, written $\xrightarrow{\mu}$, $\Rightarrow$, and $\overset{\hat{\mu}}{\Rightarrow}$, are standard and are given in [17]. We note $\mathrm{fn}(P)$ (resp. $\mathrm{fn}(\mu)$) the set of free names of P (resp. μ). We sometimes abbreviate reductions $P \xrightarrow{\tau} P'$ as $P \longrightarrow P'$.

The calculi in the paper will be typed. For simplicity we define our type systems as refinements of the most basic type system for π-calculus, namely Milner's *sorting* [19], in which names are partitioned into a collection of *types* (or sorts), and a sorting function maps types onto types. If a name type S is mapped onto a type T, this means that names in S may only carry names in T. We assume that there is a sorting system under which all processes we manipulate are well-typed. We write $\Delta \vdash P$ when process P is well-typed under Δ, and similarly for other objects, such as contexts.

The reference behavioural equivalence for us will be the context-closure of barbed bisimulation. We focus on barbed equivalence (as opposed to barbed congruence) because it is simpler (notably, we do not need to consider issues of closure of the labelled bisimulations under name substitutions). The definition of barbed bisimulation uses the reduction relation $\Longrightarrow$ along with an observation predicate $\Downarrow_a$ for each name a, which detects the possibility of performing an output to the external environment along a. Moreover, since we work in a typed setting, such an output should be allowed by the typing of the tested processes. Thus, we write $\Delta \vDash P \Downarrow_a$ if Δ is a typing for P (i.e., $\Delta \vdash P$ holds), there is an output μ with subject a s.t. $P \overset{\mu}{\Longrightarrow} P'$, and such a transition is observable under the typing Δ. The meaning of 'observable under a typing' will depend on the specific type system adopted; in the case of the plain sorting, all transitions are observable.

Having typed processes, in the definition of barbed equivalence we may only test processes with contexts that respect the typing of the processes.

Definition 1. C is a Γ/Δ *context* if $\Gamma \vdash C$ holds, using the typing for the processes plus the rule $\dfrac{}{\Delta \vdash [\cdot]}$ for the hole.

Similarly, P is a Δ-*process* if $\Delta \vdash P$. We assume (as in usual Subject-Reduction properties for type systems) that typing is invariant under reduction.

Definition 2 (Barbed bisimulation, equivalence, and congruence)**.** *Barbed Δ-bisimulation* is the largest symmetric relation $\dot{\approx}^{\Delta}$ on Δ-processes s.t. $P \dot{\approx}^{\Delta} Q$ implies:

1) whenever $P \longrightarrow P'$ then there exists Q' such that $Q \Longrightarrow Q'$ and $P' \dot{\approx}^{\Delta} Q'$;
2) for each name a, $\Delta \vDash P \Downarrow_a$ iff $\Delta \vDash Q \Downarrow_a$.

Two Δ-processes P and Q are *barbed equivalent at* Δ, written $P \simeq^{\Delta} Q$, if for each Γ/Δ static context E it holds that $E[P] \dot{\approx}^{\Gamma} E[Q]$. *Barbed congruence at* Δ, $\cong^{\Delta}$, is defined in the same way but employing all Γ/Δ contexts (rather than only the static ones).

Barbed equivalence in the plain (untyped) Aπ, $\simeq$, can be proved to coincide with the ordinary labelled early asynchronous bisimilarity, on image-finite processes, exploiting the n-approximants of the labelled equivalences. We recall that the class of *image-finite processes* is the largest subset $\mathcal{I}$ of processes that is derivation closed and s.t. $P \in \mathcal{I}$ implies that, for all μ, the set $\{P' \mid P \overset{\mu}{\Longrightarrow} P'\}$, quotiented by alpha conversion, is finite. In the remainder of the paper, we omit the adjectives 'early' and 'asynchronous' in all bisimilarities.

Definition 3 (Bisimulation)**.** A relation $\mathcal{R}$ on processes is a *bisimulation* if whenever $P \mathrel{\mathcal{R}} Q$ and $P \xrightarrow{\mu} P'$, then one of these two clauses holds:

1) there is Q' such that $Q \overset{\hat{\mu}}{\Rightarrow} Q'$ and $P' \mathrel{\mathcal{R}} Q'$;
2) $\mu = a\langle\widetilde{b}\rangle$ and there is Q' such that $Q \mid \overline{a}\langle\widetilde{b}\rangle \Rightarrow Q'$ and $P' \mathrel{\mathcal{R}} Q'$.

Moreover the converse holds too, on the transitions from Q. *Bisimilarity*, $\approx$, is the largest bisimulation.

Theorem 4 ([18])**.** On image-finite processes, relations $\simeq$ and $\approx$ coincide.

III. Sequentiality

In this section we study sequentiality. We first formalise it by means of a type system, and then we examine its impact on behavioural equivalence.

A. Type system

As mentioned in Section I, intuitively, sequentiality ensures us that at any time at most one interaction can occur in a system; i.e., there is a single computation thread. A process that holds the thread decides what the next interaction can be. It does so by offering a single particle (input or output) that controls the thread. The process may offer multiple particles, but only one of them may control the thread. The control on

the thread attached to a particle is determined by the subject name of that particle. A given name may exercise the control on the thread either in output or in input; in the former case we say that the name is *output-controlled*, in the latter case the name is *input-controlled*. For instance, suppose that x, y, z are output-controlled and u, v are input-controlled. Then the following process correctly manages the thread and will indeed be typable in our type system:

$$P \stackrel{\text{def}}{=} u.\,(\overline{x} \mid y.\,\overline{x}) \mid z.\,\overline{y} \mid \overline{v}$$

The initial particles in P are $u, z, \overline{v}$; however only u controls the thread, as z is output-controlled and v is input-controlled. When the input at u is consumed, the new particles $\overline{x}, y$ are available, where $\overline{x}$ now controls the thread, as both names x, y are output-controlled. An external process that consumes the particle $\overline{x}$ will acquire the control over the thread. For instance, a process such as $Q \stackrel{\text{def}}{=} \overline{u} \mid x.\,Q'$ initially does not hold the thread; in the parallel composition $P \mid Q$, after the two interactions at u and x, the control on the thread will be acquired by Q':

$$P \mid Q \longrightarrow\longrightarrow (y.\,\overline{x} \mid z.\,\overline{y} \mid \overline{v}) \mid Q'$$

Now Q' will decide on the next interaction; for instance, it may offer an output at y or z, or an input at v. It may only offer one of these, though it may offer other particles that do not control the thread.

Notation. *In the remainder, x, y, z range over output-controlled names, u, v, w over input-controlled names; we recall that a, b, c range over the set of all names.*

The name used is therefore an indication of its type. For instance, in $(\boldsymbol{\nu} x)P$, x is output-controlled, and can be only alpha-converted using another output-controlled name.

The type system for sequentiality is presented in Figure 1. Judgements are of the form $\eta \vdash_{\mathrm{s}} P$, for $\eta \in \{0, 1\}$. A judgement $1 \vdash_{\mathrm{s}} P$ indicates that P owns the thread, i.e., P is *active*, and $0 \vdash_{\mathrm{s}} P$ otherwise, i.e., P is *inactive*.

We recall that we only present the additional typing constraints given by sequentiality, assuming the existence of a sorting under which all processes are well-typed (thus the fully-fledged typing judgements would be the form $\Delta; \eta \vdash_{\mathrm{s}} P$, rather than $\eta \vdash_{\mathrm{s}} P$).

Some remarks on the rules in Figure 1: a rule with a double conclusion is an abbreviation for more rules with the same premises but separate conclusions. The continuation of an input always owns control on the thread; the input itself may or may not have the control (rules I-ACT and I-INA). A τ-prefix is neutral w.r.t. the thread. The rule for parallel composition makes sure that the control on the thread is granted to only one of the components; in contrast, in the rule for sum, the control is maintained for both summands. Operators **0** and match cannot own the thread; this makes sure that the thread control is always exercised.

We present some behavioural properties that highlight the meaning of sequentiality. A reduction $P \xrightarrow{\tau} P'$ is an *interaction* if it has been obtained from a communication between an input and an output (formally, its derivation in the LTS of [17] uses rule ACOMM). In a sequential system, one may not find two disjoint interactions.

Proposition 5. Whenever $\eta \vdash_{\mathrm{s}} P$, there exists no $P_1, P_2, \widetilde{a}$ such that $P \equiv (\boldsymbol{\nu}\widetilde{a})(P_1 \mid P_2)$ with $P_1 \xrightarrow{\tau} P_1'$ and $P_2 \xrightarrow{\tau} P_2'$, and both these transitions are interactions.

An inactive process may not perform interactions.

Proposition 6. If $0 \vdash_{\mathrm{s}} P$, then there is no P' with $P \xrightarrow{\tau} P'$ and this transition is an interaction.

An inactive process may however perform τ-reductions, notably to resolve internal choices. In other words such internal choices represent internal matters for a process, orthogonal with respect to the overall interaction thread. The possibility for inactive processes to accommodate internal choices will be important in our completeness proof. However, an inactive process may only perform a finite number of τ-reductions. A process P *is divergent* if it can perform an infinite sequence of reductions, i.e., there are $P_1, P_2, \ldots,$ with $P \longrightarrow P_1 \longrightarrow P_2 \ldots P_n \longrightarrow \ldots$.

Proposition 7. If $0 \vdash_{\mathrm{s}} P$ then P is not divergent.

In contrast, an active process may be divergent, through sequences of reductions containing infinitely many interactions.

Sequentiality imposes constraints on the interactions that a 'legal' (i.e., well-typed) context may undertake with a process. For the definition of barbed bisimulation and equivalence we must therefore define the meaning of observability. The following definition of *type-allowed transitions* shows what such 'legal' interactions can be.

Definition 8 (Type-allowed transitions)**.** We write $\eta \vDash_{\mathrm{s}} P \xrightarrow{\mu} P'$ if $\eta \vdash_{\mathrm{s}} P$, and $P \xrightarrow{\mu} P'$, and one of the following clauses holds:

1) $\eta = 0$
2) $\mu = \tau$
3) $\eta = 1$ and $\mu = u\langle\widetilde{a}\rangle$ for some $u, \widetilde{a}$ or $\mu = (\boldsymbol{\nu}\widetilde{a})\overline{x}\langle\widetilde{b}\rangle$ for some $\widetilde{a}, x, \widetilde{b}$.

Clause (1) says that all interactions between an inactive process and the context are possible; this holds because the context is active and may therefore decide on the next interaction with the process. Clause (2) says that internal reductions may always be performed. Clause (3) says that the only visible actions observable in active processes are those carrying the thread; this holds because the observer is inactive, and it is therefore up to the process to decide on the next interaction.

We now examine how typing evolves under legal actions. We recall that x stands for an output-controlled name.

Definition 9. We write $[\eta; P] \xrightarrow{\mu} [\eta'; P']$ when $\eta \vDash_{\mathrm{s}} P \xrightarrow{\mu} P'$ and:

1) if $\mu = x\langle\widetilde{a}\rangle$, then $\eta' = 1$.
2) if $\mu = (\boldsymbol{\nu}\widetilde{a})\overline{x}\langle\widetilde{b}\rangle$, then $\eta' = 0$.
3) otherwise $\eta' = \eta$.

$$\textsc{I-Act}\ \frac{1 \vdash_{\rm s} P}{1 \vdash_{\rm s} u(\widetilde{a}).\,P} \qquad \textsc{I-Ina}\ \frac{1 \vdash_{\rm s} P}{0 \vdash_{\rm s} x(\widetilde{a}).\,P,\ !x(\widetilde{a}).\,P} \qquad \textsc{O-Act}\ \frac{}{1 \vdash_{\rm s} \overline{x}\langle\widetilde{a}\rangle} \qquad \textsc{O-Ina}\ \frac{}{0 \vdash_{\rm s} \overline{u}\langle\widetilde{a}\rangle} \qquad \textsc{Res}\ \frac{\eta \vdash_{\rm s} P}{\eta \vdash_{\rm s} (\boldsymbol{\nu} a)P}$$

$$\textsc{Nil}\ \frac{}{0 \vdash_{\rm s} \mathbf{0}} \qquad \textsc{Par}\ \frac{\eta_1 \vdash_{\rm s} P \qquad \eta_2 \vdash_{\rm s} Q}{\eta_1 + \eta_2 \vdash_{\rm s} P \mid Q}\ \eta_1 + \eta_2 \leq 1 \qquad \textsc{Sum}\ \frac{\eta \vdash_{\rm s} G_1 \qquad \eta \vdash_{\rm s} G_2}{\eta \vdash_{\rm s} G_1 + G_2} \qquad \textsc{Tau}\ \frac{\eta \vdash_{\rm s} P}{\eta \vdash_{\rm s} \tau.\,P} \qquad \textsc{Mat}\ \frac{0 \vdash_{\rm s} G}{0 \vdash_{\rm s} [a=b]G}$$

Fig. 1. The typing rules for sequentiality

Theorem 10 (Subject Reduction)**.** If $\eta \vdash_{\rm s} P$ and $[\eta; P] \xrightarrow{\mu} [\eta'; P']$ then $\eta' \vdash_{\rm s} P'$.

Weak type-allowed transition are defined as expected, exploiting the invariance of typing under reductions: $\eta \vDash_{\rm s} P \overset{\mu}{\Longrightarrow} P'$ holds if there are P_0, P_1 with $P \Longrightarrow P_0$, $\eta \vDash_{\rm s} P_0 \xrightarrow{\mu} P_1$ and $P_1 \Longrightarrow P'$.

B. Behavioural equivalence

To tune Definition 2 of barbed bisimulation and equivalence to the setting of sequentiality, we have to specify the meaning of observables. An observable $\eta \vDash_{\rm s} P \Downarrow_a$ holds if there are P' and an output action μ such that $\eta \vDash_{\rm s} P \overset{\mu}{\Longrightarrow} P'$ and the subject of μ is a. Following Definition 1, in barbed equivalence, the legal contexts are the η/η' static contexts. We write barbed equivalence at η as $\simeq^{\eta}$. Thus $P \simeq^{\eta} Q$ holds if $\eta \vdash_{\rm s} P, Q$ and $E[P] \dot{\approx}^{\eta'} E[Q]$, for any η' and any η'/η static context E.

We are now ready to define the labelled bisimilarity to be used on sequential processes, which is our main proof technique for barbed equivalence. A *typed process relation* is a set of triplets (η, P, Q) with $\eta \vdash_{\rm s} P, Q$.

Definition 11 (Sequential Bisimulation)**.** A typed process relation $\mathcal{R}$ is a *sequential bisimulation* if whenever $(\eta, P, Q) \in \mathcal{R}$ and $[\eta; P] \xrightarrow{\mu} [\eta'; P']$, then one of the two following clauses holds:

1) there is Q' such that $Q \overset{\hat{\mu}}{\Longrightarrow} Q'$ and $(\eta', P', Q') \in \mathcal{R}$;
2) $\mu = a\langle\widetilde{b}\rangle$ and there is Q' such that $Q \mid \overline{a}\langle\widetilde{b}\rangle \Rightarrow Q'$ with $(\eta', P', Q') \in \mathcal{R}$.

Moreover, the converse of (1) and (2) holds on the transitions from Q. Processes P and Q are *sequentially bisimilar at* η, written $P \approx_{\rm s}^{\eta} Q$, if $(\eta, P, Q) \in \mathcal{R}$ for some sequential bisimulation $\mathcal{R}$.

In clause (2), $Q \mid \overline{a}\langle b\rangle$ is well-typed, be a an input- or output-controlled name. Clauses (1) and (2) are the same as for ordinary bisimilarity $\approx$ (Definition 3); typing however prevents certain transitions to be considered as challenge transitions in the bisimulation game. Thus the resulting bisimilarity becomes coarser.

Ordinary bisimilarity is included in the sequential one (the inclusion is strict, see Section III-C).

Proposition 12. For $\eta \vdash_{\rm s} P, Q$, if $P \approx Q$ then also $P \approx_{\rm s}^{\eta} Q$.

Theorem 13 (Soundness)**.** If $P \approx_{\rm s}^{\eta} Q$, then $P \simeq^{\eta} Q$.

As usual, the proof of Theorem 13 relies on the preservation of $\approx_{\rm s}^{\eta}$ under parallel composition, which requires some care in order to enforce sequentiality. This is ensured by typability. Theorem 13 allows us to use the labelled bisimilarity $\approx_{\rm s}^{\eta}$ as a proof technique for typed barbed equivalence.

This proof technique is also complete, assuming only output-controlled names (i.e., the thread may only be exercised by output particles, not by the input ones).

Theorem 14 (Completeness on output-controlled names)**.** For all image-finite processes P, Q that only use output-controlled names, and for all η, if $P \simeq^{\eta} Q$ then $P \approx_{\rm s}^{\eta} Q$.

The completeness proof can be found in [17]. While the overall structure of the proof is standard, the technical details are specific to sequentiality. As usual, we rely on a stratification of bisimilarity and approximants $\approx_{\rm s}^{\eta,n}$, and reason by contradiction to show that if $\eta \vdash_{\rm s} P, Q$ and $P \not\approx_{\rm s}^{\eta,n} Q$, then there is a η'/η static context E such that $E[P] \not\dot{\approx}^{\eta'} E[Q]$. The case $\eta = 0$ (tested processes are inactive) is rather standard: the context E is of the form $(\boldsymbol{\nu}\widetilde{x})([\cdot] \mid \overline{z} \mid z.\,R)$, for some fresh z, and some "tester process" R. The barb at z allows us to detect when the tested process interacts with R.

The delicate case is when $\eta = 1$ (tested processes are active): the context must be inactive and hence cannot have an unguarded output at z. We use in this case a context of the form $E \stackrel{\rm def}{=} (\boldsymbol{\nu}\widetilde{x})([\cdot] \mid G_R + G)$. Process G_R is the tester process, and G is $\sum_{y \in S} y(\widetilde{y'}).\,\overline{z}$, defined for some fresh z and some set S containing $\mathrm{fn}(P) \cup \mathrm{fn}(Q)$. G satisfies the following property: for any P_0 and for any x, if $1 \vDash_{\rm s} P_0 \Downarrow_{\overline{x}}$ then $1 \vDash_{\rm s} P_0 \mid G \Downarrow_{\overline{z}}$. Thus, as soon as P_0 exhibits some barb, we have $1 \vDash_{\rm s} E[P_0] \Downarrow_{\overline{z}}$, and P_0 cannot interact with R without removing the barb at z, which allows us to reason as in the case $\eta = 0$.

The proof schema above does not apply if input-controlled names are allowed, intuitively because in this case the processes being tested may be active and perform an input (at an input-controlled name), thus maintaining the thread; both before and after the transition the testing context is passive and hence unable to signal, with appropriate barbs, which interaction occurred.

C. Examples

With respect to ordinary bisimilarity, in sequential bisimilarity ($\approx_{\rm s}^{\eta}$) fewer challenges are allowed. This may both mean that certain processes, otherwise distinguishable, become equal, and that certain equalities are simpler to prove because the state space of the processes to be examined is reduced. We present some equalities of the first kind (valid for $\approx_{\rm s}^{\eta}$ only).

In Section III-C2, we also show a refinement of $\approx_{\mathrm{s}}^{\eta}$ useful for reasoning about references.

1) Basic examples: In the type system, $\mathbf{0}$ is inactive —- without the thread. We write $\mathbf{0}_1$ to abbreviate $(\boldsymbol{\nu} x)(\overline{x})$ (an active process without transitions).

Example 15. While a component of a system is active, other components cannot be observed. Thus, if the active component keeps the thread, the existence of other components is irrelevant. Indeed we have, for any R, Q inactive:

$$R \mid \mathbf{0}_1 \approx_{\mathrm{s}}^{1} R \mid (\boldsymbol{\nu} x)(\overline{x} \mid !x.\,\overline{x}) \approx_{\mathrm{s}}^{1} Q \mid (\boldsymbol{\nu} x)(\overline{x} \mid !x.\,\overline{x}) \approx_{\mathrm{s}}^{1} \mathbf{0}_1$$

Example 16. An unguarded occurrence of an input at an input-controlled name becomes the only observation that can be made in a process. This yields the following equalities

$$\begin{array}{rcll} u.\,P \mid x.\,Q & \approx_{\mathrm{s}}^{1} & u.\,(P \mid x.\,Q) & \\ u.\,P \mid \overline{v} & \approx_{\mathrm{s}}^{1} & u.\,(P \mid \overline{v}) & \qquad \text{for } u \neq v \end{array}$$

Example 17. Consider the process

$$P \stackrel{\text{def}}{=} (\boldsymbol{\nu} y', z')(!x.\,(z'.\,\overline{z} \mid \overline{y'}) \mid !y.\,\overline{z'}).$$

The output at z becomes observable if both an input at x and an input at y are consumed, so that the internal reduction at z' can take place. However the input at x acquires the thread, preventing a further immediate input at y; similarly for the input at y. Indeed we have $P \approx_{\mathrm{s}}^{0} x.\,\mathbf{0}_1 + y.\,\mathbf{0}_1$.

2) Examples with references: We now consider a few examples involving references. For this, we use the standard encoding of references into Aπ, and we enhance the bisimilarity for sequentiality so to take references into account.

We use $n, m, \ldots$ to range over the entities stored in references (which can be names or values belonging to a first-order data type like booleans and integers) and placeholders for them. Name ℓ is used to represent a reference.

In Aπ, a reference ℓ holding a value n is represented as an output particle $\overline{\ell}\langle n\rangle$. A process that contains a reference ℓ should have, at any time, exactly one unguarded output at ℓ, meaning that at any time the reference has a unique value. We say that in this case ℓ *is accessible*. The read and write operations on ℓ are written as follows:

$$\begin{array}{rcl} \mathtt{re}_{\ell}(m).\,R & \stackrel{\text{def}}{=} & \ell(m).\,(\overline{\ell}\langle m\rangle \mid R) \\ \mathtt{wr}_{\ell}\langle n\rangle.\,R & \stackrel{\text{def}}{=} & \ell(m').\,(\overline{\ell}\langle n\rangle \mid R) \quad \text{for } m' \notin \mathrm{fn}(R) \end{array}$$

Thus a name ℓ used to encode a reference is input-controlled, as an action on a reference is represented by an input at ℓ — we use ℓ rather than $u, v, \ldots$ to stress the fact that names used to represent references obey constraints that go beyond input-control.

Proof techniques for the representation of references in Aπ have been studied in [16]. Adopting them requires enhancing our type system with information about references, which simply consists in declaring which names represent references. In the definition of barbed equivalence, the main constraint is that the tested context should make sure that all existing reference names are accessible. To reason about references, several definitions of labelled bisimilarity are presented in [16], varying on the forms of constraints imposed on transitions. Here we only import the simplest such constraint: it forbids observations of input transitions $P \stackrel{\ell\langle n\rangle}{\longrightarrow} P'$ at a reference name ℓ when ℓ is accessible in P (i.e., an unguarded output at ℓ occurs in P). Such a constraint represents the fact that an observer may not pretend to own a reference when the reference is accessible in the process.

Formally, with the addition of references, judgements in the type system become of the form $S; \eta \vdash_{\mathrm{s}} P$, where S is a finite set of reference names, meaning that $\eta \vdash_{\mathrm{s}} P$ holds and that S is the set of accessible reference names in P. The definition of type-allowed transitions, $S; \eta \vDash_{\mathrm{s}} P \stackrel{\mu}{\rightarrow} P'$, is the same as before (Definition 8) with the addition, in clause (3), of the constraint

$$\text{if } \mu \text{ is an input } \ell\langle n\rangle \text{ at a reference name } \ell \text{ then } \ell \notin S. \quad (*)$$

Finally the definition of *sequential bisimilarity with references at* (S, η), written $\approx_{\mathrm{s}}^{S;\eta}$ is the same as that of sequential bisimilarity (Definition 11), just using $S; \eta \vdash_{\mathrm{s}} P, Q$ and $S; \eta \vDash_{\mathrm{s}} P \stackrel{\mu}{\rightarrow} P'$ in place of $\eta \vdash_{\mathrm{s}} P, Q$ and $\eta \vDash_{\mathrm{s}} P \stackrel{\mu}{\rightarrow} P'$.

It is straightforward to extend the soundness proof for sequential bisimilarity w.r.t. barbed equivalence (Theorem 13) to the case of sequential bisimilarity with references.

Example 18. This example shows that reading or writing on a global reference is not subject to interferences from the outside, as these operations require the thread:

$$\begin{array}{lcl} \overline{\ell}\langle n\rangle \mid \mathtt{re}_{\ell}(m).\,R & \approx_{\mathrm{s}}^{\ell;1} & \overline{\ell}\langle n\rangle \mid R\{n/m\} \\ \overline{\ell}\langle n\rangle \mid \mathtt{wr}_{\ell}\langle m\rangle.\,R & \approx_{\mathrm{s}}^{\ell;1} & \overline{\ell}\langle m\rangle \mid R \end{array}$$

Indeed, in each law, if P (resp. Q) is the process on the left-hand (resp. right-hand) side, then the relation $\{((\ell; 1), P, Q)\} \cup \mathcal{I}$ is a sequential bisimulation, when taking the constraint $(*)$ for references into account.

Example 19 (Fetch-and-add, swap)**.** We consider fetch-and-add and swap operations, often found in operating systems. The first, written $\mathtt{faa}_{\ell}\langle n\rangle(m)$ atomically increments by n the content of the reference ℓ, and returns the original value as m; the second, written $\mathtt{sw}_{\ell}\langle n\rangle(m)$, atomically sets the content of ℓ to n and returns the original value as m:

$$\begin{array}{rcl} \mathtt{faa}_{\ell}\langle n\rangle(m).\,R & \stackrel{\text{def}}{=} & \ell(m).\,(\overline{\ell}\langle m+n\rangle \mid R) \\ \mathtt{sw}_{\ell}\langle n\rangle(m).\,R & \stackrel{\text{def}}{=} & \ell(m).\,(\overline{\ell}\langle n\rangle \mid R) \end{array}$$

These operations may be mimicked by a combination of read and write operations (we take $m' \notin \mathrm{fn}(R)$):

$$\begin{array}{rcl} \mathtt{faa2}_{\ell}\langle n\rangle(m).\,R & \stackrel{\text{def}}{=} & \mathtt{re}_{\ell}(m).\,\mathtt{wr}_{\ell}\langle m+n\rangle.\,R \\ & = & \ell(m).\,(\overline{\ell}\langle m\rangle \mid \ell(m').\,(\overline{\ell}\langle m+n\rangle \mid R)) \\ \mathtt{sw2}_{\ell}\langle n\rangle(m).\,R & \stackrel{\text{def}}{=} & \mathtt{re}_{\ell}(m).\,\mathtt{wr}_{\ell}\langle n\rangle.\,R \\ & = & \ell(m).\,(\overline{\ell}\langle m\rangle \mid \ell(m').\,(\overline{\ell}\langle n\rangle \mid R)) \end{array}$$

For this mimicking to be correct, sequentiality is necessary. To see this, consider the simple case when $R \stackrel{\text{def}}{=} \overline{c}\langle m\rangle$. In the ordinary A$\pi$, processes $\mathtt{sw}_{\ell}\langle n\rangle(m).\,R$ and $\mathtt{sw2}_{\ell}\langle n\rangle(m).\,R$ are distinguished, intuitively because the observer is capable

of counting the two inputs and the two outputs at ℓ in $\mathtt{sw2}_\ell\langle n\rangle(m).\,R$ (against only one in $\mathtt{sw}_\ell\langle n\rangle(m).\,R$) and/or is capable of detecting the output $\overline{\ell}\langle m\rangle$ in $\mathtt{sw2}_\ell\langle n\rangle(m).\,R$.

The processes are also distinguished with the proof techniques for references in [16], intuitively because, after the initial input $\ell(m)$ (whereby the processes read the content of the reference), an observer may interact with the derivative of $\mathtt{sw2}_\ell\langle n\rangle(m).\,R$ and use its output $\overline{\ell}\langle m\rangle$ so to know the value that had been read. Such an observation is not possible with $\mathtt{sw}_\ell\langle n\rangle(m).\,R$.

In contrast, the two processes are equal if we take sequentiality into account. That is, we have:

$$\mathtt{sw}_\ell\langle n\rangle(m).\,R \approx_{\mathrm{s}}^{\emptyset;1} \mathtt{sw2}_\ell\langle n\rangle(m).\,R$$

This is proved by showing that the relation

$$\begin{array}{l}\cup_{m'}\{((\ell;1),\overline{\ell}\langle n\rangle \mid R\{^{m'}/_{m}\},\ \overline{\ell}\langle m'\rangle \mid \mathtt{wr}_\ell\langle n\rangle.\,R\{^{m'}/_{m}\})\\ \cup\,\mathcal{I}\ \cup\ \{((\emptyset;1),\ \mathtt{sw}_\ell\langle n\rangle(m).\,R,\ \mathtt{sw2}_\ell\langle n\rangle(m).\,R)\}\end{array}$$

is a sequential bisimulation. The equivalence between $\mathtt{faa}_\ell\langle n\rangle(m).\,R$ and $\mathtt{faa2}_\ell\langle n\rangle(m).\,R$ is established using a similar relation.

Example 20 (Optimised access)**.** Two consecutive read and/or write operations can be transformed into an equivalent single operation.

$$\begin{array}{rcl}\mathtt{wr}_\ell\langle n\rangle.\,\mathtt{wr}_\ell\langle m\rangle.\,R & \approx_{\mathrm{s}}^{\emptyset;1} & \mathtt{wr}_\ell\langle m\rangle.\,R\\ \mathtt{wr}_\ell\langle n\rangle.\,\mathtt{re}_\ell(m).\,R & \approx_{\mathrm{s}}^{\emptyset;1} & \mathtt{wr}_\ell\langle n\rangle.\,R\{^{n}/_{m}\}\\ \mathtt{re}_\ell(m).\,\mathtt{re}_\ell(m').\,R & \approx_{\mathrm{s}}^{\emptyset;1} & \mathtt{re}_\ell(m).\,R\{^{m}/_{m'}\}\end{array}$$

For the first equality, one shows that

$$\begin{array}{l}\mathcal{I}\cup\{((\emptyset;1),\mathtt{wr}_\ell\langle n\rangle.\,\mathtt{wr}_\ell\langle m\rangle.\,R,\ \mathtt{wr}_\ell\langle m\rangle.\,R)\}\\ \quad\cup\{((\ell;1),\overline{\ell}\langle n\rangle \mid \mathtt{wr}_\ell\langle m\rangle.\,R,\ \overline{\ell}\langle m\rangle \mid R)\}\end{array}$$

is a sequential bisimulation. The second law is treated similarly. In both cases, the relation exhibited is finite.

For the third equality, one defines $\mathcal{R}$ as

$$\cup_n\left\{((\ell;1),\overline{\ell}\langle n\rangle \mid \mathtt{re}_\ell(m').\,R\{^{n}/_{m}\},\overline{\ell}\langle n\rangle \mid R\{^{n,n}/_{m,m'}\})\right\}$$

Then $\{((\emptyset;1),\mathtt{re}_\ell(m).\,\mathtt{re}_\ell(m').\,R,\quad \mathtt{re}_\ell(m).\,R\{^{m}/_{m'}\})\}$ $\cup\,\mathcal{R}\ \cup\ \mathcal{I}$ is a sequential bisimulation.

IV. Well-bracketing

A. Type System

We now go beyond sequentiality, so to handle well-bracketing. In languages without control operators, this means that return-call interactions among terms follow a stack-based discipline.

Intuitively, a well-bracketed system is a sequential system offering services. When interrogated, a service, say A, acquires the thread and is supposed to return a final result (unless the computation diverges) thus releasing the thread. During its computation, A may however interrogate another service, say B, which, upon completion of its computation, will return the result to A. In a similar manner, the service B, during its computation, may call yet another service C, and will wait for the return from C before resuming its computation. B may also delegate to C the task of returning a result to A. In any case, the 'return' obligation may not be thrown away or duplicated.

The implementation of this policy requires *continuation names*. For instance, when calling B, process A transmits a fresh name, say p, that will be used by B (or other processes delegated by B) to return the result to A. Moreover, A waits for such a result, via an input at p. Therefore continuation names are *linear* [3] — they may only be used once — and *input receptive* [15] — the input-end of the name must be made available as soon as the name is created; and they are output-controlled: they carry the thread in output.

In short, the 'well-bracketing' type system defined in this section refines the type discipline for sequentiality by adding linear-receptive names and enforcing a stack discipline on the usage of such names. Proof techniques for well-bracketing will be studied in Section IV-D.

Thus, with well-bracketing, we have three kinds of names: output-controlled names (ranged over by $x, y, z, \ldots$) and input-controlled names (ranged over by $u, v, w\ldots$), as in the previous section; and continuation names, ranged over by $p, q, r\ldots$. As before, names $a, b, c\ldots$ range over the union of output- an input-controlled names.

Continuation names may only be sent at output-controlled names. Indeed, any output at an output-controlled name must carry exactly one continuation name. Without this constraint the type system for well-bracketing would be more complex, and it is unclear whether it would be useful in practice. By convention, we assume that, in a tuple of names transmitted over an output-controlled name, the last name is a continuation name. We write $\widetilde{a}, p$ for such a tuple of names.

The type system is presented in Figure 2. Judgements are of the form

$$\sigma \vdash_{\mathrm{wb}} P$$

where σ is a *stack*, namely a sequence of *input-* and *output-tagged* continuation names, in which the input and output tags alternate, always terminating with an output tag unless the sequence is empty:

$$\begin{array}{rcl}\sigma & ::= & \sigma_{\mathtt{O}} \mid \sigma_{\mathtt{I}}\\ \sigma_{\mathtt{O}} & ::= & p:\mathtt{o},\sigma_{\mathtt{I}}\qquad \sigma_{\mathtt{I}} \ ::= \ p:\mathtt{i},\sigma_{\mathtt{O}} \mid \emptyset\end{array}$$

Moreover: a name may appear at most once with a given tag; and, if a name appears with both tags, then the input occurrence should immediately follow the output occurrence, as for p in $p' : \mathtt{i}, p : \mathtt{o}, p : \mathtt{i}, \sigma$. We write $p \in \sigma$ if name p appears in σ, and $|\sigma|$ for the length of the sequence σ.

Intuitively, a stack expresses the expected usage of the free continuation names in a process. For instance, if

$$p_1 : \mathtt{o}, p_2 : \mathtt{i}, p_3 : \mathtt{o}, p_3 : \mathtt{i}, p_4 : \mathtt{o} \vdash_{\mathrm{wb}} P$$

then $p_1, \ldots, p_4$ are the free continuation names in P; among these, p_1 will be used first, in an output (p_1 may be the subject or an object of the output); then p_2 will be used, in an input interaction with the environment. P possesses both the output

$$\textsc{wb-Out1}\,\frac{}{p:\mathtt{o}\vdash_{\mathrm{wb}}\overline{p}\langle\widetilde{a}\rangle}\qquad \textsc{wb-Out2}\,\frac{}{p:\mathtt{o}\vdash_{\mathrm{wb}}\overline{x}\langle\widetilde{a},p\rangle}\qquad \textsc{wb-Out3}\,\frac{}{\emptyset\vdash_{\mathrm{wb}}\overline{u}\langle\widetilde{a}\rangle}$$

$$\textsc{wb-Inp1}\,\frac{p:\mathtt{o}\vdash_{\mathrm{wb}}P\qquad p\neq q}{q:\mathtt{i},p:\mathtt{o}\vdash_{\mathrm{wb}}q(\widetilde{a}).\,P}\qquad \textsc{wb-Inp2}\,\frac{p:\mathtt{o}\vdash_{\mathrm{wb}}P}{\emptyset\vdash_{\mathrm{wb}}x(\widetilde{a},p).\,P,\ !x(\widetilde{a},p).\,P}\qquad \textsc{wb-Inp3}\,\frac{p:\mathtt{o}\vdash_{\mathrm{wb}}P}{p:\mathtt{o}\vdash_{\mathrm{wb}}u(\widetilde{a}).\,P}\qquad \textsc{wb-Nil}\,\frac{}{\emptyset\vdash_{\mathrm{wb}}\mathbf{0}}$$

$$\textsc{wb-Res1}\,\frac{\xi,p:\mathtt{o},p:\mathtt{i},\sigma'\vdash_{\mathrm{wb}}P}{\xi,\sigma'\vdash_{\mathrm{wb}}(\boldsymbol{\nu}p)P}\qquad \textsc{wb-Res2}\,\frac{\sigma\vdash_{\mathrm{wb}}P}{\sigma\vdash_{\mathrm{wb}}(\boldsymbol{\nu}p)P}\ p\notin\sigma\qquad \textsc{wb-Res3}\,\frac{\sigma\vdash_{\mathrm{wb}}P}{\sigma\vdash_{\mathrm{wb}}(\boldsymbol{\nu}a)P}\qquad \textsc{wb-Mat}\,\frac{\emptyset\vdash_{\mathrm{wb}}P}{\emptyset\vdash_{\mathrm{wb}}[a=b]P}$$

$$\textsc{wb-Par}\,\frac{\sigma\vdash_{\mathrm{wb}}P\qquad \sigma'\vdash_{\mathrm{wb}}Q}{\sigma''\vdash_{\mathrm{wb}}P\mid Q}\,\sigma''\in\mathtt{inter}(\sigma;\sigma')\qquad \textsc{wb-Tau}\,\frac{\sigma\vdash_{\mathrm{wb}}P}{\sigma\vdash_{\mathrm{wb}}\tau.\,P}\ |\sigma|\leq 1\qquad \textsc{wb-Sum}\,\frac{\sigma\vdash_{\mathrm{wb}}P\qquad \sigma\vdash_{\mathrm{wb}}Q}{\sigma\vdash_{\mathrm{wb}}P+Q}\ |\sigma|\leq 1$$

Fig. 2. Type system for well-bracketing

and the input capability on p_3, and may use both capabilities by performing a reduction at p_3; or P may transmit the output capability and then use the input one; the computation for P terminates with an output at p_4. This behaviour however concerns only the free continuation names of P: at any time when an output usage is expected, P may decide to create a new continuation name and send it out, maintaining its input end. The Subject Reduction Theorem 25 will formalise the behaviour concerning continuations names in stacks.

As simple examples of typing, we can derive

$$p:\mathtt{o}\vdash_{\mathrm{wb}}\overline{p}\langle a\rangle\quad\text{and}\quad p:\mathtt{o}\vdash_{\mathrm{wb}}u(a).\,\overline{p}\langle a\rangle$$

In the latter typing, by rule WB-INP3, an input at an input-controlled name has the thread, and does not affect the stack because u is not a continuation name.

The same stack can be used to type a process that invokes a service at x before sending the result at p, as in

$$p:\mathtt{o}\vdash_{\mathrm{wb}}(\boldsymbol{\nu}q)(\overline{x}\langle b,q\rangle\mid q(c).\,\overline{p}\langle c\rangle)$$

where q is a fresh continuation name created when calling x. To type the process without the restriction at q, the stack should mention the input and output capabilities for q:

$$q:\mathtt{o},q:\mathtt{i},p:\mathtt{o}\vdash_{\mathrm{wb}}\overline{x}\langle b,q\rangle\mid q(c).\,\overline{p}\langle c\rangle$$

For another example, the process

$$P_0\stackrel{\text{def}}{=}p(a).\,\overline{p'}\langle a\rangle\mid q(b).\,\overline{q'}\langle b\rangle$$

can be typed using two stacks: we have both $p:\mathtt{i},p':\mathtt{o},q:\mathtt{i},q':\mathtt{o}\vdash_{\mathrm{wb}}P_0$ and $q:\mathtt{i},q':\mathtt{o},p:\mathtt{i},p':\mathtt{o}\vdash_{\mathrm{wb}}P_0$. The choice of the stack depends on whether the call answered at p has been made before or after the call answered at q.

We comment on the rules of the type system. In WB-OUT1 and WB-OUT2 the obligation in the stack is fulfilled (the output capability on the only name in the stack is used in WB-OUT1 and transmitted in WB-OUT2). As explained above, the last name in the tuple transmitted at x is a continuation name, and the only one being transmitted (to enforce the stack discipline). In contrast, in WB-OUT3 an output at an input-controlled name does not own the thread and therefore may not carry continuation names. In WB-INP1 the input-tagged name on top of the stack is used. Rule WB-INP2 is the complement of WB-OUT2. In WB-INP3, an input at an input-controlled name maintains the thread. In all rules for input and τ prefixes, the stack in the premise of the rules may not contain input-tagged continuation names because their input capability must be unguarded (as they are receptive names). The same occurs in rule WB-SUM, following [15] where choice on inputs at receptive names is disallowed (though the constraint could be relaxed). Matching is allowed on plain names, but not on continuation names; this is typical of type systems where the input and output capabilities on names are separate [1]; moreover, no continuation name may appear in the process underneath, to make sure that the obligations on continuation names are not eschewed. In WB-RES1 a continuation name is created, and then its output and input capabilities are inserted into the stack. In the rule, ξ,σ is a decomposition of the stack for $\boldsymbol{\nu}p\,P$ where σ is a stack beginning with an output tag; hence ξ is either empty or it is of the form $\sigma',p:\mathtt{i}$, (i.e. ξ is an initial prefix of the stack, either empty or ending with an input tag).

Rule WB-RES2 is for continuation names that do not appear in the body of the restriction (this form of rule is common in type systems for linearity, to simplify the assertion of Subject Reduction). In rule WB-PAR, the typing stack is split to type the two process components P_1 and P_2; splitting of the typing is usual in type systems with linearity. Here, however, the split must respect the order of the names. That is, the stack in the conclusion should be an interleaving of the two stacks in the premises, as by the following definition.

Definition 21 (Interleaving)**.** We write $\sigma_1\in\mathtt{inter}(\sigma_2;\sigma_3)$ if (i) σ_1 is a stack, and (ii) σ_1 is an interleaving of σ_2 and σ_3 as by the following inductive rules:

1) $\emptyset\in\mathtt{inter}(\emptyset;\emptyset)$
2) $p:\mathtt{o},\sigma_1\in\mathtt{inter}(\sigma_2;\mathtt{p}:\mathtt{o},\sigma_3)$ if $\sigma_1\in\mathtt{inter}(\sigma_2;\sigma_3)$
3) $p:\mathtt{o},\sigma_1\in\mathtt{inter}(\mathtt{p}:\mathtt{o},\sigma_2;\sigma_3)$ if $\sigma_1\in\mathtt{inter}(\sigma_2;\sigma_3)$
4) the same as (2) and (3) with $p:\mathtt{i}$ instead of $p:\mathtt{o}$

If a name appears both in σ_2 and in σ_3 with the same tag, then $\mathtt{inter}(\sigma_2;\sigma_3)$ may not contain any stack.

Being stack-ordered means that $p.\,\overline{q}\mid q.\,\overline{p}$ cannot be typed.

Indeed, the left process would require p before q in the stack, whereas the right process needs the opposite.

In rule RES1, having the possibility to add names in the middle of the stack is mandatory to preserve typability after reduction. Consider for instance:

$(\boldsymbol{\nu} q)((\boldsymbol{\nu} p)(\overline{b}\langle p\rangle \mid p.\,\overline{q}) \mid q.\,\overline{p'}) \xrightarrow{(\boldsymbol{\nu} p)\overline{b}\langle p\rangle} (\boldsymbol{\nu} q)(p.\,\overline{q} \mid q.\,\overline{p'})$ To type the derivative of the transition above, we have to use rule RES1 with $\xi = p : \mathtt{i}$ and $\sigma' = p' : \mathtt{o}$.

Typability in the type system of Figure 2 implies typability in the type system for sequentiality. Indeed, when $\sigma \vdash_{\mathrm{wb}} P$, if the first name in σ is output-tagged then P is active, otherwise P is inactive. We write seq for the function that 'forgets' the well-bracketing information in a stack, therefore $\mathsf{seq}(\sigma) = 1$ if $\sigma = p : \mathtt{o}, \sigma'$, for some p and σ', and $\mathsf{seq}(\sigma) = 0$ otherwise.

Proposition 22. If $\sigma \vdash_{\mathrm{wb}} P$ then also $\mathsf{seq}(\sigma) \vdash_{\mathrm{s}} P$.

In Definition 23, we extend type-allowed transitions to processes with continuation names. As previously, we must ensure that the process is typed, and that the transition is allowed by sequentiality (clauses (1) and (2) below). Clause (3) says that the first continuation name observed must be on top of the stack, and that the input or output capability on a continuation name may not be exercised by the environment when both capabilities are owned by the process.

Definition 23. We write $\sigma \vDash_{\mathrm{wb}} P \xrightarrow{\mu} P'$ when

1) $\sigma \vdash_{\mathrm{wb}} P$
2) $\mathsf{seq}(\sigma) \vDash_{\mathrm{s}} P \xrightarrow{\mu} P'$ and
3) if $p \in \mathrm{fn}(\mu)$ and $p \in \sigma$, then either $\sigma = p : \mathtt{o}, \sigma'$ or $\sigma = p : \mathtt{i}, \sigma'$ for some σ'; moreover, if $p \in \sigma'$, then p is not the subject of μ.

We exploit type-allowed transitions to define transitions with stacks, which make explicit the evolution of the stack.

Definition 24. We note $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$ when $\sigma \vDash_{\mathrm{wb}} P \xrightarrow{\mu} P'$ and

1) if $\mu = (\boldsymbol{\nu}\widetilde{b})\overline{p}\langle\widetilde{a}\rangle$, then $\sigma = p : \mathtt{o}, \sigma'$
2) if $\mu = p\langle\widetilde{a}\rangle$, then $\sigma = p : \mathtt{i}, \sigma'$
3) if $\mu = (\boldsymbol{\nu}\widetilde{c}, p)\overline{a}\langle\widetilde{b}, p\rangle$, then $\sigma' = p : \mathtt{i}, \sigma$
4) if $\mu = (\boldsymbol{\nu}\widetilde{c})\overline{a}\langle\widetilde{b}, p\rangle$, then $\sigma = p : \mathtt{o}, \sigma'$
5) if $\mu = a\langle\widetilde{b}, p\rangle$, then $\sigma' = p : \mathtt{o}, \sigma$
6) if $\mu = \tau$, then for $\sigma = p : \mathtt{o}, p : \mathtt{i}, \sigma''$ and $p \notin \mathrm{fn}(P')$, we have $\sigma' = \sigma''$, otherwise $\sigma' = \sigma$.

In cases (1), (4) (resp. (2)), we must have $\sigma = p : \mathtt{o}, \sigma''$ (resp. $p : \mathtt{i}, \sigma''$) by definition of type-allowed transitions. In clauses (1) and (2), the action is an input or an output at a continuation name that must be on top of σ, and is then removed. In clause (3), the action extrudes a continuation name, and then, following the stack discipline, the process waits for an answer on that name. In clause (4), emitting a free continuation name amounts to passing the output capability on that name to the environment. Dually, in clause (5), receiving a continuation name imposes to use it in output. Finally, in clause (6), a τ transition may come from an interaction at a continuation name, in which case σ is modified. It can also come from an interaction at a restricted name or from an internal choice; in such cases, σ is unchanged.

Theorem 25 (Subject Reduction)**.** If $\sigma \vdash_{\mathrm{wb}} P$ and $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$ then $\sigma' \vdash_{\mathrm{wb}} P'$.

If a process owns both the input and the output capability on a continuation name p, then the environment may not use p. Semantically, this is the same as having a restriction on p in the process. It is therefore safe, in the definition of barbed bisimulation and observability, to assume that all such restrictions are syntactically present, i.e., there is a single occurrence of any free continuation name. We call *clean* such processes.

Definition 26. A stack σ is *clean* if no name appears in σ both output- and input-tagged. A process P is *clean* if $\sigma \vdash_{\mathrm{wb}} P$ for some clean σ.

On clean processes, typing is preserved by reduction.

Proposition 27. If $\sigma \vdash_{\mathrm{wb}} P$ for σ clean, then $P \longrightarrow P'$ implies $\sigma \vdash_{\mathrm{wb}} P'$.

Defining barbed bisimulation on clean processes, we can use the seq function above to recast observability in the well-bracketing system from that in the sequentiality system: thus, for σ clean, we have $\sigma \vDash_{\mathrm{wb}} P \Downarrow_a$ (resp. $\sigma \vDash_{\mathrm{wb}} P \Downarrow_p$) if $\mathsf{seq}(\sigma) \vDash_{\mathrm{s}} P \Downarrow_a$ (resp. $\mathsf{seq}(\sigma) \vDash_{\mathrm{s}} P \Downarrow_p$).

In the definition of barbed equivalence, the contexts testing the processes must be clean. Writing $\simeq^{\sigma}$ for barbed equivalence at σ, we have $P \simeq^{\sigma} Q$ if $\sigma \vdash_{\mathrm{wb}} P, Q$, and for any clean σ' and any σ'/σ static context E, it holds that $E[P] \dot{\approx}^{\sigma'} E[Q]$ (note that σ itself need not be clean).

B. Discreet Processes

In this section we put forward the subclass of *discreet* processes, in which all continuation names that are exported must be private, and show how to transform any process into a discreet one. Then, on discreet processes:

(1) we express a behavioural property that formalises the stack-like discipline on the usage of continuation names;
(2) we develop proof techniques, in form of labelled bisimilarities, to reason about the behaviour of well-typed processes.

(Concerning (2), while the technical details are quite different, we follow the approach of proof techniques for receptive names in [15], where the techniques are first defined on processes where only fresh names may be sent.)

Definition 28 (Discreet processes)**.** A process P is *discreet* if any free continuation name $p \in \mathrm{fn}(P)$ may not appear in the object of an output, and, in any sub-process $x(\widetilde{a}, q).\,Q$, the same holds for q in Q. The definition is extended to contexts, yielding *discreet contexts*.

If E and P are discreet, then so is $E[P]$. We can transform all well-typed processes into discreet processes using the law in Lemma 29 below. The law transforms the output of a *global*

continuation name p into the output of a *local* name q. In general, all outputs of continuation names in a process P are local, as a global output corresponds to P delegating a stack-like obligation to another process. In other words, in general the transformation of a non-discreet process into a discreet one will modify only a few outputs of the initial process. The law in Lemma 29 is valid for barbed congruence, not just barbed equivalence, and may therefore be applied to any component of a given process.

Lemma 29. $\overline{x}\langle\widetilde{a},p\rangle \cong^{p:\circ} (\boldsymbol{\nu} q)(\overline{x}\langle\widetilde{a},q\rangle \mid q(\widetilde{b}).\,\overline{p}\langle\widetilde{b}\rangle)$.

Thus, in the definition of barbed equivalence (and congruence) it is sufficient to consider discreet contexts.

A discreet process may only export private continuation names. Dually, the process may only receive fresh continuation names from a discreet context. We call *discreet* the transitions that satisfy this property.

Definition 30 (Discreet transitions)**.** A typed transition $\sigma \vDash_{\mathrm{wb}} P \xrightarrow{\mu} P'$ is *discreet* if any continuation name in the object of μ is not free in σ (and hence also in P).

Lemma 31. If P is discreet, and $\sigma \vDash_{\mathrm{wb}} P \xrightarrow{\mu} P'$ is discreet, then P' is discreet. If, moreover, P is clean, then so is P'.

C. The Well-bracketing property on traces

Following game semantics [20], we formalise well-bracketing, that is, the stack-like behaviour of continuation names for well-typed processes, using traces of actions. In this section, all processes are discreet and clean. A trace for such a process is obtained from a sequence of discreet transitions emanating from the process, with the expected freshness conditions to avoid ambiguity among names.

Definition 32 (Trace)**.** A sequence of actions $\mu_1,\ldots,\mu_n$ is a *trace for a (discreet and clean) process* P_0 *and a stack* σ_0 if there are $\sigma_1,\ldots,\sigma_n, P_1,\ldots,P_n$ such that for all $0 \le j < n$ we have $[\sigma_j; P_j] \xrightarrow{\mu_{j+1}} [\sigma_{j+1}; P_{j+1}]$, where the transition is discreet, and moreover all continuation names appearing as object in μ_{j+1} are fresh (i.e., the names may not appear in any μ_i for $i \le j$).

The notion of discreet transition already imposes that continuation names in object position do not appear free in the process. The final condition in Definition 32 on continuation names ensures us that for actions like $(\boldsymbol{\nu} p)\overline{x}\langle\widetilde{a},p\rangle$, name p is fresh, and that after an action $\overline{p}\langle\widetilde{a}\rangle$ (thus the only allowed interaction at p has been played), name p cannot be reintroduced, e.g., in an action $x\langle\widetilde{a},p\rangle$. We simply say that $\mu_1,\ldots,\mu_n$ is a trace, or is a trace for P, when the stack or the process are clear from the context.

The well-bracketing property is best described with the notion of questions and answers.

Definition 33. For a trace $\mu_1,\ldots,\mu_n$, we set $\mu_i \curvearrowright \mu_j$ if $i < j$ and:

1) either $\mu_i = (\boldsymbol{\nu}\widetilde{c},p)\overline{a}\langle\widetilde{b},p\rangle$ and $\mu_j = p\langle\widetilde{a'}\rangle$,
2) or $\mu_i = a\langle\widetilde{b},p\rangle$ and $\mu_j = (\boldsymbol{\nu}\widetilde{c})\overline{p}\langle\widetilde{a'}\rangle$.

Actions μ_i (with a continuation name in object position) are called *questions*, while actions μ_j (with a continuation name in subject position) are called *answers*.

A discreet transition is either an internal transition, or a question, or an answer. A question mentioning a continuation name p is matched by an answer at p. When questions and answers are seen as delimiters ('$[_p$','$]_p$', different for each continuation name), a well-bracketed trace is a substring of a Dyck word.

Remark 34. For a discreet transition $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$, the value $|\sigma'| - |\sigma|$ is 1 for a question, 0 for an internal action, and -1 for an answer.

Lemma 35 (Uniqueness)**.** Given a trace $\mu_1,\ldots,\mu_n$, if $\mu_i \curvearrowright \mu_j$ and $\mu_{i'} \curvearrowright \mu_{j'}$, then we have ($i = i'$ iff $j = j'$).

Definition 36 (Well-bracketing)**.** A trace $\mu_1,\ldots,\mu_n$ is *well-bracketed* if for all $i < j$, if μ_i is a question and μ_j is an answer with $\mu_i \not\curvearrowright \mu_k$ and $\mu_k \not\curvearrowright \mu_j$ for all $i < k < j$, then $\mu_i \curvearrowright \mu_j$.

To prove that all traces are well-bracketed, we need the following property relating questions and answers to stacks.

Lemma 37. Let $\mu_1,\ldots,\mu_n$ be a trace, and $\sigma_0,\ldots,\sigma_n$ be the corresponding stacks, as in Definition 32. Suppose $\sigma_0 = \sigma_n$, and for all i, $|\sigma_i| > |\sigma_0|$. Then $\mu_1 \curvearrowright \mu_n$.

Proposition 38. Any trace (as by Definition 32) is well-bracketed.

D. Bisimulation and Full Abstraction

As in Section III, a *wb-typed relation on processes* is a set of triplets (σ, P, Q) with $\sigma \vdash_{\mathrm{wb}} P, Q$.

Definition 39 (WB-Bisimulation)**.** A wb-typed relation $\mathcal{R}$ on discreet processes is a *wb-bisimulation* if whenever $(\sigma, P, Q) \in \mathcal{R}$ and $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$ is discreet, then one of the three following clauses holds:

1) there is Q' with $Q \overset{\hat{\mu}}{\Rightarrow} Q'$ and $(\sigma', P', Q') \in \mathcal{R}$
2) $\mu = x\langle\widetilde{a},p\rangle$ and for some fresh q, there is Q' with $Q \mid (\boldsymbol{\nu} q)(\overline{x}\langle\widetilde{a},q\rangle \mid q(\widetilde{b}).\,\overline{p}\langle\widetilde{b}\rangle) \Rightarrow Q'$ and $(\sigma', P', Q') \in \mathcal{R}$
3) $\mu = u\langle\widetilde{a}\rangle$ and there is Q' with $Q \mid \overline{u}\langle\widetilde{a}\rangle \Rightarrow Q'$ and $(\sigma', P', Q') \in \mathcal{R}$,

and symmetrically for the transitions from Q.

Processes P and Q are *wb-bisimilar at* σ, noted $P \approx^{\sigma}_{\mathrm{wb}} Q$, if $(\sigma, P, Q) \in \mathcal{R}$ for some wb-bisimulation $\mathcal{R}$.

Compared to Definition 11, the clause for input actions is here split into two clauses. In clause (2), we apply Lemma 29 to obtain a discreet process.

WB-bisimulation is sound with respect to barbed equivalence for all discreet processes. The main result concerns preservation by parallel composition:

Lemma 40 (Parallel composition)**.** If $P \approx^{\sigma}_{\mathrm{wb}} Q$, then for any discreet process R and stacks σ', σ'' such that $\sigma' \vdash_{\mathrm{wb}} R$ and $\sigma'' \in \mathtt{inter}(\sigma; \sigma')$, we have $P \mid R \approx^{\sigma''}_{\mathrm{wb}} Q \mid R$.

Note that even if P, R are clean, $P \mid R$ needs not be so.

Theorem 41 (Soundness)**.** $\approx^{\sigma}_{\mathrm{wb}} \subseteq \simeq^{\sigma}$.

To prove soundness, we show that $\approx^{\sigma}_{\mathrm{wb}}$ is preserved by all discreet static contexts. By Lemma 29, we can then replace any non-discreet context with a discreet one.

We further refine the coinductive technique given by $\approx^{\sigma}_{\mathrm{wb}}$ by introducing some up-to techniques, which make it possible to work with smaller relations. We write $P \rightarrow_{\mathrm{d}} P'$ when the reduction is *deterministic*, meaning that whenever $P \xrightarrow{\mu} P''$, then $\mu = \tau$ and $P' \equiv P''$. Similarly, we write $P \Rightarrow_{\mathrm{d}} P'$ if all reduction steps are deterministic. Moreover, for a relation $\mathcal{R}$, we write $(\sigma, P, Q) \in \Rightarrow_{\mathrm{d}}\mathcal{R}^{\mathrm{C}}$ when there exists a stack σ', a σ/σ' context E, and processes P', Q' such that $Q \equiv E[Q']$, $P \Rightarrow_{\mathrm{d}} E[P']$ and $(\sigma', P', Q') \in \mathcal{R}$.

Definition 42 (Up-to static contexts and up-to deterministic reductions)**.** A wb-typed relation $\mathcal{R}$ on discreet processes is a *wb-bisimulation up-to static contexts and up-to deterministic reductions* if whenever $(\sigma, P, Q) \in \mathcal{R}$, for any discreet transition $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$, one of the following clauses holds:

1) there is Q' with $Q \overset{\hat{\mu}}{\Rightarrow} Q'$ and $(\sigma', P', Q') \in \Rightarrow_{\mathrm{d}}\mathcal{R}^{\mathrm{C}}$,
2) $\mu = x\langle\widetilde{a}, p\rangle$ and for some fresh q, there is Q' with $Q \mid (\boldsymbol{\nu} q)(\overline{x}\langle\widetilde{a}, q\rangle \mid q.\,\overline{p}) \Rightarrow Q'$ and $(\sigma', P', Q') \in \Rightarrow_{\mathrm{d}}\mathcal{R}^{\mathrm{C}}$,
3) $\mu = u\langle\widetilde{a}\rangle$ and, there is Q' with $Q \mid \overline{u}\langle\widetilde{a}\rangle \Rightarrow Q'$ and $(\sigma', P', Q') \in \Rightarrow_{\mathrm{d}}\mathcal{R}^{\mathrm{C}}$,

and symmetrically for the transitions from Q.

Lemma 43. If $\mathcal{R}$ is a wb-bisimulation up-to static contexts and up-to deterministic reductions, then $(\sigma, P, Q) \in \mathcal{R}$ implies $P \approx^{\sigma}_{\mathrm{wb}} Q$.

1) Completeness: As in Section III, we prove completeness for processes that only use output-controlled names.

Theorem 44 (Completeness)**.** For all image-finite, discreet and clean processes P, Q that only use output-controlled names, and for all σ, if $P \simeq^{\sigma} Q$ then $P \approx^{\sigma}_{\mathrm{wb}} Q$.

As for Theorem 14, the crux of the proof is defining the discriminating static contexts. The additional difficulty is related to receptiveness of continuation names: we cannot use $z.\,R$ or $G_R + T$, as in Section III, when the tester process, R or G_R, contains an input at a free continuation name.

Suppose $\sigma \vdash_{\mathrm{wb}} P$, for P discreet and clean. We decompose σ as $\xi, p_1 : \mathtt{o}, q_1 : \mathtt{i}, \ldots, p_{n-1} : \mathtt{o}, q_{n-1} : \mathtt{i}, p_n : \mathtt{o}$ for $\xi = \emptyset$ or $\xi = q : \mathtt{i}$, and then define, for fresh q_n and $\widetilde{x_i}$,

$$E^{\widetilde{x_i}}_{\sigma} \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{p_i}, \widetilde{q_i})([\cdot] \mid \prod_{i \leq n} p_i(\widetilde{y}).\,\overline{x_i}\langle\widetilde{y}, q_i\rangle).$$

We have $\xi, q_n : \mathtt{o} \vdash_{\mathrm{wb}} E^{\widetilde{x_i}}_{\sigma}[P]$. Intuitively, $E^{\widetilde{x_i}}_{\sigma}$ forwards information from the p_i's (which are in σ) to the x_i's. Accordingly, the tester process can use names in $\widetilde{x_i}$ (rather than in $\widetilde{p_i}$), and can use them in input.

Let $\mathrm{fn_o}(-)$ denote the set of free output-controlled names. We distinguish two cases. If $\xi = \emptyset$, then P is active. To follow the reasoning in the proof of Theorem 14, we work with E of the form

$$(\boldsymbol{\nu}\widetilde{x})(E^{\widetilde{x_i}}_{\sigma} \mid R + \prod_{y \in S} y(\widetilde{y'}, p).\,\overline{z}\langle p\rangle),$$

for some set $S \supseteq \mathrm{fn_o}(P) \cup \mathrm{fn_o}(Q) \cup \widetilde{x_i}$ and fresh z.

If $\xi = q : \mathtt{i}$, then P is inactive. We reason with E of the form $(\boldsymbol{\nu}\widetilde{x}, q)(E^{\widetilde{x_i}}_{\sigma} \mid \overline{z}\langle q\rangle \mid z(q').\,R)$. By typing, only the continuation name q' received at z may appear free in R. Such q' will be instantiated with q and then R will use it to test the input at q from the tested processes. (A restriction on q is needed, as the overall process has to be clean.)

In both cases, the resulting E is a $q_n : \mathtt{o}/\sigma$ static context where q_n is a fresh continuation name. More details on the proof can be found in [17].

2) An Example: We explain how the techniques we have introduced allow us to reason about a well-known example, the *well-bracketed state change* (sometimes called 'awkward', or 'very awkward', example) [21], [22], [23]. It is usually presented in ML thus:

$$M_1 \stackrel{\text{def}}{=} \mathtt{let}\ \ell\ \mathtt{=\ ref\ 0\ in\ fun\ y\ ->}\ (\ell\ \mathtt{:=\ 0;\ y()\ ;}\ \ell\ \mathtt{:=\ 1;\ y()\ ;\ !}\ell)$$

$$M_2 \stackrel{\text{def}}{=} \mathtt{fun\ y\ ->\ (y()\ ;\ y()\ ;\ 1)}$$

Function M_2 makes two calls to an external function $\mathtt{y}$ and returns 1. The other term, M_1, between the two calls, modifies a local reference ℓ, which is then used to return the final result. Intuitively, equivalence between the two functions holds because: (i) the reference ℓ in M_1 represents a local state, not accessible from an external function; (ii) computation respects well-bracketing (e.g., the language does not have control operators like call/cc).

Below are the translations of M_1 and M_2, following a standard encoding of functions and references in Aπ, and using the notations for references from Section III-C2:

$$\begin{aligned}
[\![M_1]\!]_{p'} &\stackrel{\text{def}}{=} (\boldsymbol{\nu} x, \ell)(\overline{\ell}\langle 0\rangle \mid Q) \quad \text{with} \\
Q &\stackrel{\text{def}}{=} \overline{p'}\langle x\rangle \mid !x(y,p).\,\mathtt{wr}_{\ell}\langle 0\rangle.\,(\boldsymbol{\nu} q)(\overline{y}\langle q\rangle \mid \\
&\qquad q.\,\mathtt{wr}_{\ell}\langle 1\rangle.\,(\boldsymbol{\nu} r)(\overline{y}\langle r\rangle \mid r.\,\mathtt{re}_{\ell}(n).\,\overline{p}\langle n\rangle)) \\
[\![M_2]\!]_{p'} &\stackrel{\text{def}}{=} (\boldsymbol{\nu} x)(\overline{p'}\langle x\rangle \mid !x(y,p).\,(\boldsymbol{\nu} q)(\overline{y}\langle q\rangle \mid \\
&\qquad q.\,(\boldsymbol{\nu} r)(\overline{y}\langle r\rangle \mid r.\,\overline{p}\langle 1\rangle))))
\end{aligned}$$

$[\![M_1]\!]_{p'}$ has a unique transition, $[\![M_1]\!]_{p'} \xrightarrow{(\boldsymbol{\nu} x)\overline{p'}\langle x\rangle} P_1$. Similarly, let P_2 be the unique derivative from $[\![M_2]\!]_{p'}$. The equivalence between $[\![M_1]\!]_{p'}$ and $[\![M_2]\!]_{p'}$ follows immediately from $P_1 \approx^{\emptyset}_{\mathrm{wb}} P_2$. To prove the latter, we exhibit a relation $\mathcal{R}$ containing the triple $(\emptyset, P_1, P_2)$ and show that $\mathcal{R}$ is a wb-bisimulation up-to deterministic reductions and static context. To see the importance of well-bracketing, $\mathcal{R}$ contains the triple

$$\begin{aligned}
\big(\, & q_2 : \mathtt{i}, p_2 : \mathtt{o}, r_1 : \mathtt{i}, p_1 : \mathtt{o}, \\
& (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid Q \mid q_2.\,\mathtt{wr}_{\ell}\langle 1\rangle.\,(\boldsymbol{\nu} r_2)(\overline{y}\langle r_2\rangle \\
& \qquad \mid r_2.\,\mathtt{re}_{\ell}(n).\,\overline{p_2}\langle n\rangle)) \mid r_1.\,\mathtt{re}_{\ell}(n).\,\overline{p_1}\langle n\rangle), \\
& P_2 \mid q_2.\,(\boldsymbol{\nu} r_2)(\overline{y}\langle r_2\rangle \mid r_2.\,\overline{p_2}\langle 1\rangle) \mid r_1.\,\overline{p_1}\langle 1\rangle \,\big)
\end{aligned}$$

Without the well-bracketing constraint, the first process in the triple could perform an input at r_1, an internal transition, and

finally an output $\overline{p_1}\langle 0\rangle$. The second process cannot emit 0, which would allow us to distinguish P_1 and P_2. With well-bracketing, since r_1 is not on top of the stack in the triple, the initial transition on r_1 is ruled out.

The details of the definition of $\mathcal{R}$ can be found in [17]. In the same Appendix, we also discuss a simplified example, which exposes the main difficulties. The primary simplification consists in using linear functions. Some twisting in the ML terms is necessary, as M_1 and M_2 become equivalent — even dropping well-bracketing — if they can be used at most once.

V. Related work and conclusions

Sequentiality is a form of linearity, hence our type system has similarity with, and borrow ideas from, systems with linear types, in languages for concurrency or functional languages, including types for managing locks as in [7]. The type system in [24] ensures one that terms of the Ambient calculus are single-threaded, a notion similar to the sequentiality for Aπ examined in this paper. The type system in [25] has been designed so to make the encoding of PCF into the π-calculus fully abstract. The system therefore goes beyond sequentiality as described in our paper. For instance, the system presents a form of duality on types and ensures that computations are stateless, hence also deterministic. Indeed, the only behaviours inhabited by the types are those in the image of the PCF terms. Types ensure the uniqueness of the computation thread, and such a thread is carried by outputs (the thread cannot be carried by input processes, as in our system). The system [25] has been further refined in [9], adding causality information and acyclicity constraints, so to ensure strong normalisation of well-typed processes. The issue of finding labelled bisimilarity characterisations of barbed equivalence or reduction-closed barbed equivalence is extensively discussed in [1]; see also [26] for an example involving types.

Type systems for linearity and receptiveness in the π-calculus have been introduced in [3], [27], [15]. The way we formulate well-bracketing (Definition 36) is inspired by 'well-bracketed strategies' in game semantics [20], [28], used in functional programming languages and extensions thereof (they have in turn inspired type systems for π-calculi with stack-like information and input/output alternation, e.g., [25], [29]). The notion of *well-bracketed control flow* is studied in the field of secure compilation, for a wider class of languages. In works like [30], [31], the technique of fully abstract compilation guarantees control flow correctness (and, in particular, well-bracketing) against low-level attacks.

Several methods have been proposed to establish contextual equivalence of sequential programs that include higher-order and stateful computation, including the above-mentioned game semantics, (step-indexed Kripke) logical relations [21], [22], dedicated forms of bisimulations designed on top of an operational semantics of the languages [32], [33], [34], [35], [36]. Works like [23] or algorithmic game semantics [37], aim at automatically establishing contextual equivalences, by relying on model-checking techniques.

The main goal of this paper was to tailor some of the most prominent proof techniques in the π-calculus — those based on labelled bisimilarity — to the sequentiality and well-bracketing disciplines. This is instrumental to the use of the π-calculus as a model of programming languages, as sequentiality and well-bracketing are often found in programming languages or subsets of them. We have shown the usefulness of our techniques on a number of examples, that have mainly to do with the representation of functions and store — none of the equalities in the examples is valid in the ordinary bisimilarity of the calculus.

In Section III-C we have combined our proof technique for sequentiality with techniques concerning the representation of references in π-calculus from [16]. The resulting technique allows us in some cases to reason about programs with store without an explicit representation of the store (as usually required in the techniques in the literature, recalled above). This avoids universal quantifications on the possible values contained in the store, thus reducing the size of the relation to consider, sometimes making them finite. Further possibilities of reducing the size of relations may be possible by defining 'up-to techniques' for our bisimilarities, as exemplified by the up-to technique considered in Definition 42 and applied in Section IV-D2.

Our treatment of sequentiality raises a few technical questions that deserve further investigation. We would like to see whether our proof of completeness (Theorem 14) could be extended to handle input-controlled names. Similarly, we do not know whether the result still holds if internal choice is disallowed in inactive processes. The usual encoding of an internal choice $\tau.\,P + \tau.\,Q$ in terms of parallel composition as $(\boldsymbol{\nu} c)\,(\overline{c} \mid c.\,P \mid c.\,Q)$, for some fresh c, is not applicable because the latter process is active (for instance, the encoding is not valid within a context testing active processes). Indeed, if the result still holds, the current proof might require some significant modifications. For similar reasons, it is unclear if and how our completeness proof could be tuned to handle reduction-closed variants of barbed equivalence [11], [1].

In the *asynchronous* π-calculus considered in this paper, an interaction involves only one prefixed process (the input). Therefore, in the type systems, this process always acquires the control on the thread after the interaction. In a *synchronous* setting, in contrast, an interaction involves also an output prefix. Hence the type systems could be richer, specifying, for each name, where the control on the thread goes after an interaction at that name. The representation of references in Section III-C, however, might have to be revisited as it relies on the asynchronous model.

We have studied proof techniques for sequentiality and well-bracketing in the π-calculus based on labelled bisimilarities. We would like to examine also the impact of the disciplines on algebraic theory and modal logics.

Acknowledgments

Prebet acknowledges support from the Université Franco-Italienne, programme Vinci 2020. Sangiorgi acknowledges support from the MIUR-PRIN project 'Analysis of Program Analyses' (ASPRA, ID: `201784YSZ5_004`).

References

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

[2] B. C. Pierce and D. Sangiorgi, "Typing and subtyping for mobile processes," *Math. Struct. Comput. Sci.*, vol. 6, no. 5, pp. 409–453, 1996.

[3] N. Kobayashi, B. C. Pierce, and D. N. Turner, "Linearity and the pi-calculus," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 914–947, 1999. [Online]. Available: https://doi.org/10.1145/330249.330251

[4] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in *ESOP'98, 7th European Symposium on Programming, 1998, Proceedings*, ser. Lecture Notes in Computer Science, C. Hankin, Ed., vol. 1381. Springer, 1998, pp. 122–138. [Online]. Available: https://doi.org/10.1007/BFb0053567

[5] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, and N. Yoshida, "Behavioral types in programming languages," *Found. Trends Program. Lang.*, vol. 3, no. 2-3, pp. 95–230, 2016.

[6] N. Kobayashi, "A partially deadlock-free typed process calculus," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 2, pp. 436–482, 1998. [Online]. Available: https://doi.org/10.1145/276393.278524

[7] ——, "A type system for lock-free processes," *Inf. Comput.*, vol. 177, no. 2, pp. 122–159, 2002. [Online]. Available: https://doi.org/10.1006/inco.2002.3171

[8] Y. Deng and D. Sangiorgi, "Ensuring termination by typability," *Inf. Comput.*, vol. 204, no. 7, pp. 1045–1082, 2006. [Online]. Available: https://doi.org/10.1016/j.ic.2006.03.002

[9] N. Yoshida, M. Berger, and K. Honda, "Strong normalisation in the pi -calculus," *Inf. Comput.*, vol. 191, no. 2, pp. 145–202, 2004. [Online]. Available: https://doi.org/10.1016/j.ic.2003.08.004

[10] N. Kobayashi and D. Sangiorgi, "A hybrid type system for lock-freedom of mobile processes," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 5, pp. 16:1–16:49, 2010. [Online]. Available: https://doi.org/10.1145/1745312.1745313

[11] K. Honda and N. Yoshida, "On reduction-based process semantics," *TCS*, vol. 152, no. 2, pp. 437–486, 1995.

[12] D. Sangiorgi and D. Walker, "Some results on barbed equivalences in pi-calculus," in *Proc. CONCUR '01*, ser. Lecture Notes in Computer Science, vol. 2154. Springer Verlag, 2001.

[13] R. Milner, "Functions as processes," *Math. Struct. Comput. Sci.*, vol. 2, no. 2, pp. 119–141, 1992. [Online]. Available: https://doi.org/10.1017/S0960129500001407

[14] D. Sangiorgi, "The lazy lambda calculus in a concurrency scenario," *Inf. Comput.*, vol. 111, no. 1, pp. 120–153, 1994. [Online]. Available: https://doi.org/10.1006/inco.1994.1042

[15] ——, "The name discipline of uniform receptiveness," *Theor. Comput. Sci.*, vol. 221, no. 1-2, pp. 457–493, 1999.

[16] D. Hirschkoff, E. Prebet, and D. Sangiorgi, "On the representation of references in the pi-calculus," in *31st International Conference on Concurrency Theory, CONCUR 2020*, ser. LIPIcs, I. Konnov and L. Kovács, Eds., vol. 171. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 34:1–34:20. [Online]. Available: https://doi.org/10.4230/LIPIcs.CONCUR.2020.34

[17] ——, "On sequentiality and well-bracketing in the π-calculus," Apr. 2021, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03203191

[18] R. M. Amadio, I. Castellani, and D. Sangiorgi, "On bisimulations for the asynchronous pi-calculus," *Theor. Comput. Sci.*, vol. 195, no. 2, pp. 291–324, 1998. [Online]. Available: https://doi.org/10.1016/S0304-3975(97)00223-5

[19] R. Milner, "The polyadic π-calculus: a tutorial," LFCS, Tech. Rep. ECS–LFCS–91–180, 1991, Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

[20] J. M. E. Hyland and C. L. Ong, "On full abstraction for PCF: i, ii, and III," *Inf. Comput.*, vol. 163, no. 2, pp. 285–408, 2000. [Online]. Available: https://doi.org/10.1006/inco.2000.2917

[21] A. Ahmed, D. Dreyer, and A. Rossberg, "State-dependent representation independence," in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, Z. Shao and B. C. Pierce, Eds. ACM, 2009, pp. 340–353. [Online]. Available: https://doi.org/10.1145/1480881.1480925

[22] D. Dreyer, G. Neis, and L. Birkedal, "The impact of higher-order state and control effects on local relational reasoning," *J. Funct. Program.*, vol. 22, no. 4-5, pp. 477–528, 2012. [Online]. Available: https://doi.org/10.1017/S095679681200024X

[23] G. Jaber, "Syteci: automating contextual equivalence for higher-order programs with references," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 59:1–59:28, 2020. [Online]. Available: https://doi.org/10.1145/3371127

[24] F. Levi and D. Sangiorgi, "Controlling interference in ambients," in *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2000*, M. N. Wegman and T. W. Reps, Eds. ACM, 2000, pp. 352–364. [Online]. Available: https://doi.org/10.1145/325694.325741

[25] M. Berger, K. Honda, and N. Yoshida, "Sequentiality and the pi-calculus," in *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Proceedings*, ser. Lecture Notes in Computer Science, S. Abramsky, Ed., vol. 2044. Springer, 2001, pp. 29–45. [Online]. Available: https://doi.org/10.1007/3-540-45413-6_7

[26] M. Hennessy and J. Rathke, "Typed behavioural equivalences for processes in the presence of subtyping," *Math. Struct. Comput. Sci.*, vol. 14, no. 5, pp. 651–684, 2004. [Online]. Available: https://doi.org/10.1017/S0960129504004281

[27] A. Igarashi and N. Kobayashi, "Type reconstruction for linear -calculus with I/O subtyping," *Inf. Comput.*, vol. 161, no. 1, pp. 1–44, 2000. [Online]. Available: https://doi.org/10.1006/inco.2000.2872

[28] J. Laird, "Full abstraction for functional languages with control," in *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, 1997*. IEEE Computer Society, 1997, pp. 58–67. [Online]. Available: https://doi.org/10.1109/LICS.1997.614931

[29] K. Honda, "Processes and games," *Electron. Notes Theor. Comput. Sci.*, vol. 71, pp. 40–69, 2002. [Online]. Available: https://doi.org/10.1016/S1571-0661(05)82528-9

[30] L. Skorstengaard, D. Devriese, and L. Birkedal, "Stktokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 19:1–19:28, 2019. [Online]. Available: https://doi.org/10.1145/3290332

[31] M. Patrignani, D. Devriese, and F. Piessens, "On modular and fully-abstract compilation," in *IEEE 29th Computer Security Foundations Symposium, CSF 2016*. IEEE Computer Society, 2016, pp. 17–30. [Online]. Available: https://doi.org/10.1109/CSF.2016.9

[32] V. Koutavas and M. Wand, "Small bisimulations for reasoning about higher-order imperative programs," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, J. G. Morrisett and S. L. P. Jones, Eds. ACM, 2006, pp. 141–152. [Online]. Available: https://doi.org/10.1145/1111037.1111050

[33] D. Sangiorgi, N. Kobayashi, and E. Sumii, "Environmental bisimulations for higher-order languages," in *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), Proceedings*. IEEE Computer Society, 2007, pp. 293–302. [Online]. Available: https://doi.org/10.1109/LICS.2007.17

[34] V. Koutavas, P. B. Levy, and E. Sumii, "From applicative to environmental bisimulation," in *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011*, ser. Electronic Notes in Theoretical Computer Science, M. W. Mislove and J. Ouaknine, Eds., vol. 276. Elsevier, 2011, pp. 215–235. [Online]. Available: https://doi.org/10.1016/j.entcs.2011.09.023

[35] J. Madiot, D. Pous, and D. Sangiorgi, "Bisimulations up-to: Beyond first-order transition systems," in *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014. Proceedings*, ser. Lecture Notes in Computer Science, P. Baldan and D. Gorla, Eds., vol. 8704. Springer, 2014, pp. 93–108. [Online]. Available: https://doi.org/10.1007/978-3-662-44584-6_8

[36] D. Biernacki, S. Lenglet, and P. Polesiuk, "A complete normal-form bisimilarity for algebraic effects and handlers," in *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020*, ser. LIPIcs, Z. M. Ariola, Ed., vol. 167. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 7:1–7:22. [Online]. Available: https://doi.org/10.4230/LIPIcs.FSCD.2020.7

[37] A. S. Murawski and N. Tzevelekos, "Algorithmic games for full ground references," *Formal Methods Syst. Des.*, vol. 52, no. 3, pp. 277–314, 2018. [Online]. Available: https://doi.org/10.1007/s10703-017-0292-9