

Modal Logics for Typed Mobile Processes: Completeness and Logical Full Abstraction^{*}

Martin Berger¹, Kohei Honda², and Nobuko Yoshida³

¹ Department of Informatics, University of Sussex

² Department of Computer Science, Queen Mary, University of London

³ Department of Computing, Imperial College London

Abstract. We introduce an extension of Hennessy-Milner logic for the π -calculus which gives sound and complete characterisation of representative behavioural preorder and equivalence over typed processes. New connectives are introduced representing actual and hypothetical typed parallel composition and hiding. We study two compositional proof systems, characterising the May/Must preorders for infinite processes. The mixture of the two proof systems characterises bisimilarity. These proof systems are uniformly applicable to different type disciplines. The proof rules and logical axioms originate from the proof rules for Hennessy-Milner logic for the π -calculus, studied by the preceding researchers including Amadio and Dam. We demonstrate how the use of types facilitates high-level logical reasoning through examples, including Milner's encoding of data structures, and business protocols. We also show that the logic can embed a compositional program logic for call-by-value higher-order functions fully abstractly, in the sense that validity of assertions in the program logic is preserved and reflected by validity of assertions in the process logic through encoding. By dualising modality, we obtain a fully abstract logical embedding for partial correctness. These results extend to different programming constructs including those with imperative features.

Contents

1	Introduction	1
1.1	Key Technical Elements	4
1.2	Related Work	9
1.3	Outline	12
2	Processes and Types	13
2.1	Processes and Reduction	13
2.2	Types and Typing Rules	16
2.3	Transition, Bisimilarity and Testing	19
3	Assertion and Judgement	22
3.1	Assertion Language	22

^{*} Draft version, April 15, 2009. Updated: May 25, 2009.

3.2	Judgement	26
3.3	Examples of Assertions	26
4	Semantics of Logic	29
4.1	Properties and Parametrised Properties	29
4.2	Interpretation of Assertions	31
4.3	Interpretation of Judgements	32
4.4	Names and distinction.	33
4.5	Typed Interpretation of \circ , \triangleright and ν .	33
4.6	Minimal Typing for Assertions	34
5	Reasoning about May Modalities	35
5.1	Approach to Proof Rules	35
5.2	Proof Rules for the May Modality	36
5.3	Axioms for the May Modality	39
6	Reasoning about Must and Mix Modalities	42
6.1	Must Modality and Prefix	42
6.2	No-action Predicate	43
6.3	Proof Rules for the Must Modality	45
6.4	Proof Rules for the Mixed Modality	47
6.5	Axioms and Reasoning for Must Modality	49
6.6	Axioms and Reasoning for Mix Modality	51
6.7	Admissibility	52
7	Soundness and Completeness of Proof Rules	57
7.1	Derivation of Characteristic Formulae	57
7.2	Well-Guarded Recursion	62
7.3	Properties of Derivation Systems	65
7.4	Soundness	72
7.5	Completeness	74
8	Reasoning Examples	78
8.1	Reasoning about Synchronised Stateful Interaction	78
8.2	Capturing Imperative Variables: Sequential and Concurrent	81
8.3	Extending Logic for Server-Client Types	82
8.4	Data Type Encoding: from Untyped to Typed	84
8.5	Comparison with Dam's Specification	88
9	Determinism and Elimination Results	89
9.1	Types and Affine Processes	89
9.2	Semantics of Assertions Based on \cong	92
9.3	Names, Distinctions and Symmetries	94
9.4	Elimination of ν and \circ	96
9.5	Examples	99
9.6	Call-by-value PCF in the Affine π -Calculus	100

10 Logical Full Abstraction of PCFv	102
10.1 From Equational to Logical Embedding	102
10.2 Logic for PCFv	102
10.3 Embedding PCFv into the π -Calculus	103
10.4 Logical Embedding	104
10.5 Further Logical Embeddings	107
A Proofs for Section 7	114
A.1 Remaining Cases for Proposition 7.11	114
A.2 Remaining Cases for Proposition 7.13	115
A.3 Remaining Cases for Proposition 7.15	115
A.4 Remaining Cases for Proposition 7.17	115
A.5 Proof of Lemma 7.28	116
B Axioms: Informal Presentation	122
B.1 Logical Axioms	122
B.2 Axioms for Strong Modalities	124
B.3 Axioms for Weak Modalities	126

1 Introduction

Communication is becoming a foremost element of computing, from web services to sensor networks to multicore programming. Communication systems exhibit a wide diversity of behaviour, sequential and concurrent, stateless and stateful, first-order and higher-order, deterministic and non-deterministic. A useful way of understanding this diversity is to classify behaviour into types. A compositional universe of types has a fundamental merit in engineering, guaranteeing basic safety such as the lack of synchronisation error and offering high-level abstraction in description and analysis, as well as helping distilled understanding of the semantics of processes.

The π -calculus [86] is an expressive formalism for concurrency, representing a vast array of communication behaviour in its small syntax. Starting from Milner's investigation of typed (or sorted) channels [82], many different notions of types have been studied for this calculus, yielding diverse universes of types. For example one type discipline turns the π -calculus into a semantic universe which exactly captures call-by-name and call-by-value sequential higher-order computation [17], while others ensure linearity [51, 67, 82], termination [99, 120] and different kinds of causality, e.g. [63, 66, 118].

The expressiveness of the π -calculus with respect to a broad range of typed universes comes from its central operation, name passing, and its parsimonious set of algebraic operators for composing processes such as prefix, concurrent composition and hiding, both having their origins in the preceding process algebras such as CCS [77], CSP [49] and ACP [20]. The expressiveness of this calculus leads to the power to describe and represent a wide range of behaviours of existing sequential and concurrent programming languages. Further, this representability can be made semantically exact through simple type structures based on duality [17, 18], which are in deep connection with other basic theories of computing such as game semantics [9, 61] and linear logic [38]. This exact semantic representability follows the standard principle of fully abstract semantics of programming languages in denotational semantics [76]: each process type representing a program type is inhabited by exactly those processes which semantically represent those programs inhabiting the original program type. Note this means there is an intrinsic way, as process types, to constrain the fine-grained dynamics and algebra of the π -calculus so that they can faithfully represent the dynamics and algebra of a given programming language, for a large number of language constructs. The representability leads to a potential of this calculus as a means to study diverse notions of programming languages and computational formalisms through semantically precise representation, in-

cluding functional and imperative, sequential and concurrent, message passing and shared variables, first-order and higher-order, stateful and stateless.

In this paper we investigate a basic shape of logics for typed name passing processes. The use of typed π -calculi in the present context has two motivations. Firstly, types offer abstraction of communication behaviour, giving a basis for more intelligible description and more efficient validation of their properties. Through the expressiveness and conciseness of the π -calculus, the interplay between types (which dictate the language constructs and how they can be combined) and logics (which dictate what properties such a combination may or may not satisfy) for reasoning about communicating processes can be studied in a formal and general setting without losing tractability. Such an inquiry may be engineeringly valuable given the increasing significance of concurrent and communication-centred programming and our relative lack of understanding of fundamental principles to design, specify and control their behaviour, unlike those of sequential computation. Here our aim is to use typed π -calculi for the programme to explore specification and reasoning methods for programming with communication and concurrency, carried out in a tiny syntax but without losing essential richness of communicating processes.

Secondly, the representability of diverse typed universes in the π -calculus mentioned above suggests that this programme can be extended to other universes of typed behaviour, including sequential ones. Once we have a general framework to specify and reason about typed name-passing processes, it will become applicable to a wide range of programming languages, not only concurrent, communication-based ones: the framework may be usable to suggest, analyse and justify program logics for a variety of programming languages. Such a study is not only technically interesting but also has an engineering significance when a single programming language incorporates constructs from different paradigms (as will be a norm in future concurrent programming which will combine different abstractions such as communications and shared variable concurrency for clarity and performance) or when a single application is made up from components written in different programming languages (which is already a norm in many large-scale applications, using different general and domain-specific languages to describe and combine different components).

Technically we study Hennessy-Milner logic [42] with fixed point operators [11, 68, 117] for typed π -calculi, based on the preceding studies on the modal logics for the untyped π -calculus [10, 31, 87]. Hennessy-Milner logic is originally introduced as a logic for CCS [78], which precedes the π -calculus, and is one of the most widely studied logics for communicating processes. The logic has been shown to have a significant property in the context of CCS in that it completely characterises semantics of processes [42]. Such a completeness

property is important for general specification languages, since one of the key merits of logical specifications is to be able to describe and validate essentially arbitrary semantic properties of programs – one can pinpoint any specific property of interest as an assertion, in order to use the description as a basis of diverse engineering activities, be it specification, modelling, refinement, verification, testing or program analysis.

This semantic completeness, combined with the representability of formulae of program logics in those of the typed process logic (reflecting the representability of behaviour of programs in typed processes), leads to a framework where, for an arbitrary programming language embeddable in a typed π -calculus, we can find a semantically complete logical language in the form of the typed Hennessy-Milner logic. On this basis, the compositional proof rules for the logic, which precisely follow the term structure of the π -calculus, offer a basis of compositional reasoning for programming languages as initiated by Hoare [48] based on Floyd's assertion method [36]. In other words, Hennessy-Milner logic for the π -calculus with compositional proof rules serves as a common platform for compositional program logics for diverse programming languages, which is made possible through decomposition of the algebra of programs into the more fine-grained algebra of processes. As we shall discuss in the last section, compositional proof rules for non-trivial programming languages, including Hoare logic, can indeed be exactly embeddable in the proof rules for the typed process logic.

Among possible applications, there are several recent languages and language extensions for well-structured description of communication and/or concurrency on the basis of sequential programming idioms, including [12, 26, 29, 33, 34, 60, 95, 104, 116]. The framework we present in this paper can in principle offer a uniform assertion framework for the two key elements of these languages, sequential and concurrent, through their representation in the π -calculus, which may use different type structures induced by their structured forms of concurrency. There are two challenges. First, can such representation lead to a tractable, and effective, reasoning framework for integration of sequential programs and concurrency in these languages? Such integration may not be an automatic process, but rather demands, in each case, considerable theoretical and design analysis. A general method as presented in the present paper may as well be able to assist such analysis efforts. Secondly, some of these languages already have, for their sequential part, sophisticated specification languages and tools, cf. [4, 71]. Can we use the present logical framework for a smooth extension of these existing assertion languages and methods to concurrency and communications? Other realms of interest may include emerging international standards for description of communication behaviours such as [6, 111]. The

applications of the presented framework in these and other areas may as well accompany the use of proof assistants and software model checking tools including [1–3, 5, 13, 30, 39, 43, 50, 97, 114].

1.1 Key Technical Elements

The present study owes much to the preceding studies on Hennessy-Milner logic [70, 103, 107, 108] and its extensions for the untyped π -calculus [10, 31, 87]. On the basis of these studies, we augment the original logic with a few new constructs, which indeed appeared in some way or other in the preceding studies. We are motivated by the need for logical specifications and reasoning for typed processes, while maintaining the key properties of Hennessy-Milner logic, in particular its semantic completeness. Together with these constructs, we believe that the logic offers a starting point for the research programmes we discussed above. Below we informally outline the key technical elements of the logic, which will be technically explored in the subsequent sections.

Basis. We use one of the most advanced logics for the untyped π -calculus by Dam [31] as our basis. Dam’s logic uses a parametrised form of recursion (representing minimal and maximal fixed points of properties) which is essential for expressive specifications and tractable reasoning for stateful behaviours.

Apart from minor presentational differences, one departure from [31] is the use of *weak behavioural semantics* (hence weak modalities) [79] as the semantics of the logic, abstracting away τ -actions. Many interesting semantic properties of communicating processes and programming languages are based on weak semantics (for example correctness of optimisation). Further, the correctness arguments of many encodings of data structures and programming languages in the π -calculus are almost always based on weak semantics [81, 82, 115]. These correctness arguments often rely on the deterministic nature of typed reduction in the representations, where additional reduction steps induced by the encoding can be abstracted away because they are semantically neutral. This is in fact one of the key effects of types on process semantics, imposing *deterministic interactions* [79, §15] at typed channels.

In the logic, we use the weak action modality augmented with the *semantically strong action modality*, which represent a strong action modality up to the underlying weak semantics. Concretely, suppose we wish to specify that a given process will necessarily emit an output after several τ -actions which do not change the semantics of the process. Such semantically neutral τ -actions followed by an observable action often arise in typed processes. While the eventual output itself can be specified by the standard weak modality, such semantic neutrality in τ -actions is hard to specify unless we use strong actions up to weak

semantics — under which such an output can be described by a strong output modality. In this way, the combination of strong and weak modalities leads to precise logical specification and reasoning of the semantics of typed prefixes and their interactions, thus contributing expressive specifications and powerful proof rules for deducing properties of typed processes.

The use of weak semantics is known to cause difficulty in the formulation of proof rules [31], which we address through new connectives, discussed below.

New Connectives (1). We augment the base logic with three logical connectives. We first discuss two of them, *parallel composition* and *hiding*, corresponding to two composition operators in the π -calculus. The parallel composition connective is written $A \circ B$ for formulae A and B , representing the situation when two processes, respectively satisfying A and B , interact in concurrent composition. The formula originates from [107]. The hiding $\nu x.A$, first treated by Cardelli and Caires in [24], represents a property of a process under the hiding of x , with the original process satisfying A .

These two connectives leave implicit potential modal actions which may result from composition. Through axioms associated with these connectives, we can derive a more explicit modal assertion. This process may be considered as a logical analogue of *expansion law* [79], which derives a behaviour equivalent to the composition of two behaviours by algebraic laws.

A key observation is that types often endow, through semantics of typed processes, powerful logical axioms for such deduction in comparison with the untyped setting (often in a close correspondence with the expansion law in the typed setting). Because these connectives come from the compositional structures of processes, the proof rules can be presented in a strictly compositional way, following the structure of processes. Once this “lazy” assertions are derived by the proof rules, we can deduce explicit modal actions of processes.

Therefore the introduction of these connectives is less about sheer expressiveness (for which fixed point operators serve the purpose [31, Proposition 3.3]) than the organisation of deduction. This organisation has its basis in central effects of types for processes: types change the semantic nature of parallel composition and hiding, so that different types induce different logical laws for the connectives. When typed processes use more than one notions of types (as found in many type disciplines for the π -calculus), such a presentation is essential for reasoning about typed processes.

New Connectives (2). The connective for hypothetical parallel composition, written $A \triangleright B$ for formulae A and B , allows one to assert a property of a process under a hypothetical parallel composition. In $A \triangleright B$, A is a property of a behaviour to be hypothetically composed; whereas B is a property of the resulting

composed behaviour. It originates in Stirling [108] and, in its spirit, its parallel can be found in the use of sequent-based presentation of Hennessy-Milner logic by Dam and Simpson [31, 103] for hypothetical reasoning. Philosophically and in practice, this connective plays a key role in diverse forms of rely-guarantee reasoning [65] for processes, where A in $A \triangleright B$ is about what the process relies on (or assumes), and B in $A \triangleright B$ is about what it guarantees (or commits). The significance of this connective in logical reasoning about typed processes is first observed by one of the present authors in [14, 15].

We illustrate the meaning of the connective in its relationship with \circ . Suppose a process has a property B . Then if we hypothetically compose this process with the property A , we obtain an assertion for this process under the hypothetical composition, $A \triangleright (A \circ B)$, saying under A , this process will have the property $A \circ B$. Thus the deduction involving \triangleright is almost always reducible to that for \circ .

The significance of \triangleright however lies in its use for specifications. For example, a component of a distributed application may as well be specified assuming the presence of another process. Similarly, a C program may as well be specified assuming the presence of appropriate libraries. It also plays a key role in embedding of existing program logics such as Hoare logic.

Types, Proof Rules and Axioms. We use the synchronous, polyadic π -calculus under a simple session type discipline [55, 109] as a target typed π -calculus. The session type discipline encompasses linearity and non-linearity in a simple, tractable syntax, as well as allowing a clean, structured description of typical behaviours of communication-centred applications, offering many reasoning examples for communications programs in a capsuled form, while the use of the synchronous version simplifies reasoning.

Suggested by our study on logics for higher-order functions [54], this logic is given two proof systems, one for the May modality (total correctness, or liveness) and one for the Must modality (partial correctness, or safety), with respective recursion rules. By merging these two systems, we obtain the third one for the mixed modality. We show all of these proof systems are sound under the same semantics; and that they respectively characterise the May testing preorder, the Must testing preorder, and the weak bisimilarity. We further prove that these proof systems are relatively complete for respective semantics. These completeness results extend to other type disciplines for the same syntax, under a mild condition (in particular partial commutativity and associativity of parallel composition [51], as well as guarded summations).

The compositional proof rules for the logic, in particular those for prefixes, are based on the preceding studies on proof rules for Hennessy-Milner logics for CCS and π -calculus, including [10, 31, 70, 103, 108].

We also apply the same framework to a different type discipline, the affine typed π -calculus, which precisely captures higher-order sequential functions, as we shall discuss below.

Reasoning with Typed Process Logic. We apply the logic to various reasoning examples, including a non-trivial reasoning for a stateful business protocol, involving structured conversations among multiple peers and merging of synchronised states. The subtleties in reasoning about composition of stateful interactions are enunciated for which we show how our logical framework can offer an effective solution.

The logic for the π -calculus with session types serves as an example usage of the logic for reasoning about communication-centred programs. As another reasoning example of a different kind, we treat Milner’s encoding of data structures [82], clarifying how types offer concise assertions and efficient reasoning for inductively defined agents. It also elucidates the nature of typed reasoning in comparison with Dam’s treatment of the same example in the untyped setting [31]. The encoding exhibits many essential aspects of the π -calculus (name creations, scope opening and name passing): we show a strong logical characterisation by assertions for these class of behaviours.

Representing Program Logics: Logical Full Abstraction. Following the second programme we discussed above, we also explore another usage of the present logic, representability of the program logics for total and partial correctness of call-by-value higher-order functions [54] in the typed process logic. Concretely, we show how we can embed a non-trivial program logic for call-by-value higher-order functions in the process logic. We use a different type discipline, the affine typed π -calculus studied in [17], using Milner’s encoding [82] of call-by-value λ -calculus.

The logic for the affine π -calculus uses the same proof rules as those for the logic for the session type discipline. It also uses the same axioms for linear modalities, augmented with additional axioms coming from its more restrictive type discipline. Using these axioms, we show the embedding satisfies a strong semantic property which we may call *logical full abstraction* inspired by [72]: the encoding, in both programs and formulae, embeds and reflects the validity in the program logic in the validity in the process logic. The result extends to provability via respective relative completeness, suggesting a unifying view on compositional logics for sequential and concurrent programs based on the representability of the π -calculus for a wide variety of programming languages. We note such a result together with completeness of the process logic, does *not* directly entail completeness of the original program logic as established in [54],

since the characteristic formulae [40, 105, 106] we gain at the level of process logics may not directly be translated back to those for the original program logic.

This last point relates to the following observation: the status of the typed process logic as a meta-logical framework may *not*, at least always, decide the shape of a high-level logic for programs: determining this shape may as well be a deed of creative abstraction, carried out at its proper abstraction level. Such an abstraction can then be assisted by having a uniform basis, though the shape of the underlying modal formulae, through their semantics, and through the analysis and justification the process logic offers.

As we suggest at the end, the embedding result extends to the logics for first-order imperative programs (Hoare Logic) [48], for polymorphic higher-order functions [58], for imperative higher-order functions [59], for imperative higher-order functions with aliasing [19] and for local state [121, 122].

Technical Contributions. We summarise the technical contributions of the present work.

- (i) A modal logic for the π -calculus with session types, which involve both types for determinism and non-determinism. The logical language allows expressive specifications involving actual and hypothetical composition; and, through the use of combination of weak and strong modalities, deterministic and non-deterministic modal behaviours (Sections 3 and 4).
- (ii) Three compositional proof systems and associated axioms for the logic, centring on May, Must and mixed modalities, respectively. All the proof rules and axioms are justified under the common semantics, and enable the reasoning about different aspects of interactional behaviour, including recursion, which we explore through simple examples (Sections 5 and 6).
- (iii) Establishment of completeness properties of these proof rules, where we show the May/Must proof systems are sound and complete with respect to May/Must preorders, respectively, while the proof system for the mixed modality characterises weak bisimilarity (Section 7).
- (iv) Demonstration of the power of the proposed proof rules and axioms through reasoning examples, including, among others, a non-trivial stateful business protocol and Milner's encoding of recursive data structures [82] (Section 8).
- (v) Applicability of the logical framework to the π -calculus with a different type discipline, the affine typed π -calculus, which represents higher-order deterministic computation. The logic uses the same logical language and proof rules, with the same completeness results. We show that, in this logic, we can eliminate some of the newly introduced connectives (Section 9).

- (vi) A fully abstract embedding of compositional program logic for higher-order functions in the π -calculus with affine types, for both partial and total correctness (Section 10), which extends to imperative first-order programs (Hoare logic) as well as its various higher-order extensions.

1.2 Related Work

Hennessey-Milner Logic for the π -Calculus Hennessey-Milner logic of the π -calculus is first studied in [87] where early and late bisimilarities are characterised by modal logics. In [10], Amadio and Dam study model checking and proof systems of Hennessey-Milner logic for the π -calculus, using minimal and maximal fixed point operators as well as a box modality for co-finite names. In [31], Dam presents a proof system with ordinal-indexed fixed point operators with a powerful discharge rule, which is sound and complete for finite state processes. He explores specification and reasoning in the logic using various examples including Milner’s encodings of data structures. Our logic is built on these two preceding works on modal logics for the untyped π -calculus: the axioms for parallel composition in the present work come from its treatment as the proof rules in [10, 31], refined by reflecting semantic properties of typed synchronisation algebra. Another difference is the study of weak modality, which we find advantageous for reasoning about complex behaviour. In addition to completeness results, we show useful axioms for weak modality enriched by the semantic analogue of strong modality, which may not be found in the literature.

The second author studied a rely-guarantee logic for a sequentially typed π -calculus [53]. The logic is for a purely functional fragment of the π -calculus [17], with a restricted form of parallel composition, from which a Hoare Logic for PCF is derived via embedding. The work is informally based on logical specifications of typed processes in modal logics for processes, even though this connection is not substantiated in [53]. The present work differs from [53] in that it is directly based on Hennessey-Milner logic, and that it can treat a wide range of deterministic and non-deterministic typed behaviours using the same set of proof rules, subsuming those considered in [53].

Composition Operators The operator for hypothetical parallel composition is one of the key elements of the proposed logic for its use in specification of and reasoning about process behaviours. By this operator, we can specify a property of a process assuming a property of a hypothetical process to be composed with. In our inquiry, the introduction of this operator has led to the other two new connectives, the connectives for actual parallel composition and hiding.

Apart from the early work by Stirling [108] mentioned already, this idea has been used and studied in the preceding works through the sequent presentation

of logical judgements combined with cut rules. Among others, the modal logics for the untyped π -calculus by Amadio and Dam [10] and Dam [31] mentioned above use this idea. In [31], this notion is enunciated as “open correctness assertions”. Simpson [103] shows that the general framework of sequent calculi for different formats of process calculi, where open correctness assertions in this sense play an essential role. Having this operator

The logical connectives for parallel composition, hypothetical parallel composition and hiding are also introduced in diverse kinds of spatial logics for concurrency, studied for Ambient by Cardelli and Gordon [27], where hypothetical composition is called an “adjunct” operator and where, in addition to hypothetical composition, an operator for hypothetical hiding is introduced, and for the untyped π -calculus by Caires and Cardelli [23, 24]. An early instance of the hiding operator is also found in Caires’s work on a spatial logic for concurrent objects [25]. A recent work by Hoare and O’Hearn, discussed later, also follows a closely related approach. Spatial logics are intended to capture the geometric structures of concurrent processes rather than their behaviour, though they also capture part of the interactions. When such geometric notions as locations and distributions become essential features of behaviour [28, 84], these two aspects — behaviour and locations — may need be combined in process logic.

From a different viewpoint, the use of these operators, which correspond to the algebra of the π -calculus, is a natural consequence of the introduction of a compositional proof system for the π -calculus in the present logic, and in that sense finds its origin in Hoare’s original compositional proof system [48], in spite of differences in technical contexts. In particular we use the consequence rule as in [48], leading to a modular organisation of compositional proof rules for processes and reasoning rules for logical formulae.

Operators for Fresh Names The logic for the untyped π -calculus studied by Dale, Tiu and others [75, 110] is aimed for efficient proof search using a freshness quantifier ∇ . The quantifiers for freshness are studied by Pitts, Gabbay, Shinwell and others [37, 93, 102] in combination with the “swapping” operators (which represent permutation of finite names). These meta-logical studies on crucial elements as found in the π -calculus will offer an indispensable basis when formalising the present logic in theorem provers.

In the context of proof rules for fixed point formulae, the proof rules we used for building characteristic formulae are essentially those introduced and studied by Larsen [70] for maximal fixed points. The use of admissibility allows us to extend it to minimal fixed points. The proof rules for fixed point operators themselves have been studied by many researchers, starting from Kozen [68].

Our treatment is suggested by standard axioms for fixed points, among others those from [31]. In particular we use fixed point formulae in which an ordinal κ in their unfolding is made explicit, for the use of well-founded induction, which is (as far as we know) first done in Dam [31] and which is highly useful for proof rules. In the examples we have explored so far, the use of general κ does not hinder practical reasoning.

Unifying Different Program Logics and Logical Full Abstraction As noted initially, one of the aims of the present work lies in possible usage of the presented framework as a unifying basis of different program logics. An embedding of a program logic in Hennessy-Milner logic is first studied by Milner [79], using concurrent imperative programs. As already noted, another background is various embedding results of different kinds of computational behaviours including a variety of programming languages starting from [80] (cf. [83]), combined with the denotational study of program semantics based on games starting from [9, 61] and reaching [8, 35, 57, 69, 73], and the link between game-based semantics the π -calculus first observed in [62].

We are inspired by the preceding work by Longley and Plotkin [72] for the notion of logical full abstraction. In their work, this term is used for denoting the preservation and reflection of validity of a program logic for a programming language and for its denotation. Our terminology is closely related to their notion by observing that the validity of the program logic in the universe of processes is indeed induced by our logical embedding: thus through the encoding of formulae, our result does imply logical full abstraction in their sense for a program logic, albeit using typed processes instead of CPOs.

Abramsky [7] studies a general framework to extract program logics from a class of algebraic CPOs. As far as semantics of programs is given in CPOs with a class of operators and they satisfy certain conditions, this method induces program logics by regarding co-prime filters as properties of programs. His framework is recently used in [21] for giving a semantic basis for a modal logic for the untyped monadic π -calculus up to strong bisimilarity. Our approach also uses representation of programs' semantics in typed mathematical universes, but using the dynamics and algebra of the π -calculus with types. Reasoning examples as discussed in the present paper may not have been explored in the preceding studies. The combination of the expressiveness of the may-must modalities in Hennessy-Milner logic and that of name passing interactions looks effective and flexible for logical reasoning on diverse programs' behaviour. Domain-theoretic structures and techniques form an essential basis for semantics of programming languages based on processes and interactions [9, 17, 61].

As a related work in a different direction, Hoare and He [46] study different (observational, algebraic and operational) ways of presenting a theory for imperative programming, and show they are equivalent in the sense that each is derivable from another in a cyclic way. Different presentations serve different engineering purposes and clarify distinct forms mathematical theories of computing can take. The present approach differs from theirs in that we focus on embeddings of different languages and their logics into a specific mathematical structure, name passing processes. The latter unification may be able to strengthen the unification in the former approach. More recently Hoare and O’Hearn [47] introduced a logical method for concurrent processes based on an idea of separation and concurrency, extending separation logic by Reynolds and O’Hearn [96] to concurrency. The notion of concurrent composition in their logic has interesting features such as demanding geometric separation and capturing not only process-theoretic parallel composition but other forms of program compositions. These features make their approach close to a specification framework which uses programs and predicates interchangeably [45].

A main difference between the presented framework and [45–47] is the use of the π -calculus and its logic as a unifying basis. As suggested already, we believe that these and other efforts to provide logical reasoning methods for programs at appropriate abstraction levels will have fruitful interplays with the presented analytical approach.

To our knowledge, precise embeddings of various program logics as we demonstrated in the present work, may not have been known in the literature, using either CPOs, processes or other structures as underlying semantic universes. The shape of axioms for stateful behaviours as we have discussed in Section 8 may also be new.

There are many different approaches to program logics, cf. [32, 41, 88, 89, 96, 112]. The applicability of the present framework to these and other forms of program logics would be an interesting subject of study.

1.3 Outline

Section 2 illustrates the typed π -calculus used in the present study, including the syntax of processes, types and behavioural pre-orders/bisimilarity. Section 3 introduces the assertion language. Section 4 presents the semantics of the assertion language. Sections 5 and 6 present three proof systems for different modalities. Section 7 proves soundness and completeness of the proof systems. Section 8 presents reasoning examples. Section 9 applies the same logical framework for a different, and this time deterministic, type discipline, and proves partial elimination results for \circ and \vee . Section 10 establishes a logical full abstraction result.

2 Processes and Types

2.1 Processes and Reduction

Processes We use the synchronous polyadic π -calculus [82] with recursion [31] under session type discipline [55, 109]. Session types centre on the notion of a structured sequence of interactions which form a logical unit of a conversation, or *session*. Processes typed under session types allow a clean description of application scenarios such as business protocols under a tractable type discipline.

We use the following sets of symbols:

- We use two symbols for *channels*, also called *names*: k, k', \dots indicate *linear names*; a, b, c, \dots are *shared names*; and their union are written u, u', \dots
- Expressions (e, e', \dots) are first-order arithmetic and boolean expressions as well as channels.⁴ v, w, \dots range over constants and channels.
- x, y, \dots range over *variables*; X, Y, \dots over *process variables*; and l, l_i, \dots over (branching/selection) *labels*.

Then the grammar of processes, ranged over by P, Q, \dots , is given by:

$$P ::= \mathbf{0} \mid a(k).P \mid \bar{a}(k).P \mid k(x).P \mid \bar{k}\langle e \rangle.P \mid k \triangleleft l.P \mid k \triangleright [l_i : P_i]_{i \in I} \\ \mid \text{if } e \text{ then } P \text{ else } Q \mid P \mid Q \mid (\nu u)P \mid (\mathbf{rec} X(\tilde{x}\tilde{k}).P)\langle \tilde{e} \rangle \mid X\langle \tilde{e} \rangle$$

The syntax is from one of the calculi based on session types [55, 109] except we omit session channel passing (delegation) for simplicity. The session type discipline represents interactional behaviours as a collection of typed, structured sessions, each session collecting possibly unbounded message exchanges using a fresh channel. It was born from an analysis of the usage of channels in the encoding of data structures and programs in the π -calculus [80]. Its orientation is practical, partly inspired by one of the initial presentations of CSP [44].

$a(k).P$ receives a request to establish a fresh session from its dual $\bar{a}(k).Q$. In both, k is bound in the body (note k is a channel name, not a variable). Through k thus established, $k(x).P$ represents the reception of a value from $\bar{k}\langle e \rangle.Q$; while $k \triangleright [l_i : P_i]_{i \in I}$ (where I is finite) waits with I -labelled branches from its dual $k \triangleleft l.P$, which selects one of them. This local branching has a simple encoding in the monadic untyped π -calculus [82] and has a clean and effective typing under duality-based typing (cf. [38]).

In the second line, $\text{if } e \text{ then } P \text{ else } Q$ is the standard conditional. $P \mid Q$ is a parallel composition and $(\nu u)P$ is a hiding. The recursion $(\mathbf{rec} X(\tilde{x}\tilde{k}).P)\langle \tilde{e} \rangle$, often omitting \tilde{k} , consists of two parts, *definition* $(\mathbf{rec} X(\tilde{x}\tilde{k}).P)$ and *parameter*

⁴ We use first-order expressions for legibility. Their representability in processes [82] carries over to process logic, see §8.

vector \tilde{e} . $\mathbf{rec}X(\tilde{x}\tilde{k}).P$ consists of a process variable X (a binder), its parameters $\tilde{x}\tilde{k}$ (binders for values and session channels), and its body P . Following the standard convention, we assume each occurrence of X in P is *guarded*, i.e. directly or indirectly under some prefix. This form of recursion is found in CCS [79], and is essential for tractable description and typing of stateful behaviour [55, 109] as well as for logical reasoning, as we shall see later.

Definition 2.1 (subject). *Given $u(x).P$ (resp. $\bar{u}(k).P$), its subject is the initial u . Given $k(x).P$ (resp. $\bar{k}(e).P$, resp. $k \triangleleft l.P$, resp. $k \triangleright [l_i : P_i]_{i \in I}$), its subject is its initial k .*

Notation 2.2 (processes).

1. (free names) $\text{fn}(P)$ denotes the set of free channels in P .
2. (trailing zero) We often omit $\mathbf{0}$ and an empty/inessential argument or parameter, e.g. \bar{a}, \bar{k} and $\mathbf{rec}X.P$ for $\bar{a}(k).\mathbf{0}, \bar{k}(\langle \rangle).\mathbf{0}$ and $(\mathbf{rec}X(\langle \rangle).P)(\langle \rangle)$.
3. (replication) We use the standard notation for replication, $!a(k).P$, which stands for $\mathbf{rec}X.a(k).(P|X)$ for fresh X .

Structural Congruence and Reduction The structural congruence \equiv is standard [55, 109], in which we include the following unfolding rule for recursion (below $e \downarrow v$ means e evaluates to v).

$$(\mathbf{rec}X(\tilde{x}\tilde{k}).P)(\tilde{e}) \equiv P[\tilde{v}/\tilde{x}\tilde{k}][\mathbf{rec}X(\tilde{x}\tilde{k}).P/X] \quad (e_i \downarrow v_i)$$

The reduction rules are defined over closed terms (i.e. those which do not contain free process variables). The main rules are:

$$\begin{aligned} a(k).P \mid \bar{a}(k).Q &\longrightarrow (vk)(P \mid Q) \\ k(x).P \mid \bar{k}(e).Q &\longrightarrow P[v/x] \mid Q \quad (e \downarrow v) \\ k \triangleright [l_i : P_i]_{i \in I} \mid k \triangleleft l_j.Q &\longrightarrow P_j \mid Q \quad (j \in I) \end{aligned}$$

with the standard if-then-else rules, closing under the evaluation contexts and structure rules.

Above, the first rule carries out *session initiation*, by interaction between two dual prefixes, using bound name passing. The second rule is for value passing using a session channel k , and the third is for branching/selection, again using a session channel k .

Example of Processes (1): Imperative Variable As a simple example, we consider a process encoding of imperative variables, following Milner [79].

$$\text{Var}\langle xv \rangle = \left(\mathbf{rec}X\langle v \rangle.x(k).k \triangleright \left[\begin{array}{l} \text{read} : \bar{k}\langle v \rangle.X\langle xv \rangle, \\ \text{write} : k(w).X\langle xw \rangle \end{array} \right] \right) \langle v \rangle$$

This variable first establishes a session at $x(k)$ and then offers the two options, for reading and for writing. If it is read through k , its content is returned through k , and it recurs without changing state. If it is written through k , it acknowledges this fact through k , and it recurs with its content changed.

Example of Processes (2): ATM As another example with stateful behaviour, with some more elaboration, we present a simple behaviour of ATM, automatic teller machine.

$$\begin{aligned} \text{rec } X(x).(a(k).(\text{rec } Y(yk).(k \triangleright [& \text{balance : } \bar{k}\langle y \rangle.Y\langle yk \rangle, \\ & \text{deposit : } k(w).\bar{k}\langle y+w \rangle.Y\langle yk \rangle, \\ & \text{quit : } X\langle x \rangle]))\langle xk \rangle \\ &)\langle 300 \rangle \end{aligned}$$

This process first establishes a session identified by k ; and offers three options, balance, deposit, and quit. If the user selects balance, then it shows a balance of the account, and recurs to the same point with the same amount (y). If the user selects deposit, then it receives a deposited amount w , and recurs to the same option with the new state as their sum $y+w$. Finally if the option quit is chosen, it exits the loop.

At the end of the definition, $\langle x \rangle$ gives the parameter for the initial value of y when this loop starts, given as the argument of the inner recursion for Y ; and “ $\langle 300 \rangle$ ” is the initial value of x when each session starts, given as the argument to the outermost recursion.

Example of Processes (3): A Stateless Server As a final example, we show replicated stateless processes (which may be typed by a discipline for uniform receptiveness [98], or a server-client type discipline [17], as we shall do later). Let P and Q be given by:

$$\begin{aligned} P &\stackrel{\text{def}}{=} \bar{f}(k).\bar{k}\langle n \rangle.k(x).\bar{f}(k').\bar{k}'\langle m \rangle.k'(y).\bar{h}\langle x \text{ div } y \rangle. \\ Q &\stackrel{\text{def}}{=} !f(k).k(n).\bar{k}\langle n! \rangle \end{aligned}$$

where $n!$ indicates the factorial of n . Then $P|Q$ calculates a binomial coefficient, $\binom{n}{m}$, outputted at h , i.e.

$$(\nu f)(P|Q) \approx \bar{h}\langle \binom{n}{m} \rangle.$$

2.2 Types and Typing Rules

Syntax of Types We use the following syntax of session types.

$$\begin{aligned}\alpha &::= \text{nat} \mid \text{bool} \mid (\tau) \mid \mathbf{rect}. \alpha \mid t \\ \tau &::= \downarrow \alpha; \tau \mid \uparrow \alpha; \tau \mid \&\{l_i : \tau_i\}_{i \in I} \mid \oplus \{l_i : \tau_i\}_{i \in I} \mid \mathbf{rec} t. \tau \mid \text{end} \mid t\end{aligned}$$

α is a *shared type* which is either a *non-deterministic channel type* (τ) ; an *atomic types*, nat or bool ; a recursive type $\mathbf{rect}. \tau$ and a type variable. We take an *equi-recursive* view of recursive types, not distinguishing between a type $\mathbf{rect}. \alpha$ and its unfolding $\alpha[\mathbf{rect}. \alpha/t]$ [55, 123].

τ is a *session type*, also called *linear type*. A session type builds the structure of interactions to be carried out inside a session (hence assigned to a session, or linear, channel). Type $\downarrow \alpha; \tau$ describes the behaviour of first inputting values of type α , then performing the actions typed by τ ; type $\uparrow \alpha; \tau$ is its dual, sending values instead of receiving. Type $\&\{l_i : \tau_i\}_{i \in I}$ says that a process waits with n options, and behaves as τ_i if i -th action is selected; type $\oplus \{l_i : \tau_i\}_{i \in I}$ says that it would select one of l_i and then behaves as τ_i , according to the selected l_i . Type end represents the inaction and is often omitted.

Linear types also include \perp , not in the grammar above, which indicates that no further connection is possible at a given linear name. Symbols τ, τ', \dots still range over linear types excepting \perp .

Given τ , we write $\bar{\tau}$ for the *dual type* of τ , given by exchanging $!$ and $?$, \uparrow and \downarrow , and $\&$ and \oplus . Formally the operation is defined by induction on the structure of types as follows:

- $\overline{\downarrow \alpha; \tau} = \uparrow \alpha; \bar{\tau}$, $\overline{\uparrow \alpha; \tau} = \downarrow \alpha; \bar{\tau}$,
- $\overline{\&\{l_i : \tau_i\}_{i \in I}} = \oplus \{l_i : \bar{\tau}_i\}_{i \in I}$, $\overline{\oplus \{l_i : \tau_i\}_{i \in I}} = \&\{l_i : \bar{\tau}_i\}_{i \in I}$,
- $\overline{\mathbf{rect}. \alpha} = \mathbf{rect}. \bar{\alpha}$, $\overline{\mathbf{rec} t. \tau} = \mathbf{rec} t. \bar{\tau}$, $\bar{\bar{t}} = t$, and $\overline{\text{end}} = \text{end}$.

We then define the partial operator for type composition \odot [109] given by:

$$(1) \tau \odot \bar{\tau} = \perp \quad (2) \alpha \odot \alpha = \alpha$$

Above, (1) says that once we compose two processes at a linear channel, then that channel becomes no longer composable. (2) says that inputs and outputs at a shared channel can be composed arbitrarily as far as they have compatible interactions.

As an example of a session type and its dual, the following is a type for the simple ATM in § 2.1 and its client:

$$\begin{aligned}\tau_{\text{ATM}} &\stackrel{\text{def}}{=} \mathbf{rect}. \oplus \{\text{balance} : \uparrow \text{nat}; t, \text{deposit} : \downarrow \text{nat}; t, \text{quit} : \text{end}\} \\ \tau_{\text{Client}} &\stackrel{\text{def}}{=} \mathbf{rect}. \&\{\text{balance} : \downarrow \text{nat}; t, \text{deposit} : \uparrow \text{nat}; t, \text{quit} : \text{end}\}\end{aligned}$$

Note τ_{ATM} and τ_{Client} are dual to each other.

Typing Judgement and Environment The typing judgement has the form $\Gamma \vdash P$, which reads: “process P conforms to the typing environment Γ ” (on this shape of the typing judgement, see our remark later). We syntactically define the set of typing environments, ranged over by Γ, Δ, \dots , as follows:

$$\Gamma ::= \emptyset \mid \Gamma, a : \alpha \mid \Gamma, X : \tilde{\alpha}\tilde{\tau} \mid \Gamma, k : \tau$$

where, as is standard[90], the comma “,” indicates the disjoint union in domains: for example, $\Gamma, a : \alpha$ indicates that a does not occur in Γ . Thus each typing environment defines a finite function from names and variables to types, mapping each shared name (resp. each variable) to a shared type, each process variable to a vector of shared types, and a linear name to a linear type. For this reason we shall often treat typing environments as functions they define. Under this convention, we write:

- $\text{dom}(\Gamma)$ for the domain of (the function defined by) Γ (e.g. $\text{dom}(\Gamma) = \{x, a, k\}$ when $\Gamma = x : \text{nat}, a : (\tau), k : \tau'$); and
- $\text{cod}(\Gamma)$ for the range of Γ (e.g. $\text{cod}(\Gamma) = \{\text{nat}, (\tau), \tau'\}$ with the above Γ), and $\Gamma(u)$ for the value for u in Γ assuming $u \in \text{dom}(\Gamma)$.

We also use the following operators for typing environments in our typing rules.

Notation 2.3 (operators on typing environments).

1. Γ, Δ denotes the concatenation (union) of Γ and Δ such that $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$ so that Γ, Δ defines a typing environment.
2. $\Gamma; \Delta$ denotes the same as Γ, Δ except we further demand Γ is for non-linear typing (i.e. $\text{dom}(\Gamma)$ only includes shared names, variables and process variables) and Δ only includes for linear typing (i.e. $\text{dom}(\Delta)$ only includes linear names).
3. $\Gamma_0 \asymp \Gamma_1$ (Γ_0 and Γ_1 are *compatible*) if $\Gamma_0(u) \odot \Gamma_1(u)$ is defined for each $u \in \text{dom}(\Gamma_0) \cap \text{dom}(\Gamma_1)$ and their process variables are disjoint. If $\Gamma_0 \asymp \Gamma_1$, we set:

$$\Gamma_0 \odot \Gamma_1 = \{(\Gamma_0 \odot \Gamma_1)(u) \mid u \in \text{dom}(\Gamma_0) \cap \text{dom}(\Gamma_1)\} \cup \Gamma_0 \setminus \text{dom}(\Gamma_1) \cup \Gamma_1 \setminus \text{dom}(\Gamma_0)$$

In other words, $\Gamma_0 \odot \Gamma_1$ assigns to each common name/variable the composition of their types; and to other identifiers their original types.

The operator \odot is used when we compose two typed processes in parallel, where we demand their interface match.

Fig. 1 Typing System

$\Gamma, a : \alpha \vdash a : \alpha \quad \Gamma \vdash \text{true}, \text{false} : \text{bool}$	$\frac{\Gamma \vdash e_i : \text{bool}}{\Gamma \vdash e_1 \text{or } e_2 : \text{bool}}$	Name, Bool, Or
$\frac{\Gamma \vdash a : (\tau) \quad \Gamma, k : \tau \vdash P}{\Gamma \vdash a(k).P}$	$\frac{\Gamma \vdash a : (\tau) \quad \Gamma, k : \bar{\tau} \vdash P}{\Gamma \vdash \bar{a}(k).P}$	Acc, Req
$\frac{\Gamma, k : \tau \cdot x : \alpha \vdash P}{\Gamma, k : \downarrow \alpha ; \tau \vdash k(x).P}$	$\frac{\Gamma \vdash e : \alpha \quad \Gamma, k : \tau \vdash P}{\Gamma, k : \uparrow \alpha ; \tau \vdash \bar{k}(e).P}$	Rcv, Send
$\frac{\Gamma, k : \tau_i \vdash P_i \quad \forall i \in I}{\Gamma, k : \&\{l_i : \tau_i\}_{i \in I} \vdash k \triangleright [l_i : P_i]_{i \in I}}$	$\frac{\Gamma, k : \tau_j \vdash P \quad j \in I}{\Gamma, k : \oplus \{l_i : \tau_i\}_{i \in I} \vdash k \triangleleft l_j.P}$	Bra, Sel
$\frac{\Gamma \vdash e \triangleright \text{bool} \quad \Gamma \vdash P_i \quad i = 1, 2}{\Gamma \vdash \text{if } e \text{ then } P_1 \text{ else } P_2}$	$\frac{\Gamma_i \vdash P_i \quad (i = 1, 2) \quad \Gamma_1 \succ \Gamma_2}{\Gamma_1 \odot \Gamma_2 \vdash P_1 \mid P_2}$	If, Conc
$\frac{\Gamma, a : (\tau) \vdash P}{\Gamma \vdash (\nu a)P}$	$\frac{\Gamma, k : \perp \vdash P}{\Gamma \vdash (\nu k)P}$	NRes, CRes, Inact
$\frac{\Gamma \vdash \bar{e} : \tilde{\alpha}}{\Gamma, X : \tilde{\alpha} \tilde{\tau} ; \tilde{k} : \tilde{\tau} \vdash X \langle \bar{e} \tilde{k} \rangle}$	$\frac{\Gamma, X : \tilde{\alpha} \tilde{\tau}, \tilde{x} : \tilde{\alpha} ; \tilde{k} : \tilde{\tau} \vdash P \quad \Gamma \vdash \bar{e} : \tilde{\alpha}}{\Gamma ; \tilde{k} : \tilde{\tau} \vdash (\mathbf{rec} X(\tilde{x} \tilde{k}).P) \langle \bar{e} \tilde{k} \rangle}$	Var, Rec

Typing Rules The typing judgement for processes $\Gamma \vdash P$, together with that for expressions, $\Gamma \vdash e : \alpha$, are derived from the rules in Figure 1. All rules are from [123] except we use the one-sided sequent. We use the following notations:

We illustrate the central typing rules. (Name) is the standard rule for variables. (Bool, Or) are also standard. Other expressions including those typed under nat are typed similarly.

(Acc) and (Req) are for initiating sessions. In (Acc), the type expected for the session channel τ to that portion of the declared session type for the shared identifier. In (Req), the session channel is expected as reverse.

(Rcv) handles the reception (input) of values; the relevant consumption α is composed in the conclusion's session environment, in a way that agrees with the protocol. (Send) is dual. (Bra) and (Sel) are the rules for branching and selection.

(If) is standard. In (Par), we parallel-compose two processes, checking the environments of the processes are composable. (NRes) and (CRes) hide a shared name and a pair of session channels, respectively. The latter erases, in the session environment, complementary communication patterns for the two end-

points of k , in order to ensure compatible dyadic interactions. In (Nil), we start from the sessions in the environment only with end-usages (for weakening).

(Var) is the rule for the introduction of the recursive variable; (Rec) is a corresponding rule for the recursive agent.

Remark 2.4. In the original session type discipline [55, 123], the typing sequent has the shape $\Gamma \vdash P \triangleright \Theta$ where Γ is an environment for shared names (mapping shared names and variables to shared types) and Θ an environment for linear names (mapping names to session types). This shape has a merit in clarity because non-linear and linear channels are treated quite differently in the session type discipline. In the present paper the sequent $\Gamma \vdash P$ puts both the non-linear and linear typing into a single typing environment, which gives a more generic form usable for diverse type disciplines. Informally, the sequent $\Gamma \vdash P$ says that a process P conforms to a process type Γ . If we allow more complex expressions in Γ including causality relations, many type disciplines for the π -calculus (in particular all that the present authors are aware of) can be written in this format.

2.3 Transition, Bisimilarity and Testing

Transition *Transition labels* (ℓ, ℓ', \dots) are given by the grammar:

$$\ell ::= \tau \mid a(k) \mid \bar{a}(k) \mid kv \mid \bar{k}v \mid k(a) \mid \bar{k}(a) \mid k \triangleright l \mid k \triangleleft l$$

where the parenthesised names (k) and (a) introduce binding. $\text{fn}(\ell)$, $\text{bn}(\ell)$ and $\text{sbj}(\ell)$ denote free, bound and subject names of ℓ . We write $\bar{\ell}$ for a dual of ℓ , defined by dualising the input and output (for example $\overline{a(k)} = \bar{a}(k)$). $\bar{\tau}$ is undefined. An untyped early labelled transition relation is defined in Figure 2 (omitting the else and symmetric rule for \mid).

The typed early transition relation has the shape:

$$\Gamma \vdash P \xrightarrow{\ell} \Gamma \setminus \ell \vdash Q \quad \text{if } P \xrightarrow{l} Q \text{ and } \Gamma \setminus \ell \text{ is defined} \quad (2.3.1)$$

The predicate $\Gamma \setminus \ell$ ensures that ℓ agrees with environment Γ and the resulting environment is given as $\Gamma \setminus \ell$. This is defined as follows:

- For all Γ , $\Gamma \setminus \tau = \Gamma$
- If $\Gamma = \Delta, a : (\tau)$ then $\Gamma \setminus a : (k) = \Gamma, k : \tau$.
- If $\Gamma = \Delta, a : (\tau)$ then $\Gamma \setminus \bar{a} : (k) = \Gamma, k : \bar{\tau}$.
- If $\Gamma = \Delta, k : \downarrow \alpha; \tau$ then $\Gamma \setminus ka = \Delta \odot a : \alpha, k : \tau$; or $\Gamma \setminus kv = \Delta, k : \tau$ (v constant); or $\Gamma \setminus k(a) = \Delta, a : \alpha, k : \tau$

Fig. 2 Untyped Transition System

$$\begin{array}{c}
 a(k).P \xrightarrow{a(k)} P \quad \bar{a}(k).P \xrightarrow{\bar{a}(k)} P \quad k(x).P \xrightarrow{kv} P[v/x] \quad k(x).P \xrightarrow{k(x)} P \\
 \bar{k}\langle e \rangle.P \xrightarrow{\bar{k}v} P \quad (e \downarrow v) \quad k \triangleleft l.P \xrightarrow{k \triangleleft l} P \quad k \triangleright [l_i : P_i]_{i \in I} \xrightarrow{k \triangleright l_j} P_j \quad (j \in I) \\
 \\
 \frac{P \xrightarrow{\ell} P' \quad e \downarrow \text{true}}{\text{if } e \text{ then } P \text{ else } Q \xrightarrow{\ell} P'} \quad \frac{P \xrightarrow{\ell} P' \quad \text{bn}(l) \cap \text{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q} \\
 \\
 \frac{P \xrightarrow{\ell} P' \quad Q \xrightarrow{\bar{\ell}} Q' \quad \text{bn}(l) = \{\bar{u}\}}{P \mid Q \xrightarrow{\tau} (v\bar{u})(P' \mid Q')} \quad \frac{P \xrightarrow{\bar{k}a} P'}{(va)P \xrightarrow{\bar{k}(a)} P'} \\
 \\
 \frac{P \xrightarrow{\ell} P' \quad u \notin \text{bn}(l) \cup \text{fn}(l)}{(vu)P \xrightarrow{\ell} (vu)P'} \quad \frac{P \equiv_{\alpha} P' \quad P' \xrightarrow{\ell} Q}{P \xrightarrow{\ell} Q} \\
 \\
 \frac{P[\bar{v}/\bar{x}][\mathbf{rec}X(\bar{x}).P/X] \xrightarrow{\ell} P' \quad e_i \downarrow v_i}{(\mathbf{rec}X(\bar{x}).P)\langle \bar{e} \rangle \xrightarrow{\ell} P'}
 \end{array}$$

- If $\Gamma = \Delta, k : \uparrow\alpha; \tau$ then $\Gamma \setminus \bar{k}a = \Delta \odot a : \alpha, k : \tau$; or $\Gamma \setminus \bar{k}v = \Delta, k : \tau$ (v constant); or $\Gamma \setminus \bar{k}(a) = \Delta, a : \alpha, k : \tau$
- If $\Gamma = \Delta, k : \&\{l_i : \tau_i\}_{i \in I}$ then $\Gamma \setminus k \triangleright l_j = \Delta, k : \tau_j$ ($j \in I$)
- If $\Gamma = \Delta, k : \oplus\{l_i : \tau_i\}_{i \in I}$ then $\Gamma \setminus k \triangleleft l_j = \Delta, k : \tau_j$ ($j \in I$)

The environment $\Gamma \setminus \ell$ represents what remains of Γ after the transition labelled by ℓ has happened. Session channels are consumed, while replicated channels are not consumed. The new previously bound channels are released. For a concrete example, consider the process $\bar{x}.y \mid \bar{y}.x$ which is typed in the environment $x : \perp, y : \perp$. Although the process has some untyped transitions, none of them is allowed by the environment.

We say $\ell (\neq \tau)$ is typed by α (resp. τ) under Δ if $\Gamma(\text{sbj}(\ell)) = \alpha$ (resp. τ). We often leave Γ and Δ implicit from the transition system. We write \Longrightarrow for a reflexive and transitive closure of $\xrightarrow{\tau}$; $\xRightarrow{\ell}$ for \Longrightarrow if $\ell = \tau$ else for $\Longrightarrow \xrightarrow{\ell} \Longrightarrow$; and \xRightarrow{s} for $\xRightarrow{\ell_1} \dots \xRightarrow{\ell_n}$ with $s = \ell_1 \dots \ell_n$.

Proposition 2.5. (1) (Subject congruence) If $\Gamma \vdash P$ and $P \equiv Q$, then $\Gamma \vdash Q$.

(2) (**Subject reduction**) If $\Gamma \vdash P$ and $P \xrightarrow{\tau} Q$, then $\Gamma \vdash Q$.

(3) If $\Gamma \vdash P$, $P \xrightarrow{\ell} Q$ and $\Gamma \setminus \ell$ is defined, then $\Gamma \setminus \ell \vdash Q$.

Proof. Straightforward, as in [17, 123]. ■

Bisimilarity and May/Must Preorder We now introduce semantic equivalence and preorders we shall use for our logic.

Definition 2.6 (weak bisimulation). A *weak bisimulation* is a symmetric relation \mathcal{R} which relates a typed process to a typed process of the same type, writing $\Gamma \vdash P \mathcal{R} Q$ when $\Gamma \vdash P$ and $\Gamma \vdash Q$ are related by \mathcal{R} , such that whenever $\Gamma \vdash P \mathcal{R} Q$ we have: $\Gamma \vdash P \xrightarrow{\ell} \Gamma' \vdash P'$ implies $\Gamma \vdash Q \xRightarrow{\hat{\ell}} \Gamma' \vdash Q'$ such that $\Gamma' \vdash P' \mathcal{R} Q'$; and \approx is the maximum such relation.

Definition 2.7 (May and Must Preorder). Assuming P and Q are typed identically, we write $P \sqsubseteq_{may} Q$ when $P \xrightarrow{s} \implies Q \xrightarrow{s}$. We also write $P \sqsubseteq_{must} Q$ when each Must test with success \bar{c} which P can meet, can also be met by Q where a *Must test* is a process O with complementary typing to P with an additional fresh channel c such that $P|O \implies P'$ implies $P' \xrightarrow{\bar{c}} R$.

Above we use the standard Must test which incorporate divergence (non-termination). This definition is natural in distributed computation where part of a system may as well be diverging without affecting other parts. With appropriate technical changes, we believe the same result holds for the Must test which demands termination.

Taking the condition for compatibility under prefix with the standard proviso about value input, i.e. demanding

$$k(x).P \mathcal{R} k(x).Q \text{ when } P[v/x] \mathcal{R} Q[v/x] \text{ for all typed } v,$$

it is easy to show \approx , \sqsubseteq_{may} and \sqsubseteq_{must} are typed (pre)congruences (the proofs are analogous to those in Appendix C.3 of [120]).

Remark 2.8 (closure under name substitution). \approx as given above is not closed under non-injective substitution of names (the proof follows the same result for the synchronous mini- π -calculus by Boreale and Sangiorgi [22]). This does not hinder our subsequent technical development (even though such closure gives a more intuitive treatment of some technical notions). This is because the notion of “property” and its parametrised version do not assume closure under non-injective name substitution. We come back to this point when we discuss semantics of logical formulae.

3 Assertion and Judgement

3.1 Assertion Language

Grammar. The logical language, based on [10, 31], is Hennessy-Milner logic with equality, value/name passing modality and parametrised fixed point operators, augmented with new connectives (\circ , \triangleright and \vee). The grammar of formulae, often called assertions, ranged over by A, B, C, \dots , is given below.

$$A ::= e_1 = e_2 \mid A \wedge B \mid \neg A \mid \forall x^\rho . A \mid \langle\!\langle \ell \rangle\!\rangle A \mid \langle \ell \rangle A \mid (\mu X(\tilde{x}).A)\langle \tilde{e} \rangle \mid X\langle \tilde{e} \rangle \\ \mid \neg x^\rho . A \mid A \circ B \mid A \triangleright B$$

ρ is a type annotation denoting either α or τ . Note that the grammar above uses expressions (e, e', \dots) from the previous section (cf. §2.1) as terms. We also use the notations $A \vee B$, $A \supset B$, $\exists x^\rho . A$, $[\ell]A$, $\llbracket \! \! \! \square \rrbracket A$ and $(\nu X(\tilde{x}).A)\langle \tilde{e} \rangle$, defined by dualisation in the standard way. E.g. $\llbracket \! \! \! \square \rrbracket A \equiv \neg(\langle\!\langle \ell \rangle\!\rangle \neg A)$.

Notation 3.1. We often omit type annotations for quantifiers. \top denotes $1 = 1$, F its negation. The standard association of operators is assumed: for example, $\forall x.A \wedge \langle\!\langle \ell \rangle\!\rangle B \supset C$ is parsed as $((\forall x.A) \wedge (\langle\!\langle \ell \rangle\!\rangle B)) \supset C$. For other operators, \circ , \triangleright , $\nu x.A$ associate as \wedge , \supset , $\exists x.A$. If deemed ambiguous we use parenthesis.

May and Must Modalities. The first line of the grammar are from [10, 31]. In $\langle \ell \rangle$, ℓ is one of the transition labels from § 2.3, excepting τ and extending $k\nu$ and $\bar{k}\nu$ to ke and $\bar{k}e$, respectively. If a process satisfies $\langle \ell \rangle A$, then it *can* do an immediate ℓ action, and immediately satisfies A . As we shall illustrate later, this “immediately” is taken semantically, i.e. the process can do ℓ after zero or more τ -actions, each of which does not change the state of the process up to the weak bisimilarity. Dually, $[\ell]A$ says that if a semantically immediate ℓ -action ever takes place, then the process satisfies A immediately after that action.

The next modality extends these semantically strong modalities to the standard weak modalities. If a process satisfies $\langle\!\langle \ell \rangle\!\rangle A$, then after zero or more silent transitions, the process can reach a state which satisfies A . Dually, if a process satisfies $\llbracket \! \! \! \square \rrbracket A$, then whatever zero or more silent actions the process may do, it will be always in a state which satisfies A . If a process P satisfies $\llbracket \! \! \! \square \rrbracket A$, we say P *stably satisfies* A .

We can now define the standard weak action modality by combining it with strong modality. We start from the may modality.

$$\langle\!\langle \ell \rangle\!\rangle A \equiv \langle\!\langle \ell \rangle\!\rangle \langle \ell \rangle \langle\!\langle \ell \rangle\!\rangle A . \quad (3.1.1)$$

If the process P satisfies $\langle\langle\ell\rangle\rangle A$, then P can reach a state satisfying A after a weak ℓ -transition (i.e. a sequence of zero or more τ -actions and a visible ℓ -action followed by another sequence of zero or more τ -actions). Dually we set:

$$\llbracket\ell\rrbracket A \equiv \llbracket\llbracket\ell\rrbracket\rrbracket A \quad (3.1.2)$$

which is equivalent to $\neg(\langle\langle\ell\rangle\rangle\neg A)$. If the process satisfies $\llbracket\ell\rrbracket A$, then *any* weak ℓ -transition always ends up satisfying A (to be more precise, whatever silent transitions the process may have, it is always in a state where any ℓ action results in a state which stably satisfies A). We shall show later that having the combination of strong and weak modalities gives a powerful specification and reasoning framework for typed processes.

Fixed Points. The minimal and maximal fixed points, $(\mu X(\bar{x}).A)\langle\bar{e}\rangle$ and $(\nu X(\bar{x}).A)\langle\bar{e}\rangle$, use parameters following [10, 31]. Their use is essential for describing stateful interactions, for example when a conversation structure includes a state-changing loop as in the ATM example. As is standard, we assume, in $(\mu X(\bar{x}).A)$ and $(\nu X(\bar{x}).A)$, that X never occurs in A negatively, ensuring monotonicity.

Remark 3.2 (abstraction in fixed point formulae). Dam [31] decomposes the standard modalities into *transition modalities* (representing actions at subjects) and *IO modalities* (representing value passing), writing, for example, $\langle a \rangle \rightarrow b.A$ for $\langle ab \rangle A$. By this he can treat parametrised fixed point operators as a special instance of parametrised formulae. We do not take his approach in the present inquiry, mainly for conciseness of presentation of examples.

Composition and Hiding. The second line lists the new constructs, \circ and ν , which correspond to the basic algebraic operators in the π -calculus (parallel composition and hiding), and \triangleright , a hypothetical version of \circ .

First, $A \circ B$, read as “ A *par* B ”, is understood as A, B in [108]. A typed process $\Gamma \vdash P$ satisfies $A \circ B$ when $\Gamma \vdash P$ has the same behaviour as $Q|R$, typed under Γ , for some Q and R , such that Q satisfies A and R satisfies B . This places typing constraint on A and B : concretely, if A and B has the minimal typings Δ and Θ then we demand $\Delta \succ \Theta$ and that $\Delta \odot \Theta \subset \Gamma$.

Secondly, $\nu x^\rho.A$, read as “ A *with* x *hidden*”, uses the quantifier the νx^ρ for name hiding, where we demand ρ to be either a shared type or \perp . A process, say P , satisfies $\nu x^\rho.A$ if there is a fresh name u of type ρ and P' such that $(\nu u)P' \approx P$ and P' satisfies $A[u/x]$. Its logical nature substantially differs from \exists , as studied in [121]. This construct plays a key role in logical calculation of assertions (which can be used for e.g. simplifying assertions) in combination with \circ . We often omit ρ for brevity.

$A \triangleright B$, which reads “*rely A then B*”, originates in the consequence relation studied in [107]. As its reading suggests, the connective embodies the rely-guarantee reasoning [65] (cf. [53]). A process $\Gamma \vdash P$ satisfies $A \triangleright B$ if, for every typed Q satisfying A , the composition of P and Q satisfies B . Again this gives a constraint on the typing of A and B : if the overall typing is Γ , and A has the minimal typing Δ , then we demand $\Gamma \asymp \Delta$ and that B is typed under $\Gamma \odot \Delta$. For example, if we wish to say $\Gamma \vdash P$ with $k : \bar{\tau} \in \Gamma$ satisfies $B \triangleright C$, then k can be typed as τ in B and if so k should be typed as \perp in C (since $\tau \odot \bar{\tau} = \perp$).

Remark 3.3 (hypothetical hiding). Caires and Cardelli [24] study a *hypothetical hiding operator*, which we may write $\bar{\nu}u.A$. This asserts that, if u were hidden in the target behaviour, then the result would satisfy A . This logical operator has been used, at the level of sequent, in many proof systems of Hennessy-Milner logic, cf. [10, 31, 107]. This operator is also useful in the context of the present logic, as we shall discuss in Section 6.6.

Mixed Modality The weak action modalities $\langle\langle\ell\rangle\rangle A$ and $\llbracket\ell\rrbracket A$ have already been introduced in (3.1.1) and (3.1.2) above. Another derived modality combines not only strong and weak modalities but also may and must modalities.

Definition 3.4 (mixed modality). $\langle\ell\rangle A = \llbracket\llbracket\langle\ell\rangle\top \wedge \llbracket\ell\rrbracket A\rrbracket$.

$\langle\ell\rangle A$ reads “surely ℓ then A ”, representing the strong may/must modalities inside the weak semantics. If P satisfies $\langle\ell\rangle A$ then whatever zero or more silent transitions P may have, it will be ready to perform ℓ . And if ℓ happens, it will immediately and necessarily reach a state which satisfies A .

Operationally, one of the significant aspects of the mixed modality is that it captures the semantics of a *prefixed process*: for example, consider $a(k).P$ and assume P satisfies A . Then $a(k).P$ satisfies $\langle a(k)\rangle A$ because it does not change state and, if an input action at a ever takes place, it will immediately become P which satisfies A . $a(k).P$ also satisfies $\langle\langle a(k)\rangle\rangle A$, $\langle a(k)\rangle A$, $\llbracket a(k)\rrbracket A$ and $\llbracket a(k)\rrbracket A$: but $\langle a(k)\rangle A$ most accurately captures its behaviour, in fact we can check easily that $\langle a(k)\rangle A$ implies these other modalities. Note also, under the lack of mixed summations (as in the present calculus), $\langle\ell\rangle A$ implies the action ℓ never gets cancelled by another action, including an internal one.

Remark 3.5 (Mixed Modality and Interference). The mixed modality (Definition 3.4) is used for capturing the semantics of typed prefixes. As a result, it can precisely describe different ways reductions induce (and do not induce) state change. As an example, writing $a.P$ for $a(k).P$ with $k \notin \text{fn}(P)$, similarly for $\bar{a}.P$, let $S_1 \stackrel{\text{def}}{=} \bar{b}$ and $S_2 \stackrel{\text{def}}{=} c.\bar{b}|\bar{c}$. Then we can show S_1 satisfies $\langle\bar{b}\rangle\top$ while S_2

does not do so. That is:

$$\vdash S_1 \blacktriangleright (\bar{b})\top \quad \vdash S_2 \blacktriangleright \neg(\bar{b})\top$$

This is because, while S_2 surely reaches, by one τ -action, a state in which it will emit at b , this τ -action is not semantically neutral — the reduction does change the meaning of the process.

Even without the “strong” modality, the present logic is expressive enough to distinguish these processes in the sense that S_1 further satisfies the assertion:

$$[[b]]F \wedge [[c]]F \wedge [[\bar{c}]]F \tag{3.1.3}$$

which, together with $(\bar{b})\top$, is enough to characterise this behaviour up to \approx under the typing $b : (\text{end}), c : (\text{end})$. However the above example suggests that just having the weak May and Must modalities may not give us a straightforward way to assert what we wish to describe: this is why we have the “strong” modality by which we can distinguish S_1 and S_2 easily. Further deriving $(\ell)A$ in the proof system is not hard partly because they are precisely the properties which prefixed processes own, and partly because they can be inferred when extracting modalities from linear and replicated interactions, as we shall see in later sections. It should also be noted that, following Dam [31], we can encode weak modalities in strong modalities combined with fixed point operators. In the present context, such equivalence can be expressed by introducing the strong τ -modality $\langle \tau \rangle A$. (end of Remark 3.5)

Remark 3.6 (Bound Name Input and Output). We use a bound name input, say $\langle\langle a(k) \rangle\rangle A$, and a bound name output, say $\langle\langle \bar{a}(k) \rangle\rangle A$, for modalities for session initiation. Since the process syntax uses bound input and bound output, these modalities come naturally. We also have bound name input and output for sending and receiving modality, say $\langle\langle k(c) \rangle\rangle A$ and $\langle\langle \bar{k}(c) \rangle\rangle A$, which may demand some understanding. First of all, for input, we may induce this modality using the following axiom, *for a fresh c* :

$$\forall x. \langle\langle kx \rangle\rangle A \supset \langle\langle k(c) \rangle\rangle A[c/x] \tag{3.1.4}$$

Assuming A specifies all cases about x , and assuming c is treated as a constant (whose freshness distinguishes it from all other constant names), it soundly describes the so-called *scope intrusion* [85]. Symmetrically, we have the following axiom, *again for a fresh c* :

$$\forall x. \langle\langle \bar{k}x \rangle\rangle A \equiv \langle\langle \bar{k}(c) \rangle\rangle A[c/x] \tag{3.1.5}$$

This now describes *scope extrusion* [85], saying it is outputting a new name c local to its scope ($A[c/x]$), noting c is taken fresh.

3.2 Judgement

We use the following notation for the judgement ⁵

$$\Gamma; E \models P \blacktriangleright A \quad (3.2.1)$$

where:

- Γ is the typing for P and A , including typing of variables in P . ⁶
- E is a finite collection of assignments of the form $X : (\tilde{x})A$, mapping a process variable to a parametrised formula (\tilde{x} are binders), and
- P is a process typed under Γ .

We read (3.2.1):

Under E , a typed process $\Gamma \vdash P$ has a property A .

Similarly, the sequent for provability is written $\Gamma; E \vdash P \blacktriangleright A$. We often write $E \models P \blacktriangleright A$ and $E \vdash P \blacktriangleright A$, leaving Γ implicit, though a sequent is always typed, both in P and A . If E is empty, we write $\Gamma \models P \blacktriangleright A$ and $\Gamma \vdash P \blacktriangleright A$; further we may omit Γ , writing $\models P \blacktriangleright A$ and $\vdash P \blacktriangleright A$. Note that, even if we often omit the typing environment Γ , it is usually non-empty, since a process P can do no actions if it has no free channels (which is quite different from calculi for functions such as λ -calculi).

3.3 Examples of Assertions

A Server-Client Example. We first illustrate the use of \circ and \triangleright using a simple example (an example usage of $\nu x.A$ is given at the end of this section). Consider:

$$P \equiv !b(k).k(x).\bar{k}\langle x+1 \rangle.\mathbf{0} \quad (3.3.1)$$

$$Q \equiv \bar{b}(k).\bar{k}\langle 2 \rangle.k(y).\bar{h}\langle y \rangle.\mathbf{0} \quad (3.3.2)$$

where $!b(k).R$ denotes $\mu X.!b(k).(R|X)$. Process P repeatedly does the following: after accepting a session request, receives a number and returns its increment, ending the session. Q in turn is the user of P , requesting a session and sending 2, receiving the result and forwarding it to h . Note P is typed as $b : (\downarrow \text{nat}; \uparrow \text{nat}; \text{end})^1$ while Q has type $b : (\uparrow \text{nat}; \downarrow \text{nat}; \text{end})^2, h : \uparrow \text{nat}; \text{end}$.

⁵ In the present work, a “judgement” (or “sequent”) is always about validity/provability as to the satisfiability of formulae by processes; while “assertions” (or “formulae”) are formulae in Hennessy-Milner logic.

⁶ In practice, we may use the standard auxiliary variables (variables which do not occur in processes) in A which are not in $\text{dom}(\Gamma)$. Formally we always extend Γ with the typing for these variables.

We can now assert for P and Q and their composition. First for P and Q individually:

$$A = \forall x^{\text{nat}}. \langle\langle b(k) \rangle\rangle \langle\langle kx \rangle\rangle \langle\langle \bar{k}x + 1 \rangle\rangle \top \quad (3.3.3)$$

$$B = \forall y^{\text{nat}}. \langle\langle \bar{b}(k) \rangle\rangle \langle\langle \bar{k}2 \rangle\rangle \langle\langle ky \rangle\rangle \langle\langle \bar{h}y \rangle\rangle \top \quad (3.3.4)$$

(We could use for A a fixed point formula, though this suffices for our present purpose.) Then $A \circ B$ is a specification for $P|Q$. The formula, $A \circ B$, entails

$$\langle\langle \bar{h}3 \rangle\rangle \top \quad (3.3.5)$$

through axioms we study in later sections. From this we can also infer, using the logical equivalence $B \equiv (A \triangleright (A \circ B))$ (which holds for any A and B as far as well-typed), that Q satisfies

$$A \triangleright \langle\langle \bar{h}3 \rangle\rangle \top, \quad (3.3.6)$$

saying:

If the process is composed with a behaviour satisfying A , then it can emit 3 via h .

Above we have only used May modality. In fact, when P and Q are composed and if further b is hidden, it can *always* emit 3. To obtain this property, we strengthen the assertions A and B to the strong modality defined in Definition 3.4.

$$A' = \forall x^{\text{nat}}. \langle\langle b(k) \rangle\rangle \langle\langle kx \rangle\rangle \langle\langle \bar{k}x + 1 \rangle\rangle \top \quad (3.3.7)$$

$$B' = \forall y^{\text{nat}}. \langle\langle \bar{b}(k) \rangle\rangle \langle\langle \bar{k}2 \rangle\rangle \langle\langle ky \rangle\rangle \langle\langle \bar{h}y \rangle\rangle \top \quad (3.3.8)$$

We can then show, through our semantics defined later, that $\forall b.(A' \circ B')$ entails

$$\langle\langle \bar{h}3 \rangle\rangle \top, \quad (3.3.9)$$

which says that $(\forall b)(P|Q)$ surely emits 3 via h . This entailment depends on the hiding of b : With b being sharable (hence non-deterministic), if no hiding is applied, this entailment *does not* hold.

Also note $(\forall b)(P|Q)$ does have an additional reduction before emitting b : but it still satisfies (3.3.9) since this reduction is semantically neutral: they are inside \approx .

The specifications based on Mixed modalities are stronger than the corresponding specifications based on May or Must modalities. Thus (3.3.9) entails (3.3.6), allowing us to extract part of the specification which we may be interested in.

Simple ATM A partial specification of the simple ATM in § 2.1 is given as:

$$\langle a(k) \rangle ((\nu Y(y, k). \langle k \triangleright \text{balance} \rangle \langle \bar{k}y \rangle Y \langle y, k \rangle) \langle 300, k \rangle) \quad (3.3.10)$$

which says that, at a , the process is ready to receive a session request: then it enters a loop, and if asked to show a balance, it will show y , and returns to the top of the loop again. As above we often use commas to separate arguments. This balance is initialised as 300. Now consider a user of this simple ATM satisfying:

$$\forall x. \langle \bar{a}(k) \rangle \langle k \triangleleft \text{balance} \rangle \langle kx \rangle \langle \bar{h}x \rangle \top \quad (3.3.11)$$

If we combine (3.3.10) and (3.3.11) by \circ , then we obtain $\langle \langle \bar{h}300 \rangle \rangle \top$. Again unless we hide b , we *cannot* derive $\langle \bar{h}300 \rangle \top$ since another user may as well interfere at the shared channel a before this user invokes this simple ATM. This point is further discussed in § 8.

Remark 3.7 (properties and name substitution). As noted in Remark 2.8, the weak bisimilarity \approx for the synchronous calculus is usually not closed under non-injective name substitution. For this reason, we allow, in the present notion of “parametrised property,” to be such that its interpretation of an assertion variable \underline{X} , a parametrised property, may map “ ab ” to say p and “ aa ” to q such that q may not be related to P by a direct substitution of a for b . Fortunately, by the general constructions of fixed points, we are still guaranteed to have both minimal and maximal fixed points.

4 Semantics of Logic

4.1 Properties and Parametrised Properties

The semantics of logical formulae follow [31], extended to the new connectives and to the typed semantics of processes (our terminology also follows [74]). A *behaviour* is a typed process up to \approx . A *property* is a collection of behaviours (that is, we are only interested in how a typed process will behave disregarding differences up to \approx). Thus a formula stands for a collection of behaviours: later we shall see, conversely, any behaviour can be characterised by some formula.

As an example of a property, consider

$$\langle\langle \bar{k}\langle 1 \rangle \rangle\rangle \top \quad (4.1.1)$$

under the typing $k : \uparrow \text{nat}; \text{end}$. This formula indicates:

“*The property of being able to output 1 through a linear channel k* ”.

That is, it denotes a collection of (the \approx -equivalence classes of) all the processes each of which can, after zero or more reductions, output 1 through k , and may do anything afterwards.

Since \approx is closed under all process operators (including prefixes due to the use of variables), we can naturally equip properties with algebra through the algebra of processes, which we define below. Recall a typed process is *closed* if its typing environment (hence the process itself) does not contain any free value/process variables.

Convention 4.1. *In the following we assume relations on processes only relate those under the same typing, often omitting type annotations. We often call such a relation typed relation.*

Definition 4.2 (algebra of properties). A *property* (written p, q, \dots) is a set of closed typed processes under an identical typing modulo \approx . We define operations on properties as:

$$\begin{aligned} p|q &= \bigcup_{P \in p, Q \in q} [P|Q]_{\approx} & (\nu u)p &= \bigcup_{P \in p} [(\nu u)P]_{\approx} \\ \langle\langle \rangle\rangle p' &= \{P \mid P \Longrightarrow P' \in p'\} & \langle \ell \rangle p' &= \{P \mid \exists P_0. (P \approx P_0 \xrightarrow{\ell} P' \in p')\} \\ p\sigma &= \{P\sigma \mid P \in p\} \end{aligned}$$

where we set:

1. $[P]_{\approx}$ is the equivalence class of \approx including P .
2. σ in the last line denotes an injective renaming over values (including names).

Proposition 4.3. *Each of the operators given in Definition 4.2 returns a property. Moreover each operator is closed under injective renaming over values.*

Proof. For the closure under \approx , all are immediate (for $\langle\langle\hat{\ell}\rangle\rangle p'$ note if $P \approx Q$ and $P \xrightarrow{\hat{\ell}} P' \in p$ then we can find $Q \xrightarrow{\hat{\ell}} Q' \approx P'$ by definition, and again by definition we have $Q' \in p$). Closure under name/value permutation is immediate (in the case of name permutation itself, this comes from the composition of two permutations being another permutation). ■

Remark 4.4 (properties). When we interpret a logical formula for a number theory, say “ $1 = 1$ ”, the underlying universe of discourse is usually natural numbers and relations on them: for example “1” denotes a number “one” and “=” the identity. In the definition above, our universe of discourse consists of typed processes modulo \approx . While we can use (e.g.) the underlying rooted labelled transition with essentially the same consequence, the present definition has a merit in that it easily extends to other process equivalences which may abstract away from concrete labels, as we shall see in a later section.

Remark 4.5 (semantics of strong modality). The clause for $\langle\ell\rangle p'$ in Definition 4.2 will be used later for defining semantics of strong modality. This definition is mediated by \approx rather than directly based on transition. This mediation is necessary for capturing strong actions inside the framework of weak semantics, and leads to a significant merit in descriptive power and reasoning. In particular, a simple definition of Mixed modality may become impossible without it. Note we cannot use strong transition directly for this purpose since in that case the logic discriminates too much: for example $(\nu k)(k!\langle 1\rangle; P \mid k?(x); Q)$ and $(\nu k)(P \mid Q[1/x])$ may be distinguished. While other notions of behavioural semantics other than the weak bisimilarity may as well be used, the choice of \approx has a merit in the present inquiry in that it enables us to obtain characterisations of well-known equivalences and pre-orders such as May and Must equivalences using the fact that \approx is one of the finest weak equivalences on processes [113]. (end of Remark 4.5)

We also need a notion of property parametrised by values (including channel names), for interpreting process variables. We only need parametrised properties that are closed under injective renaming.

Definition 4.6 (parametrised property). A *parametrised property of type \tilde{p}* is a function which maps a vector of values typed \tilde{p} to a property, written f, g, \dots , such that $f(\langle\tilde{v}\rangle\sigma) = (f(\langle\tilde{v}\rangle))\sigma$, where $f(\langle\tilde{v}\rangle)$ denotes the application of values to f , and σ is the obvious application of name/value permutation acting on values and the resulting properties.

4.2 Interpretation of Assertions

An *interpretation environment* (or sometimes just an *environment*) if there is no danger of confusion, is a finite type-respecting map from variables and assertion variables to values and parametrised properties, respectively. The symbols ξ, ξ', \dots range over interpretation environments. A *typed formula* is a pair Γ and A written $\Gamma \vdash A$, where Γ types the free channels and free variables in A . The following clauses interpret assertions as properties.

Definition 4.7 (interpretation of assertions). The *interpretation of a typed formula* $\Gamma \vdash A$ under ξ , written $\llbracket \Gamma \vdash A \rrbracket \xi$, or $\llbracket A \rrbracket \xi$ when Γ is known from the context, by the following clauses. Below we assume formulae, processes, and all data given are well-typed.

$$\begin{aligned}
\llbracket \Gamma \vdash \langle \rangle A \rrbracket \xi &= \langle \rangle \xi \llbracket \Gamma \vdash A \rrbracket \xi \\
\llbracket \Gamma \vdash \langle \ell \rangle A \rrbracket \xi &= \langle \ell \rangle \xi \llbracket \Gamma \setminus \ell \vdash A \rrbracket \xi \\
\llbracket \Gamma \vdash e_1^\alpha = e_2^\alpha \rrbracket \xi &= \{ \Gamma \vdash P \mid \xi(e_1) = \xi(e_2) \} \\
\llbracket \Gamma \vdash A \wedge B \rrbracket \xi &= \llbracket \Gamma \vdash A \rrbracket \xi \cap \llbracket \Gamma \vdash B \rrbracket \xi \\
\llbracket \Gamma \vdash A \circ B \rrbracket \xi &= \bigcup_{\Delta \odot \Theta = \Gamma} \llbracket \Delta \vdash A \rrbracket \xi \mid \llbracket \Theta \vdash B \rrbracket \xi \\
\llbracket \Gamma \vdash A \triangleright B \rrbracket \xi &= \max p^{\Gamma \xi}. ((p \mid \llbracket \Delta \vdash A \rrbracket \xi) \subset \llbracket \Delta \odot \Gamma \vdash B \rrbracket \xi) \\
\llbracket \Gamma \vdash \forall x^\alpha . A \rrbracket \xi &= \max p^{\Gamma \xi}. (\forall v : \alpha . p \subset \llbracket \Gamma, x : \alpha \vdash A \rrbracket (\xi \cdot x \mapsto v)) \\
\llbracket \Gamma \vdash \forall x^\rho . A \rrbracket \xi &= (\forall u) \llbracket \Gamma, u : \rho \vdash A \rrbracket (\xi \cdot x \mapsto u) \\
\llbracket \Gamma \vdash X \langle \tilde{e} \rangle \rrbracket \xi &= (\xi(X))(\tilde{e}) \\
\llbracket \Gamma \vdash (\mu X \langle \tilde{x} \rangle . A) \langle \tilde{e} \rangle \rrbracket \xi &= (\text{fix } \lambda f . \lambda \tilde{v} . (\llbracket A \rrbracket (\xi \cdot X \mapsto f \cdot \tilde{x} \mapsto \tilde{v}))) (\llbracket \tilde{e} \rrbracket) \\
\llbracket \Gamma \vdash \neg A \rrbracket \xi &= \{ \Gamma \vdash P \} \setminus \llbracket \Gamma \vdash A \rrbracket \xi
\end{aligned}$$

where $\max p^\Gamma . \mathcal{P}(p)$ denotes the maximum property p typed under Γ which satisfies $\mathcal{P}(p)$, where $\mathcal{P}(p)$ is a predicate whose only free variable is p .

Note well-typedness is assumed in each clause: in particular $\Gamma \xi$ mentioned in several clauses should be a well-formed environment (in the interpretation of \triangleright , we take Δ to be the minimum typing of A , and the domain of ξ should cover variables in both A and B). We can mechanically check that:

Proposition 4.8. For each typed $\Gamma \vdash A$ and type respecting ξ , $\llbracket \Gamma \vdash A \rrbracket \xi$ gives a property typed under $\Gamma \xi$.

The defining clauses in Definition 4.7 follow [31] except that (1) they are type-respecting; that (2) we use variables for channels (channels themselves are

interpreted through essentially identity, regarding them as constants); and that (3) we have extra clauses for \circ , \triangleright , and \vee . We shall illustrate these features in the following subsection.

4.3 Interpretation of Judgements

Define $\llbracket E \rrbracket \xi$ as a finite map from $\text{dom}(E)$, a set of process variables, to parametrised properties defined by:

$$\begin{aligned} \llbracket X \mapsto (\tilde{x})A \rrbracket \xi &= \{X \mapsto \lambda \tilde{v}. (\llbracket A \rrbracket (\xi, \tilde{x} \mapsto \tilde{v}))\} \\ \llbracket E, E' \rrbracket \xi &= \llbracket E \rrbracket \xi \cup \llbracket E' \rrbracket \xi. \end{aligned}$$

Intuitively, $\llbracket E \rrbracket \xi$ specifies the (parametrised) property each process variable is assumed to satisfy.

Using this notation, let us write:

$$E \vdash \sigma \prec \xi \tag{4.3.1}$$

where σ is a finite map such that:

1. For each non-process variable, say x , $\sigma(x) = \xi(x)$;
2. For each process variable, say X , in $\text{dom}(E)$, $\sigma(X) \in \xi(X)$ (i.e. if $\llbracket E \rrbracket \xi$ maps X to f and σ maps X to a parametrised process say $(\tilde{x})P$, then $P[\tilde{v}/\tilde{x}] \in f(\tilde{v})$ for each well-typed \tilde{v}).

We also write $P\sigma$ for the closed process by applying σ as above (which closes both variables and process variables) on P .

Definition 4.9 (satisfiability). We say $\Gamma \vdash P$ satisfies A under E , which we write $\Gamma; E \models P \blacktriangleright A$, when $P\sigma \in \llbracket \Gamma \vdash A \rrbracket (\xi \cdot \llbracket E \rrbracket \xi)$ for each ξ and σ such that $E \vdash \sigma \prec \xi$.

Above we need to have ξ and σ affect P variables and process variables in P .

Notation 4.10. In $\Gamma; E \models P \blacktriangleright A$, we often leave Γ implicit, writing $E \models P \blacktriangleright A$, though we assume a sequent is always typed. If further E is empty we write $\models P \blacktriangleright A$.

We conclude this section with several comments on Semantics of Assertions.

4.4 Names and distinction.

Existing studies of modal logics for the untyped π -calculus [10, 31, 87] interpret channel names as channel names, but possibly collapsing two or more names into one. Note also that, in first-order logic, both constants and variables are interpreted as elements in the target domain which may thus collapse constants, with the only difference being that we vary the interpretation of variables when defining validity.

In the present approach, we distinguish *channel names* (as constants) and *channel variables*. This is also standard in programming language semantics [117] and conforms to the syntax of processes we give in Section 2.1. We have chosen this approach since it seems to give a simpler interpretation for typed formulae and because one can recover, if one wants, the same semantic effects as the untyped one by considering formulae which use only variables. However it is also possible to extend the approach taken in [31] to the typed setting.

4.5 Typed Interpretation of \circ , \triangleright and ν .

In the interpretation of \circ :

$$\llbracket \Gamma \vdash A \circ B \rrbracket \xi = \bigcup_{\Delta \odot \Theta = \Gamma} \llbracket \Delta \vdash A \rrbracket \xi \mid \llbracket \Theta \vdash B \rrbracket \xi$$

Note we are taking Δ and Θ to be arbitrary typings such that (1) they type A and B respectively, and (2) their \odot -composition becomes Γ . An alternative is to demand A and B have the minimal typings whose composition precisely gives Γ (this does not lose generality since we can always add spurious conjunction with identities on the missing names and channels). However the present formulation gives a more flexible framework, allowing us to freely use entailment inside each component (A or B) without annotating them with explicit typings.

Similarly, in the interpretation of \triangleright :

$$\llbracket \Gamma \vdash A \triangleright B \rrbracket \xi = \max p^{\Gamma \xi}. ((p \mid \llbracket \Delta \vdash A \rrbracket \xi) \subset \llbracket \Delta \odot \Gamma \vdash B \rrbracket \xi)$$

we assume that Δ is such that $\Gamma \odot \Delta$ types B . Further ξ should cover variables in both Γ and Δ .

Along the same line, the interpretation of ν :

$$\llbracket \Gamma \vdash \nu x^{\rho}. A \rrbracket \xi = (\nu u) \llbracket \Gamma, u : \rho \vdash A \rrbracket (\xi \cdot x \mapsto u)$$

which interprets x in A as a fresh channel constant u , then hides that u , assumes the typing $\Gamma, u : \rho$ for fresh u .

4.6 Minimal Typing for Assertions

The following says that it suffices to consider the minimal typing of an outermost process/formula for judging validity. Below Γ, Δ is the disjoint union of Γ and Δ (i.e. we assume the domain of Γ and that of Δ are disjoint).

Proposition 4.11 (weakening, thinning). *Suppose $\Gamma \vdash A$ and $\Gamma \vdash P$. Then $\Gamma, \Delta \vdash P$ satisfies $\Gamma, \Delta \vdash A$ if and only if $\Gamma \vdash P$ satisfies $\Gamma \vdash A$.*

Proof. By induction on A . The only non-trivial cases are \circ and \triangleright . For the former we use $(\Gamma \odot \Gamma'), \Delta$ is the same thing as $\Gamma \odot (\Gamma', \Delta)$ (by disjointness). For the latter we use the observation: given A and B such that $A \asymp B$, the minimal typing of $A \odot B$ is the result of composing the minimal typings of A and B . ■

5 Reasoning about May Modalities

5.1 Approach to Proof Rules

In standard proof systems for Hennessy-Milner logic, including those for the π -calculus [10, 31, 108], we usually derive modal formulae applying the proof rules which rely on both the shape of processes and the shape of formulae. This approach, which gives tractable proof rules for simple calculi such as CCS, is no longer feasible in the present context, partly because of difficulty in deriving assertions about weak transitions; partly because of the inherent complexity of the π -calculus; and partly because of the diversity of type disciplines (hence of typed behaviours, hence of reasoning principles) for name passing processes. For these reasons we use the following parametrised approach:

- Compositional proof rules, where each rule corresponds to a single process construct, augmented with the standard rule of consequence.
- Axioms for logical formulae, which stipulates the entailment relation among assertions.

A similar distinction is found in Hoare logic, where the domain of discourse, number theory, is so rich that mixing number theory and compositional derivation of formulae is not practical. In the same way, we distinguish compositional derivation (proof rules) and axiomatic transformation of these formulae (axioms on formulae). The proof rules and axioms use the key ideas from the preceding works on Hennessy-Milner logic [10, 31, 108] and program logic [54].

There are three proof systems, respectively for the May modality, the Must modality and the Mixed modality, completely characterising May/Must pre-orders and bisimilarity, respectively. The derived judgements in three proof systems are all sound under the semantics given in §4, i.e we can mix these rules to derive sound judgements.

One of the key merits of having these three proof systems is that they offer separation of concerns. If one is only interested in the actions a process *can* carry out, then the first proof system may as well suffice, while the second system tells us what behaviour the process may not do. This may be useful, for example, when we reason about different parts of an application which demand different classes of properties (say total and partial correctness). All three proof systems use the same rules for parallel composition (using \circ) and hiding (using ν). Using \circ and ν hides the complexity of the behaviour of composed processes: they are unfolded through the axioms for formulae.

Apart from several basic shared axioms, the three modalities also accompany different sets of logical axioms. These axioms are again sound under the semantics given in §4.

Fig. 3 Proof System (May Modality)

$\frac{E \vdash P \blacktriangleright A}{E \vdash a(k).P \blacktriangleright \langle\langle a(k) \rangle\rangle A}$	$\frac{E \vdash P \blacktriangleright A}{E \vdash \bar{a}(k).P \blacktriangleright \langle\langle \bar{a}(k) \rangle\rangle A}$	Acc,Req
$\frac{E \vdash P \blacktriangleright A}{E \vdash k(x).P \blacktriangleright \forall x. \langle\langle kx \rangle\rangle A}$	$\frac{E \vdash P \blacktriangleright A}{E \vdash \bar{k}\langle e \rangle.P \blacktriangleright \langle\langle \bar{k}e \rangle\rangle A}$	Rcv,Send
$\frac{E \vdash P_i \blacktriangleright A_i \quad \forall i \in I}{E \vdash k \triangleright [l_i : P_i]_{i \in I} \blacktriangleright \bigwedge_{i \in I} \langle\langle k \triangleright l_i \rangle\rangle A_i}$	$\frac{E \vdash P \blacktriangleright A_j}{E \vdash k \triangleleft l_j.P \blacktriangleright \langle\langle k \triangleleft l_j \rangle\rangle A_j}$	Bra, Sel
$\frac{E \vdash P_i \blacktriangleright A_i \quad i = 1, 2}{E \vdash P_1 P_2 \blacktriangleright A_1 \circ A_2}$	$\frac{E \vdash P \blacktriangleright A \quad x \text{ fresh}}{E \vdash (\nu u)P \blacktriangleright \forall x. A[x/u]}$	$\frac{-}{E \vdash \mathbf{0} \blacktriangleright \top}$ Conc, Res, Inact
$\frac{-}{E, X : (\bar{x})A \vdash X \langle \bar{e} \rangle \blacktriangleright A[\bar{e}/\bar{x}]}$	$\frac{E, X : (\bar{x})(\forall j \leq i. A(j)) \vdash P \blacktriangleright A(i)}{E \vdash (\mathbf{rec} X(\bar{x}).P) \langle \bar{e} \rangle \blacktriangleright \forall i. A(i)[\bar{e}/\bar{x}]}$	Var, Rec-ind
$\frac{E \vdash P_1 \blacktriangleright e \supset A \quad E \vdash P_2 \blacktriangleright \neg e \supset A}{E \vdash \text{if } e \text{ then } P_1 \text{ else } P_2 \blacktriangleright A}$	$\frac{E \vdash P \blacktriangleright A \quad A \supset B}{E \vdash P \blacktriangleright B}$	If, Conseq

5.2 Proof Rules for the May Modality

The proof rules for the May modality are given in Figure 3. There is a single rule for each typing rule in Figure 3, except for Conseq. The typing environment for each judgement is not mentioned in these rules, assuming it exactly follows Figure 1.

The first six rules are about the six kinds of prefixes. As is known from the preceding studies (cf. [31, 107]), deriving the May modality from prefixes is relatively straightforward. In the first two rules, Acc and Req, we infer for initiation of session, which let two parties share fresh session (linear) channels. In session types, the newly introduced channels at session initiation are always bound, hence bound actions are used.

Rcv infers for receiving a value: since we do not know what value to be received, we quantify what to obtain universally, which can be instantiated at will. Send infers for sending a value, in which case we know what is being sent. When a name is sent, the use of ν -binder allows us to turn the free name output to the bound name output.

In Bra, the set $\{l_i\}$ of branch labels are determined by typing, which means that the assertion in the conclusion indicates all possible input actions the prefixed process can have. Note this understanding depends on the underlying typing context and the typability of the process: thus a typing gives a basis for this derivation. In Sel we know the label which is selected hence we can directly use it as part of the modal action.

Conc and Res construct formulae for composition and hiding using \circ and ν . These connectives do *not* explicitly specify the behaviour of processes but rather indicate potential structures of processes. For example, if we have the following two processes:

$$P \stackrel{\text{def}}{=} a(k).k?(x).\bar{b}(h).h!(x).h?(y).k!\langle y \rangle.0 \quad (5.2.1)$$

$$Q \stackrel{\text{def}}{=} b(h).h?(w).h!\langle w+1 \rangle.0 \quad (5.2.2)$$

Through Inact (discussed below), Acc, Req, Bra and Sel, we can infer, for P and Q respectively, the following two assertions:

$$A_P \stackrel{\text{def}}{=} \langle\langle a(k) \rangle\rangle \forall n. \langle\langle kn \rangle\rangle \langle\langle \bar{b}(h) \rangle\rangle \langle\langle \bar{h}n \rangle\rangle \langle\langle hn+1 \rangle\rangle \langle\langle \bar{k}n+1 \rangle\rangle \top \quad (5.2.3)$$

$$B_Q \stackrel{\text{def}}{=} \langle\langle b(h) \rangle\rangle \forall m. \langle\langle hm \rangle\rangle \langle\langle \bar{h}n+1 \rangle\rangle \top \quad (5.2.4)$$

Now through Conc, we can infer $A_P \circ B_Q$. Note we may as well wish to know that $P|Q$ may receive an input at a and returns its increment: this is however only implicit in $A_P \circ B_Q$, which solely juxtaposes two specifications without telling us the resulting modal behaviour. It is through axioms, on which we shall discuss soon, that we can derive explicit assertions on behaviour. The rules Conc and Res are shared in the proof systems for the Must and Mixed modalities.

Inact is the well-known proof rule reflecting the nature of May semantics [70]: the inaction never has any potential to do anything, hence it has least information. As we shall see later, the same inaction is treated quite differently in the Must and Mixed modalities.

For reasoning recursion, each modality induces a distinct rule. In the case of the May modality, we obtain the rule which uses mathematical induction. First, the proof rule for process variables, Var is standard [70, 108] and is common to all modalities. Second, on its basis, Rec-ind infers for recursion, through the course-of-value induction. In the rule, we assume i, j are in some well-ordered set [54].⁷ Rule Rec-ind says that, if, assuming that for each index j strictly smaller than an index i , that a process variable satisfies a certain property, we can infer that the same holds for the body of a recursion for the next ordinal

⁷ Mathematical induction often suffices, though higher ordinals are useful when we need to deal with e.g. lexicographic ordering.

i , then we can conclude that the recursion satisfies the stated property for every possible index. As an example usage of Rec-ind, but this time without fixed point formulae, consider a process P given as:

$$P \stackrel{\text{def}}{=} a(k).k(x).\text{if } x = 0 \text{ then } \bar{k}\langle 1 \rangle \text{ else } R \quad (5.2.5)$$

$$R \stackrel{\text{def}}{=} (\nu b)(X\langle b \rangle | \bar{b}\langle k' \rangle . \bar{k}'\langle x-1 \rangle . k'(y) . \bar{k}\langle x \times y \rangle) \quad (5.2.6)$$

The following shows how we can infer $(\mathbf{rec}X(a).P)\langle a \rangle$ computes a factorial using Rec-ind.

$$\frac{X : (a)\forall j \lesssim i. \langle a(k) \rangle \langle ki \rangle \langle \bar{k}j! \rangle \top \vdash P \blacktriangleright \langle a(k) \rangle \langle ki \rangle \langle \bar{k}i! \rangle \top}{\vdash (\mathbf{rec}X(a).P)\langle a \rangle \blacktriangleright \forall i. \langle a(k) \rangle \langle ki \rangle \langle \bar{k}i! \rangle \top} \quad (5.2.7)$$

Note the recursion computes the factorial by generating sub-processes, which is naturally captured by induction through the recursion rule.

When using Rec-ind, the indexed fixed-point formulae become useful:

$$(\mu/\nu X^\kappa(\tilde{x}).A)\langle \tilde{e} \rangle \quad (5.2.8)$$

which follow Dam [31], with κ ranging over ordinals. The notation stands for the standard κ -th approximations, given as, with λ a limit ordinal in the last line:

$$\begin{aligned} (\mu X^0(\tilde{x}).A)\langle \tilde{e} \rangle &\equiv F \\ (\mu X^{\kappa+1}(\tilde{x}).A)\langle \tilde{e} \rangle &\equiv A[(\mu X^\kappa(\tilde{x}).A)/X][\tilde{e}/\tilde{x}] \\ (\mu X^\lambda(\tilde{x}).A)\langle \tilde{e} \rangle &\equiv \exists_{i \lesssim \lambda} (\mu X^i(\tilde{x}).A)\langle \tilde{e} \rangle, \end{aligned}$$

dually for ν -recursion. As an example usage, let:

$$A(i) = (\nu Y^i(k). \langle \bar{k}1 \rangle Y\langle k \rangle)\langle k \rangle. \quad (5.2.9)$$

where we use the extended fixed point. Then we reason:

$$\begin{array}{l} 1. \text{} \\ \hline 2. X : (k)\forall j \lesssim i. A(j) \vdash \bar{k}1.X\langle k \rangle \blacktriangleright A(i) \\ \hline 3. \vdash (\mathbf{rec}X(k).\bar{k}1.X\langle k \rangle)\langle k \rangle \blacktriangleright (\nu Y(k). \langle \bar{k}1 \rangle Y\langle k \rangle)\langle k \rangle \end{array}$$

showing a process can repeatedly perform an output of 1 at k .

The conditional rule If is standard. The final proof rule is the consequence rule: as discussed at the outset of this section, one of the roles of the entailment in this rule is to extract modal content from bare assertions using \circ and ν . This is also the place where we carry out logical calculations for assertions on values which messages may carry (which in particular include number theory in the present logic: or if we are to use structured messages such as those based on XML, a logic for those structured messages).

We shall later show this system characterises the May preorder.

5.3 Axioms for the May Modality

Because of the introduction of \circ and ν , it becomes imperative to extract behavioural (modal) content of assertions using axioms. Below we illustrate some of the key axioms useful for deriving the May modality through examples. A comprehensive list of axioms can be found in Appendix B.

Recall the two processes P and Q in (5.2.1) and (5.2.2) before. Their respective assertions A_P and B_Q in (5.2.3) and (5.2.4) may be precise, but leaves the action of composed behaviour implicit. Now we show how we can make this implicit behaviour explicit. The following laws are among the basic laws for the May modality. Each axiom is valid under a typability condition: that the both sides of the entailment have the same typing (where the typing of formulae is given following that for processes in §2), as we shall illustrate later.

$$\langle\langle\ell\rangle\rangle A \circ B \supset \langle\langle\ell\rangle\rangle (A \circ B) \quad (5.3.1)$$

$$\langle\langle\ell\rangle\rangle A \circ \langle\langle\bar{\ell}\rangle\rangle B \supset \nu \text{bn}(\ell). \langle\langle\ell\rangle\rangle (A \circ B) \quad (5.3.2)$$

$$\langle\langle\rangle\rangle \langle\langle\rangle\rangle A \equiv \langle\langle\rangle\rangle A \quad (5.3.3)$$

$$\langle\langle\rangle\rangle \langle\langle\ell\rangle\rangle A \equiv \langle\langle\ell\rangle\rangle A \quad (5.3.4)$$

$$\langle\langle\ell\rangle\rangle \langle\langle\rangle\rangle A \equiv \langle\langle\ell\rangle\rangle A \quad (5.3.5)$$

$$A \circ \top \supset A \quad (5.3.6)$$

The first two rules are a logical analogue of the expansion law [79]. The first rule is particularly sensible to the typing constraint. While we can infer:

$$\langle\langle a(k) \rangle\rangle A \circ \langle\langle \bar{a}(k) \rangle\rangle B \supset \langle\langle a(k) \rangle\rangle (A \circ \langle\langle \bar{a}(k) \rangle\rangle B) \quad (5.3.7)$$

we *cannot* infer

$$\langle\langle k?1 \rangle\rangle A \circ \langle\langle k!1 \rangle\rangle B \supset \langle\langle k?1 \rangle\rangle (A \circ \langle\langle k!1 \rangle\rangle B) \quad (5.3.8)$$

since if the left-hand side of (5.3.8) is typable, then its right-hand side will not be, noting the linear channel k will given the type \perp in the former but not for the latter. By this typing condition, the axiom in effect says:

If one part of the composition can do a visible action, and if that action is not type-wise prohibited from interacting with the outside in the typing of the composed process, then the composed process can indeed do that visible action, with its continuation having the corresponding composition property.

The underlying typing constraint is of the sort which can be easily inferred by e.g. an editor.

In the next three laws, (5.3.3), (5.3.4) and (5.3.5), the first one (5.3.3) implies the remaining ones (by the definition of the weak modal action, $\langle\langle\ell\rangle\rangle A \equiv \langle\langle\ell\rangle\rangle\langle\langle\rangle\rangle A$). Finally (5.3.6) again relies on typability, following the same semantic principle as noted above.

We can now extract explicit modality from $A \circ B$ for A and B in (5.2.3) and (5.2.4) as follows. Below we set A' as $\langle\langle\bar{b}(h)\rangle\rangle\langle\langle\bar{h}n\rangle\rangle\langle\langle hn+1\rangle\rangle\langle\langle\bar{k}n+1\rangle\rangle\top$, i.e. A' is the “remaining part” of A_P after two May actions.

$$\begin{aligned} A_P \circ B_Q &\supset \langle\langle a(k)\rangle\rangle\forall n.\langle\langle kn\rangle\rangle\langle\langle\rangle\rangle(A' \circ B_Q) \\ &\supset \langle\langle a(k)\rangle\rangle\forall n.\langle\langle kn\rangle\rangle\langle\langle\rangle\rangle\forall h.(\langle\langle\bar{k}n+1\rangle\rangle\top \circ \top) \\ &\supset \langle\langle a(k)\rangle\rangle\forall n.\langle\langle kn\rangle\rangle\langle\langle\bar{k}n+1\rangle\rangle\top \end{aligned}$$

In the first implication above, we use (5.3.1). In the next implication, we use (5.3.2) combined with (5.3.5), with standard quantification rules. In the last implication, we use (5.3.6) and the standard entailment for \top .

We next show how we can use \triangleright for a more modular reasoning. In essence, \triangleright allows the use of a cut through the following axioms:

$$B \supset A \triangleright (A \circ B) \quad (5.3.9)$$

$$(A \triangleright B) \circ A \supset B \quad (5.3.10)$$

There are other rules, but we only use these two rules. The first rule says that once we know B holds for a process, then automatically we can say:

The well-typed composition of the process with any process satisfying B , will automatically satisfy $A \circ B$.

On the other hand the second rule states the symmetric case:

If we know that the well-typed composition of the process with any process satisfying A leads to a property B , and if we combine this with a process satisfying A , then the result satisfies B .

The axiom (5.3.9) allows us to reason, for A_P in (5.2.3):

$$\begin{aligned} A_P &\supset B_Q \triangleright (A_P \circ B_Q) \\ &\supset B_Q \triangleright \langle\langle a(k)\rangle\rangle\forall n.\langle\langle kn\rangle\rangle\langle\langle\bar{k}n+1\rangle\rangle\top \end{aligned}$$

If we have this assertion for P , then by the axiom 5.3.10, we immediately know P satisfies $\langle\langle a(k)\rangle\rangle\forall n.\langle\langle kn\rangle\rangle\langle\langle\bar{k}n+1\rangle\rangle\top$. Such reasoning is useful when we wish to infer a property of a process (or a code) knowing it would be composed with a process (or code) of a certain property.⁸

⁸ The corresponding method is used in [31] and [103] using process variables and the cut rule.

We also have the standard laws from the modal logic, such as

$$\langle\langle\ell\rangle\rangle(A \wedge B) \supset \langle\langle\ell\rangle\rangle A \wedge \langle\langle\ell\rangle\rangle B \quad (5.3.11)$$

and

$$\langle\langle\ell\rangle\rangle(A \vee B) \equiv \langle\langle\ell\rangle\rangle A \vee \langle\langle\ell\rangle\rangle B. \quad (5.3.12)$$

which are natural from the viewpoint of process behaviour. The remaining axioms can be found in Appendix B, while more reasoning examples using these and other axioms can be found in Section 8.

6 Reasoning about Must and Mix Modalities

6.1 Must Modality and Prefix

With the May modality, we can specify what potential behaviours a process has, up to non-deterministic branching. In contrast, with its dual, the Must modality, we specify what behaviour a process does *not* have, that it can *not* reach a certain state. The starting point of the specification with the Must modality is the following simple assertion under, say, $k : \uparrow \text{ nat}; \text{ end}$.

$$\llbracket \bar{k} 1 \rrbracket F \quad (6.1.1)$$

This assertion says that the process can *never* output 1 through k . By typing, we assume that the process may only, if ever, output a natural number through k : hence the assertion (6.1.1) says that (1) either the process can never output at k ; or (2) if it does output a number via k , it will emit something other than 1. This is a safety specification, showing some event never happens.

This simplest usage of the Must modality, specifying the incapability of a single action, is expanded by having the use of a non-trivial predicate after the Must modality, instead of F :

$$\llbracket \bar{h} 2 \rrbracket \llbracket \bar{k} 1 \rrbracket F \quad (6.1.2)$$

The formula (6.1.2) says that, if the process ever emits 2 through h , then it will never emit 1 through k . The formula as a whole is again about safety, since it tells that the process cannot do some action and it can reach only such a state. This specification is satisfied by the following process:

$$\text{if } e \text{ then } h!2.k!2.\mathbf{0} \text{ else } h!1.k!1.\mathbf{0} \quad (6.1.3)$$

Note that, whether e evaluates to truth or falsity, the process satisfies the assertion, since, for example, when falsity, 2 is not emitted through h .

The specification (6.1.2) consists of a sequence of Must modal actions terminating with F . Can we derive such specifications compositionally? Yes we can, in the same spirit of various proof rules for Must modality in the previous work (cf. [10, 107, 108]), combined with the typing environment. The basic idea is simple. Suppose P is prefixed. Then we can infer:

The only action this process can do, among those at free channels in the typing environment or silent actions, is indeed one based on its prefix, and then it reaches the specification of its continuation.

A sequence of such derivations will give us a desired safety specification. To get an idea, take the following simple process, the typing $h : \uparrow \text{ nat}; \text{ end}$, $k : \uparrow \text{ nat}; \text{ end}$.

$$h!1.k!1.\mathbf{0}, \quad (6.1.4)$$

Then starting from $\mathbf{0}$ and following each prefix one by one, we may derive the following specification:

Initially it has no output actions except sending 1 via h. If it ever does so, it has no output actions except sending 1 via k. And if it ever does so, it has no action whatever.

This informal specification captures the whole behaviour of this process *as far as the Must modality goes* — it does not say what this process can do, but it does say what it cannot do, as well as what can ever happen if it does a potential action it may be capable of. For inferring such properties from prefixes, we use the notion of *non-action predicate*, introduced in the next subsection.

6.2 No-action Predicate

We define the *no-action predicate* at Γ , written $\text{noact}(\Gamma)$. The predicate $\text{noact}(\Gamma)$ asserts that “no actions at Γ are possible”. We use a notation for refined IO-types for non-deterministic channels, written $(\tau)^{\text{I}}$, $(\bar{\tau})^{\text{O}}$ as well as the original (τ) , where the first two respectively restrict the capability to input and output only [91]. This allows us precisely to specify the absence of a certain action.⁹

Assume Γ is the typing environments whose co-domain is extended with these refined IO-types. Then the *no-action predicate for Γ* , written $\text{noact}(\Gamma)$, is defined by:

- $\text{noact}(k : \downarrow\alpha; \tau) = \forall x^\alpha. \llbracket kx \rrbracket F$, $\text{noact}(k : \&\{l_i : \tau_i\}) = \bigwedge_i \llbracket k \triangleright l_i \rrbracket F$, dually for linear output and selection.
- $\text{noact}(a : (\tau)^{\text{I}}) = \llbracket a(k) \rrbracket F$, $\text{noact}(a : (\tau)^{\text{O}}) = \llbracket \bar{a}(k) \rrbracket F$, and $\text{noact}(a : (\tau)) = \text{noact}(a : (\tau)^{\text{I}}) \wedge \text{noact}(a : (\tau)^{\text{O}})$.
- $\text{noact}(\tilde{u} : \tilde{\rho}) = \bigwedge_i \text{noact}(u_i : \rho_i)$.

We refine the no-action predicate by specifying an action possible by a prefix, written $[\ell, \Gamma]$ (which says that “this process can immediately do ℓ , and nothing else, as far as the typing Γ goes”), defined as follows. Below we assume $\ell \neq \tau$ and Γ may again use refined IO types.

- $[\ell, \Gamma]A \stackrel{\text{def}}{=} [\ell]A \wedge \text{noact}(\Gamma)$ if ℓ is an input or a branching;
- $[\bar{k}e, \Gamma]A \stackrel{\text{def}}{=} [\bar{k}e]A \wedge \forall x. (x \neq e \supset \llbracket \bar{k}x \rrbracket F) \wedge \text{noact}(\Gamma)$; and
- $[k \triangleleft l_i, \Gamma]A \stackrel{\text{def}}{=} [k \triangleleft l_i]A \wedge \bigwedge_{j \in \mathcal{N}_i} [k \triangleleft l_j]F \wedge \text{noact}(\Gamma)$.

We can now introduce the Must predicate for prefixes.

⁹ There is no change in the typing system for processes: the introduction of refined IO-types for non-deterministic channels is solely for defining the no-action predicate.

Definition 6.1 (extended Must modality). *The extended Must modality for ℓ under Γ , written $\llbracket \ell, \Gamma \rrbracket A$, is defined as: $\llbracket \ell, \Gamma \rrbracket A \stackrel{\text{def}}{=} \llbracket \llbracket \ell, \Gamma \rrbracket A$.*

Note that there is a $\llbracket \llbracket$ before ℓ , but *not* after ℓ . The predicate $\llbracket \ell, \Gamma \rrbracket A$, where the typing Γ is the typing of the whole (prefixed) process **minus** the capability given by ℓ (e.g. an output at a channel a), says the two things:

- (1) Whatever τ -actions the process may have, it is always ready to do ℓ , and nothing else; and
- (2) If ever it does ℓ , it will immediately reach A .

Note (2) does *not* say the resulting process stably satisfies A . Let us illustrate this predicate through simple examples. First, take the inaction, which we assume to be typed under $a : (\tau)$, with $\tau = \uparrow \text{Nat}; \text{end}$. Then we have:

$$\vdash 0 \blacktriangleright \text{noact}(a : (\tau)) \quad (6.2.1)$$

which expands to

$$\vdash 0 \blacktriangleright \llbracket \bar{a}(h) \rrbracket F \wedge \llbracket a(h) \rrbracket F \quad (6.2.2)$$

The formula in (6.2.2) says that an action at a , either input or output, is impossible, hence in effect it asserts that process has no visible actions — as is indeed true with the inaction agent.

As the second example, let $\Gamma' = a : (\tau), b : (\tau')$ and assume $\Gamma' \vdash a(k).P$. Then we “subtract” $a : (\tau)^I$ from Γ' to obtain $\Gamma = a : (\tau)^O, b : (\tau')$, so that, assuming A is the assertion for P , we assert for $a(k).P$ as $\llbracket a(k), \Gamma \rrbracket A$, which expands to:

$$\llbracket a(k) \rrbracket A \wedge \llbracket \bar{a}(k) \rrbracket F \wedge \llbracket b(k') \rrbracket F \wedge \llbracket \bar{b}(k') \rrbracket F \quad (6.2.3)$$

The assertion says that there is no action at b and no output action at a , and that, after an input at a of a fresh session channel k , a state satisfying A follows.

Finally we formalise the idea of “subtracting ℓ from Γ ”, which we write $\Gamma - \ell$. Take ℓ typable under Γ and assume the type of ℓ (using refined IO-types) is $u : \alpha$. We then define $\Gamma - u : \alpha$ as the maximum typing environment Δ such that $\Delta \odot u : \alpha = \Gamma$ (here the maximality is defined, following [51], with respect to the ordering \sqsubseteq induced by composability, given as: $\Delta_1 \sqsubseteq \Delta_2$ if whenever $\Gamma \odot \Delta_2$ is defined, $\Gamma \odot \Delta_1$ is defined, for each Γ). In this case we say Δ is the *result of extracting ℓ from Γ* .

For example, with Γ and Γ' above, and $\ell = a(k)$, the result of extracting ℓ from Γ' is:

$$(a : (\tau), b(\tau')) - a(k) = a : (\tau)^O, b(\tau') \quad (6.2.4)$$

which is the maximum typing environment which, when composed with $a : (\tau)^I$, yields $a : (\tau), b(\tau')$.

Fig. 4 Proof System (Must Modality): with Conc,Res,Var,Rec-ind and If from Fig. 3

$\frac{E \vdash P \blacktriangleright A}{E \vdash a(k).P \blacktriangleright \llbracket a(k), \Gamma \rrbracket A}$	$\frac{E \vdash P \blacktriangleright A}{E \vdash \bar{a}(k).P \blacktriangleright \llbracket \bar{a}(k), \Gamma \rrbracket A}$	Acc,Req
$\frac{E \vdash P \blacktriangleright A}{E \vdash k(x).P \blacktriangleright \forall x. \llbracket kx, \Gamma \rrbracket A}$	$\frac{E \vdash P \blacktriangleright A}{E \vdash \bar{k}\langle e \rangle.P \blacktriangleright \llbracket \bar{k}e, \Gamma \rrbracket A}$	Rcv,Send
$\frac{E \vdash P_i \blacktriangleright A_i \quad \forall i \in I}{E \vdash k \triangleright [l_i : P_i]_{i \in I} \blacktriangleright \bigwedge_{i \in I} \llbracket k \triangleright l_i, \Gamma \rrbracket A_i}$	$\frac{E \vdash P \blacktriangleright A_j}{E \vdash k \triangleleft l_j.P \blacktriangleright \llbracket k \triangleleft l_j, \Gamma \rrbracket A_j}$	Bra,Sel
$\frac{-}{E \vdash \mathbf{0} \blacktriangleright \text{noact}(\Gamma)}$	$\frac{E, X : (\bar{x})A \vdash P \blacktriangleright A \quad A \text{ admissible}}{E \vdash (\mathbf{rec} X(\bar{x}).P)\langle \bar{e} \rangle \blacktriangleright A[\bar{e}/\bar{x}]}$	Inact,Rec-adm

6.3 Proof Rules for the Must Modality

The proof rules for the Must modality are presented in Figure 4, assuming implicit typing environments in each rule, corresponding to the typing rules in Figure 1. We also use Conc,Res,Var, and If from the May proof system in Figure 3. Thus the only differences from Figure 3 are (1) the prefix rules, (2) the inaction rule and (3) the recursion rule, each of which is illustrated below.

(1) Prefix Rules There are six prefix rules: Acc, Req, Rcv, Send, Bra, and Sel. Each of these rules uses an extended Must modal formula following Definition 6.1, of form $\llbracket \ell, \Gamma \rrbracket A$, where ℓ depends on the shape of the prefix used. Assume the implicit typing environment for the conclusion is Δ . Then Γ is given by $\Delta - \ell$, that is Δ minus the capability ℓ , in the sense of the previous subsection.

Using the extended Must modality, each rule precisely specifies what the process cannot do and, if ever it does something it can, what would be the resulting state. Soundness of these rules depend on the avoidance of $\llbracket \ell \rrbracket$ after ℓ in Definition 6.1, since, in each rule, A may not be stable for P (i.e. P may have some τ -actions after which A may no longer hold).

(2) Inaction Rule In the rule for May modality in Figure 3, \top indicates “we cannot say anything about what this process does.” Here in contrast we explicitly assert what this process cannot do. In Inact in Figure 4, we assume Γ is the whole typing environment of $\mathbf{0}$ in the conclusion.

As a small example to show how we can infer properties of processes using these prefix and inaction rules, consider a simple agent $a(k).\bar{k}\langle 3 \rangle.\mathbf{0}$ under the

typing $a : (\downarrow \text{nat}; \text{end})$. Applying these rules we obtain, with $\alpha = (\uparrow \text{nat}; \text{end})^\circ$:

$$\vdash a(k).\bar{k}\langle 3 \rangle.\mathbf{0} \blacktriangleright \llbracket a(k), a : \alpha \rrbracket \llbracket \bar{k}3, a \rrbracket \text{noact}(a, k) \quad (6.3.1)$$

The assertion part can be expanded into:

$$\text{noact}(a : \alpha) \wedge \llbracket a(k) \rrbracket (\text{noact}(a) \wedge \forall i \neq 3. \llbracket \bar{k}i \rrbracket F \wedge \llbracket \bar{k}3 \rrbracket \text{noact}(a, k)) \quad (6.3.2)$$

which says that the process will *not* do any action except an input at a to establish a session k ; then will *not* do any action later with an output 3 via k : and no further actions are possible. This is close to a partial correctness assertion, which constrains a possible final state if the program ever terminates (we shall pursue this informal connection in more detail in §10).

(3) Recursion Rule The recursion rule **Rec-adm** uses the standard notion of admissibility [101], defined formally in §6.7 later (in brief, an admissible formula is one satisfied by the inaction and that which is closed under a limit of a chain of unfolding, or a rational chain: Admissibility has various ramifications which make reasoning easier). As an example usage of **Rec-adm**, we consider:

$$(\mathbf{rec} X(k).\bar{k}1.X\langle k \rangle)\langle k \rangle \quad (6.3.3)$$

from Section 5.2. Now consider the following specification, considered under the typing $k : \mathbf{rect}.\uparrow \text{nat}$.

$$C \equiv (\nu Y(k).\llbracket \bar{k}1, \emptyset \rrbracket Y\langle k \rangle)\langle k \rangle. \quad (6.3.4)$$

We can expand C into:

$$(\nu Y(k).\llbracket \bar{k}1 \rrbracket Y\langle k \rangle \wedge \forall i \neq 1. \llbracket \bar{k}i \rrbracket F)\langle k \rangle \quad (6.3.5)$$

By Proposition 6.5 later C is admissible. We now apply **Rec-adm** via the standard logical equivalence for an unfolding of a fixed point operator:

$$\frac{\begin{array}{l} 1. \dots\dots \\ \hline 2. X : (k)C \vdash \bar{k}1.X\langle k \rangle \blacktriangleright C \end{array}}{3. \vdash (\mathbf{rec} X(k).\bar{k}1.X\langle k \rangle)\langle k \rangle \blacktriangleright C}$$

which in fact is the dual derivation of what we have done in Section 5.2.

Further examples of reasoning which uses the Must modality will be discussed in §6.5 later.

Fig. 5 Proof System (Mixed Modality): with Conc,Res,Var and If from Fig. 3.

$$\begin{array}{c}
 \frac{E \vdash P \blacktriangleright A}{E \vdash a(k).P \blacktriangleright \langle a(k), \Gamma \rangle A} \quad \frac{E \vdash P \blacktriangleright A}{E \vdash \bar{a}(k).P \blacktriangleright \langle \bar{a}(k), \Gamma \rangle A} \quad \text{Acc,Req} \\
 \\
 \frac{E \vdash P \blacktriangleright A}{E \vdash k(x).P \blacktriangleright \forall x. \langle kx, \Gamma \rangle A} \quad \frac{E \vdash P \blacktriangleright A}{E \vdash \bar{k}\langle e \rangle.P \blacktriangleright \langle \bar{k}e, \Gamma \rangle A} \quad \text{Rcv,Send} \\
 \\
 \frac{E \vdash P_i \blacktriangleright A_i \quad \forall i \in I}{E \vdash k \triangleright [l_i : P_i]_{i \in I} \blacktriangleright \bigwedge_{i \in I} \langle k \triangleright l_i, \Gamma \rangle A_i} \quad \frac{E \vdash P \blacktriangleright A_j}{E \vdash k \triangleleft l_j.P \blacktriangleright \langle k \triangleleft l_j, \Gamma \rangle A_j} \quad \text{Bra,Sel} \\
 \\
 \frac{-}{E \vdash \mathbf{0} \blacktriangleright \text{noact}(\Gamma)} \quad \frac{E, X : (\bar{x})X' \langle \bar{x} \rangle \vdash P \blacktriangleright A}{E \vdash (\text{rec } X(\bar{x}).P) \langle \bar{e} \rangle \blacktriangleright (\nu X'(\bar{x}).A) \langle \bar{e} \rangle} \quad \text{Inact,Rec-mix}
 \end{array}$$

6.4 Proof Rules for the Mixed Modality

We now move to the proof system which infers the behaviour of processes using both May and Must modalities. We do this by the use of the Mixed modality (cf. Definition 3.4), strengthened by incorporating the specification for the lack of actions from the extended Must modality (cf. Definition 6.1). In this way, each prefix rule becomes the combination of the corresponding May and Must prefix rules. The strengthened Mix modality is given as follows.

Definition 6.2 (extended Mix modality). *The extended Must modality for ℓ under Γ , written $\langle \ell, \Gamma \rangle A$, is defined as $\langle \ell \rangle A \wedge \llbracket \ell, \Gamma \rrbracket A$.*

Note $\langle \ell \rangle A \wedge \llbracket \ell, \Gamma \rrbracket A$ is equivalent to $\llbracket \llbracket \ell \rrbracket \langle \ell \rangle A \wedge \llbracket \llbracket \ell, \Gamma \rrbracket A$ (since $\llbracket \llbracket \cdot \rrbracket$ distributes over conjunction), entailing both $\langle \ell \rangle A$ and $\llbracket \llbracket \ell, \Gamma \rrbracket A$. The predicate $\langle \ell, \Gamma \rangle A$ says:

- (1) After any (zero or more) τ -actions, the process can do ℓ , and no other; and
- (2) if the process does so, it satisfies A immediately after ℓ .

As we shall see later this modality can fully capture the semantics of prefixes.

The proof system for the Mix modality consists of those rules given in Figure 5 combined with Conc,Res,Var and If from Figure 3. As before, we assume the premise and conclusion of each rule in fact use implicit typing environments, precisely following the corresponding typing rule in Figure 1. The rule Inact is in fact the same as Inact from Figure 4, so in effect the only new rules are the prefix rules and the recursion rule, which we illustrate below.

(1) Prefix Rules The prefix rules use the extended Mix modality, subsuming (in terms of the properties derived) the corresponding rules in the May/Must proof systems. For example, `Send` in Figure 5 says that all and only action this process can do is an output of e via k , and immediately after that action it satisfies A .

The mixed modality is particularly useful for deriving deterministic behaviour of processes. This is useful since many communication-based applications may consist of deterministic local parts together with their non-deterministic composition. In the absence of general summation, the Mix modality is enough to ensure that an action can never be cancelled. Thus in `Send` the formula $(\bar{k}e, \Gamma)A$ fully describes the behaviour of $\bar{k}\langle e \rangle.P$ up to the precision of A w.r.t. P . Since it combines both the May modality and the Must modality, we can derive these modalities as needed from a specification based on mixed modalities.

As a simple example of the use of prefix rules with Mix modality, consider the following process:

$$a(k).\bar{k}\langle 3 \rangle.\mathbf{0} \quad (6.4.1)$$

Let $\alpha = (\downarrow \text{nat}; \text{end})^\circ$ and $\beta = (\downarrow \text{nat}; \text{end})$. Then under $a : \beta$ we can infer for this process with the following conclusion:

$$\vdash a(k).\bar{k}\langle 3 \rangle.\mathbf{0} \blacktriangleright (a(k), a : \alpha) (\bar{k}3, a : \beta) \text{noact}(a : \beta, k : \text{end}) \quad (6.4.2)$$

which says that this agent can initially do an input $a(k)$, and no others; then can do an output $\bar{k}\langle 3 \rangle$, and no other; and has no more actions, precisely specifying all the behaviour of the process in (6.4.1).

(2) Recursion Rule The key difference of the proof system for the mixed modality is the proof rule for recursion¹⁰, due to Larsen [70], reproduced below.

$$\frac{E, X : (\bar{x})\underline{X}\langle \bar{x} \rangle \vdash P \blacktriangleright A}{E \vdash (\mathbf{rec} X(\bar{x}).P)\langle \bar{e} \rangle \blacktriangleright (\mathbf{v} \underline{X}(\bar{x}).A)\langle \bar{e} \rangle} \quad \text{Rec-mix}$$

Note the rule assumes that an assertion variable is assigned to a process variable, assuming an injective function mapping X (a process variable) to \underline{X} (an assertion variable). Thus the fixed-point formula in the conclusion precisely corresponds to the recursion in the syntax of the process. We observe:

- From our experience, for many processes with recursive definitions, a formula thus derived offers a useful specification, see §8 later.
- If we need abstraction, one may use the preceding two recursion rules which give better abstractions for different purposes, for liveness properties (`Rec-ind`) and for safety properties (`Rec-adm`).

We prove later in §7 that `Rec-mix` leads to completeness.

¹⁰ This mixed recursion rule corrects the corresponding rule in our previous presentation [16].

6.5 Axioms and Reasoning for Must Modality

It is well-known in the context of the standard Hennessy-Milner logic (which is based on the strong modalities) that it is a delicate task to derive Must modal actions, especially in the presence of parallel composition [108]. This is because, to derive a Must modal action from a composed process, we should know all possible ways to reach that action and all possible states from that action, which becomes complex especially when that modal action is the silent (τ) action.

In the current setting, the situation becomes worse because we should reason about weak Modal actions. To see this, observe the following equivalence:

$$\llbracket \ell \rrbracket A \equiv \llbracket \llbracket \ell \rrbracket \rrbracket A \quad (6.5.1)$$

Thus to derive $\llbracket \ell \rrbracket A$, one needs to show *all* possible zero or more τ -actions should lead to A , which means we need to go into arbitrarily nested potential interactions between two behaviours inhabiting the formulae.

The present logical framework partly compensates this difficulty through the use of linearly typed actions. This is particularly visible when we combine two prefixed processes, hence their corresponding properties, by parallel composition. First, the general laws of interest are distribution of the must modality over conjunction, i.e. $\llbracket \ell \rrbracket (A \wedge B) \equiv (\llbracket \ell \rrbracket A) \wedge (\llbracket \ell \rrbracket B)$ and $\llbracket \llbracket \rrbracket (A \wedge B) \equiv (\llbracket \llbracket A) \wedge (\llbracket \llbracket B)$.

Now suppose we have a pair of dually prefixed processes at a shared name, say $a(k).P$ and $\bar{a}(k).Q$, satisfying the following assertions:

$$\begin{aligned} A &\stackrel{\text{def}}{=} \llbracket a(k), \Gamma \rrbracket A' \\ B &\stackrel{\text{def}}{=} \llbracket \bar{a}(k), \Delta \rrbracket B' \end{aligned}$$

which use Must assertions. Now suppose $A' \circ B' \supset \llbracket \llbracket C$. Can we infer, from $A \circ B$, the assertion $\llbracket \llbracket C$? Certainly not, since at a there can be interactions with the outside.

However, the linearity in actions changes the situation. If we have two processes $k(x).P$ and $\bar{k}(y).Q$ at a linear channel k , then the assertions for these processes will look like:

$$\begin{aligned} E &\stackrel{\text{def}}{=} \llbracket kv, \Gamma \rrbracket E' \\ G &\stackrel{\text{def}}{=} \llbracket \bar{k}v, \Delta \rrbracket G' \end{aligned}$$

In this case, if we have $E' \circ G' \supset \llbracket \llbracket H$, we can reason as follows, under the typing $\Theta = k : \perp, \Gamma \odot \Delta$:

$$E \circ G \supset \llbracket \llbracket (H \vee \text{noact}(\Theta))$$

The second step is by the following axiom, again assuming $A \circ B \supset \llbracket \rrbracket C$ and under the typing $\text{sbj}(\ell) : \perp$, $\Gamma \odot \Delta$ ($\text{sbj}(\ell)$ is the subject of the transition label ℓ):

$$\llbracket \ell, \Gamma \rrbracket A \circ \llbracket \bar{\ell}, \Delta \rrbracket B \equiv \llbracket \rrbracket (C \vee \text{noact}(\Theta)) \quad (\ell \text{ linear}) \quad (6.5.2)$$

where for simplicity we assume $\text{bn}(\ell) = \emptyset$.

As another rule for linear actions, under the typing $\text{sbj}(\ell) : \tau$, $\Gamma \odot \Delta$ where $\tau \neq \perp$, and again assuming $A \circ B \supset \llbracket \rrbracket C$,

$$(\llbracket \ell \rrbracket A) \circ B \equiv \llbracket \ell \rrbracket C \quad (\ell \text{ linear}) \quad (6.5.3)$$

Validity of this axiom crucially relies on the typing: since ℓ cannot be compensated except by the dual action from the outside, we know it is never consumed by the other party: hence if this action ever happens, we shall still have A and B intact. We also observe the following simple axiom for no-action predicates, typed under $\Gamma \odot \Delta$:

$$\text{noact}(\Gamma) \circ \text{noact}(\Delta) \equiv \text{noact}(\Gamma \odot \Delta) \quad (6.5.4)$$

which says that if there is a behaviour which does nothing at Γ , and if it is composed with another behaviour which does nothing at Δ , and the composed process is typed under $\Gamma \odot \Delta$, then this composed process can do nothing at all of its interaction points.

Below we show a simple example as an illustration of the usage of these axioms. Consider the following processes:

$$\begin{aligned} P &\stackrel{\text{def}}{=} k(x).h!\langle x+1 \rangle.\mathbf{0} \\ Q &\stackrel{\text{def}}{=} \bar{k}\langle 2 \rangle.\mathbf{0} \end{aligned}$$

Then P and Q respectively satisfy, with $G_1 = \text{noact}(k : \text{end}, h : \text{end})$ and $G_2 = \text{noact}(k : \text{end})$,

$$\begin{aligned} A &\stackrel{\text{def}}{=} \llbracket [k2, h] \bar{h}3 \rrbracket G_1 \\ B &\stackrel{\text{def}}{=} \llbracket \bar{k}2 \rrbracket G_2 \end{aligned}$$

which are respectively typed under $k : \downarrow \text{nat}; \text{end}$, $h : \uparrow \text{nat}; \text{end}$ and $k : \uparrow \text{nat}; \text{end}$, $h : \downarrow \text{nat}; \text{end}$ (in fact both G_1 and G_2 are equivalent to \top). By noting $G_1 \circ G_2 \supset G$ where $G = \text{noact}(k : \perp, h : \text{end})$ by (6.5.4) and $\text{noact}(\Gamma) \supset \llbracket \rrbracket \text{noact}(\Gamma)$ by definition, we can apply (6.5.3) to obtain:

$$(\llbracket \bar{h}3 \rrbracket G_1) \circ G_2 \supset \llbracket \bar{h}3 \rrbracket G \quad (6.5.5)$$

By applying (6.5.2) and noting $G \vee G \supset G$, we reach:

$$A \circ B \supset \llbracket \bar{h}3 \rrbracket G \quad (6.5.6)$$

which says that $P|Q$ will surely output 3 via h , and will have no more visible behaviour.

6.6 Axioms and Reasoning for Mix Modality

As we observed, formulae with Mix modal actions, especially those with extended Mix modal actions, can precisely capture the behaviour of prefixed processes hence, through the use of \circ and \vee , the whole behaviour of processes (this informal observation extends to formal results including processes with recursion, as we shall see in Section 7).

Even if so, such formulae are not process themselves: these Mix modal formulae can be weakened through axioms into weaker formulae, taking off branches or conjuncts, adding disjunction, and turning Mix modalities to the corresponding May/Must modalities. These aspects are particularly significant because of the following reasons:

1. The resulting weaker specifications may as well be what we may wish to have for a given process.
2. Recursion rules (Rec-ind and Rec-adm) are most easily applied to formulae which use May and Must formulae.
3. Axioms for the Mix modality, especially those which involve \circ and \vee , tend to be simpler than axioms for May and Must modalities, in close correspondence with the expansion law but with added logical flexibility (e.g. cancellation of irrelevant branches and conjuncts).

Thus, in addition to the use for asserting for the whole behaviour of processes, mixed modal actions can be used for effective reasoning for processes.

As an example, the axiom corresponding to (6.5.2) becomes, assuming $A \circ B \supset \llbracket \rrbracket C$ and under $\Theta = \text{sbj}(\ell) : \perp, \Gamma \odot \Delta$:

$$(\ell, \Gamma)A \circ (\bar{\ell}, \Delta)B \equiv \llbracket \rrbracket C \quad (\ell \text{ linear}) \quad (6.6.1)$$

where again for simplicity we assume $\text{bn}(\ell) = \emptyset$. Note that we do not have to consider the possibility that these dual linear actions do not interact.

When these actions are possibly non-linear, we may use the underlying May axiom to obtain:

$$(\ell, \Gamma)A \circ (\bar{\ell}, \Delta)B \equiv \langle \rangle \vee \text{bn}(\ell).(A \circ B) \quad (6.6.2)$$

which is much weaker than (6.6.1) but is still useful.

We also record, again assuming $A \circ B \supset \llbracket \rrbracket C$ and under $\text{sbj}(\ell) : \tau, \Gamma \odot \Delta$ such that $\tau \neq \perp$,

$$((\ell)A) \circ B \equiv (\ell)C \quad (\ell \text{ linear}) \quad (6.6.3)$$

Again the linearity of ℓ and the overall typing for $\text{sbj}(\ell)$ are essential for the axiom.

We use these axioms to reason about an example similar to the one we used in the previous subsection.

$$\begin{aligned} P &\equiv b(k).k(x).\bar{k}\langle x+1 \rangle.\mathbf{0} \\ Q &\equiv \bar{b}(k).\bar{k}\langle 2 \rangle.k(y).\bar{h}\langle y \rangle.\mathbf{0} \end{aligned}$$

We consider these processes under the obvious minimal typing. Let us define:

$$\begin{aligned} A' &= \forall x^{\text{nat}}. (\bar{b}(k)) (\langle kx \rangle) (\bar{k}x+1) \top \\ B' &= \forall y^{\text{nat}}. (\bar{b}(k)) (\bar{k}2) (\langle ky \rangle) (\bar{h}y) \top \end{aligned}$$

We can then easily infer, using the proof system with mixed modalities:

$$\begin{aligned} \vdash P \blacktriangleright A' \\ \vdash Q \blacktriangleright B' \end{aligned}$$

From which we obtain, using Conc:

$$\vdash P|Q \blacktriangleright A' \circ B' \tag{6.6.4}$$

We can then infer:

$$\begin{aligned} A' \circ B' &\supset (\bar{b}(k)) (\langle k2 \rangle) (\bar{k}3) \top \circ (\bar{b}(k)) (\bar{k}2) (\langle k3 \rangle) (\bar{h}3) \top \\ &\supset \langle \langle \rangle \rangle \nu k. ((\langle k2 \rangle) (\bar{k}3) \top \circ (\bar{k}2) (\langle k3 \rangle) (\bar{h}3) \top) \\ &\supset \langle \langle \rangle \rangle \nu k. (\top \circ (\bar{h}3) \top) \\ &\supset \langle \langle \bar{h}3 \rangle \rangle \top \end{aligned}$$

By applying Conseq, we now obtain

$$\vdash P|Q \blacktriangleright \langle \langle \bar{h}3 \rangle \rangle \top. \tag{6.6.5}$$

In Section 8 we shall see larger inference examples.

6.7 Admissibility

In this subsection we formally define the notion of admissibility, needed for reasoning about the recursion for deriving the must modality (or partial correctness, cf. [54]). It is a property satisfied by formulae which naturally arise in practice as well as theoretical inquiries. For this reason we first present the notion in its most general form: then present syntactic characterisations which help us in reasoning about processes.

For the purpose of defining admissibility, we use the syntactic unfolding and the characterisation of the recursion as its limit [92, 94]. We now define:

Definition 6.3 (unfolding chain). We call a family $\{\Gamma \vdash P_i\}_{0 \leq i}$ of typed processes is an unfolding chain with the limit $(\mathbf{rec} X(\bar{x}).P)\langle \bar{e} \rangle$ or often simply unfolding chain if $P_i \stackrel{\text{def}}{=} Q_i[\bar{e}/\bar{x}]$ such that, for i , Q_i is given by the following induction:

$$\begin{aligned} Q_0 &\stackrel{\text{def}}{=} \mathbf{0} \\ Q_{n+1} &\stackrel{\text{def}}{=} P[(\bar{x})Q_n/X] \end{aligned}$$

We often write P_ω for the limit of an unfolding chain $\{\Gamma \vdash P_i\}_{0 \leq i}$.

Definition 6.4 (admissibility).

1. A property $\Gamma \vdash p$ is admissible if the following two conditions hold: (1) $\mathbf{0} \in p$; and (2) For each unfolding chain $\{\Gamma \vdash P_i\}$ with limit P_ω , if $P_i \in p$ for each i , then $P_\omega \in p$.
2. An interpretation environment ξ is admissible if it maps each assertion variable to an admissible property.
3. A typed formula $\Gamma \vdash A$ is admissible if $\llbracket \Gamma \vdash A \rrbracket \xi$ is an admissible property under each admissible ξ .

Note the conditions above correspond to the shape of the typing rule for recursion $(\mathbf{rec} X(\bar{e}).P)\langle \bar{e} \rangle$: if P^n satisfies A , we can replace X in P with P^n and the result satisfies A , so that we can continue the unfolding indefinitely, which by the given condition ensures that $(\mathbf{rec} X(\bar{x}).P)\langle \bar{e} \rangle$ satisfies A .

Proposition 6.5 (syntactic approximation for admissibility). Let A, B, \dots below be generally open formulae.

1. $X\langle \bar{e} \rangle, \top, \mathbf{F}, e = e'$ and $e \neq e'$ are admissible.
2. If A and B are admissible then so are $A \wedge B$ and $A \vee B$.
3. If $\neg A$ and B are admissible then so is $A \supset B$.
4. If A is admissible then $\forall x.A$ is admissible.
5. If A is admissible then $(\forall X(\bar{x}).A)\langle \bar{e} \rangle$ is admissible
6. If A is admissible then both $[\ell]A$ and $\llbracket \ell \rrbracket A$ are admissible
7. If A is admissible and $A \equiv B$ then B is admissible

Proof. For \forall -recursion, by the third clause of Definition 6.4, its finite approximants are all admissible. For each limit case, we take the conjunction, which is again closed under admissibility. ■

If process recursion is sufficiently well-behaved, we can expand the set of syntactically admissible formulae. The following notions are also important in our later development, when we analyse the construction of characteristic formulae.

Definition 6.6 (prefix). A typed process P is prefixed if $P \equiv Q$ such that Q has one of the following forms:

$$v(k).P', \bar{v}(k).P', k(x).P', \bar{k}\langle e \rangle.P', \bar{k}(va).P', k \triangleleft [l_i.P'_i], \text{ and } k \triangleright l.P', \quad (6.7.1)$$

where $\bar{k}(va).P'$ stands for $(va)\bar{k}\langle a \rangle.P'$. The subject of P is v in the first two cases, k in the last four cases.

Definition 6.7 (well pointed recursion).

1. A typed process $Q \stackrel{\text{def}}{=} (\mathbf{rec} X(\bar{x}).P)\langle \bar{e} \rangle$ is a well-pointed recursion if P is prefixed in the sense of Definition 6.6. In this case, the prefix of recursion Q is the prefix of P .
2. A typed process P is well-pointed if each occurrence of a recursion in P is a well-pointed recursion.

Thus, in well-pointed processes, the body of each recursion is a prefixed process. A key nature of well-pointed processes is that its unfolding in effect takes place only when this prefix is invoked. Thus we have a clear interaction point to observe when the behaviour inside a recursion body is launched and gets engaged in interactions.

Well-pointed recursions are those recursions which naturally arise in practice and in encoding of programs. Further, under many type disciplines for the π -calculus in general and those treated in this paper in particular, well-pointedness is not semantically restrictive, as the following proposition proves.

Proposition 6.8 (well-pointedness suffices). For any well-typed $\Gamma \vdash P$, we can algorithmically construct $\Gamma \vdash Q$ such that $P \approx Q$ and Q is well-pointed.

Proof. We inductively transform each recursion. Consider $(\mathbf{rec} X(\bar{x}\tilde{k}).P)\langle \bar{e}\tilde{k} \rangle$ (setting the parameter to \tilde{k} does not lose generality because linear channels never get coalesced). For illustration we let \tilde{k} be given as kh . We recall, following the standard convention (cf. §2.1), a recursion variable X in P is under some prefix. So we can write:

$$P \equiv (v \tilde{u}) \Pi_{1 \leq i \leq n} Q_i \quad (6.7.2)$$

where each Q_i is prefixed. Note k (say) cannot occur in both Q_i and Q_j ($i \neq j$) by typing¹¹. Assume for simplicity k and h both occur in Q_1 (it should be said that, in this case, X cannot occur in other Q_i , $i \geq 1$: the encoding below caters for all these cases). Further without loss of generality we assume \tilde{u} do not contain a

¹¹ This restriction is not obeyed if we have a delegation or, in general, when a linear name is passed in communication: however in that case we can pass these linear channels as we do for other channels.

linear channel (since if so, say for k' , then we can add two mutual dual session initiations, say $c(k')$ and $\bar{c}(k')$, for a fresh hidden name c , and add a new hiding for c). Then we replace $(\mathbf{rec} X(\tilde{x}\tilde{k}).P)\langle\tilde{e}\rangle$ with

$$(\tilde{b})(\overline{b_0}\langle\tilde{e}\rangle \mid \prod_{0 \leq i \leq n} Q'_i) \quad (6.7.3)$$

where b_0, b_1, \dots, b_n are fresh and $\{Q'_i\}$ are given as follows. Below we write $a(\tilde{x}).P$ for $b(h').h'?\tilde{x}.R$ and $\bar{a}\langle\tilde{e}\rangle.P$ for $\bar{b}(h').h'!\tilde{e}.R$, and use the notation $!a(\tilde{x}).R$ following §2.1, standing for $\mathbf{rec} X.a(h).h(\tilde{x}).(R|X)$ (which is obviously well pointed).

$$\begin{aligned} Q'_0 &\stackrel{\text{def}}{=} !b_0(\tilde{x}).(\nu \tilde{u})(\prod_{1 \leq i \leq n} \bar{b}_i\langle\tilde{x}\tilde{u}\rangle) \\ Q'_1 &\stackrel{\text{def}}{=} (\mathbf{rec} Y(kh).b_1(\tilde{x}\tilde{y}).Q_1[\tilde{u}/\tilde{y}][(\tilde{x}kh)(\overline{b_0}\langle\tilde{x}\rangle|Y\langle kh\rangle)/X])\langle kh\rangle \\ Q'_i &\stackrel{\text{def}}{=} !b_i(\tilde{x}).(Q_i[\tilde{u}/\tilde{y}][(\tilde{x}kh)\overline{b_0}\langle\tilde{x}\rangle/X]) \quad (i \geq 2) \end{aligned}$$

which has the same observable behaviour as the original recursion. By inductively applying this mapping, we obtain well-pointed process up to \approx . ■

Proposition 6.9 (admissibility for well-pointed recursions). *If we only treat well-pointed typed processes and if A and B are admissible then so is $A \circ B$.*

Proof. Since a recursion is now prefixed, its unfoldings can satisfy $A \circ B$ only when one of them includes zero (say B) and one of them is satisfied by the unfoldings (say A). Then we use induction. For (ii), if each unfolding P^i satisfies $\nu x.A$, then (without loss of generality) $(\nu a)Q_i$ satisfies A . ■

Remark 6.10 (admissibility and μ -recursion). The μ -recursion (least fixed point) is *not* generally well-behaved w.r.t. admissibility. For example, the following formula is admissible under the typing $\Gamma, k : \mathbf{rec} t.(\uparrow \text{nat}; t)$, assuming X is interpreted by an admissible property.

$$[[\bar{k}1, \Gamma]]X \quad (6.7.4)$$

However the result of taking its least fixed point:

$$\mu X.([[\bar{k}1, \Gamma]]X) \quad (6.7.5)$$

is *not* admissible: the formula (6.7.5) is satisfied by all *finite* agents which have a sequence of zero or more outputs $\bar{k}1$, and nothing else (we can easily construct such agents under the given typing), but it is not satisfied by its limit, i.e. the recursive behaviour $\mathbf{rec} X.([\bar{k}1].X)$ (to see this, take the collection of all such finite agents as a property, which immediately gives a fixed point). In contrast, the maximal fixed points, ν -recursions, are better behaved with respect to admissibility.

Remark 6.11 (admissibility and ν -quantifier). In the same way the existential quantifier is not well-behaved w.r.t. admissibility, the ν -quantifier is not well-behaved w.r.t. admissibility. However if the quantifier can identify the hidden name sufficiently clearly, we can use the induction hypothesis. This is the case when, for example, we consider Must characteristic formulae in §7.3.2.

7 Soundness and Completeness of Proof Rules

This section proves soundness and completeness of the three proof systems. The proofs of completeness largely follow [54], based on the syntactic derivation of characteristic formulae. In addition, we use the close relationship between proof rules and the derivation rules for characteristic formulae for proving soundness. For these reasons the technical development starts from the introduction and study of the derivation rules for characteristic formulae.

7.1 Derivation of Characteristic Formulae

Characteristic formulae [40, 105, 106] precisely characterise the semantics of processes. There are three systems for the derivation of characteristic formulae. Each system consists of a set of the derivation rules which exactly follow the syntax of typed processes, centring on distinct modality (May, Must and Mixed). The systems however differ in terms of the main modality they use. Thus each derivation system defines a function from a typed process (including its typing environment) to an assertion in Hennessy-Milner logic. The rules generally follow the proof rules in the previous two sections, using \circ and ν , with differences in recursion rules and conditional. Because of the use of \circ and ν , the derived formulae may not directly describe the behaviour of agents: rather it leaves the most difficult part, how a behaviour arises from parallel composition and hiding, implicit. Nevertheless they serve significant purposes:

1. The results lead to (relative) completeness of the three proof systems, and also contribute to soundness.
2. The derived formulae characterise three representative semantic preorders/equivalence, shedding light on the semantic nature of the proof systems.
3. The completeness results also suggest that the only complexity to derive full modal specification of processes come from these two operators.

Thus the derivation of characteristic formulae offer significant information about the present logic, both in terms of the nature of its underlying semantic universe and in terms of proof systems.

Convention 7.1 (typing environments in derivation rules). When deriving characteristic formulae for Must and Mixed modalities, the typing plays a fundamental role. For this purpose we assume that we weaken the typing for shared names as far as possible, so that we can specify the lack of actions on those names as necessary. For example, when we derive a formula for the composition of $a(k).P$ and $b(h).Q$, we start from typing *each* of them with the environment whose domain is $\{a, b\}$, which allows us to specify $a(k).P$ does not have a b -action and $b(h).Q$ does not have an a -action.

Fig. 6 Derivation of Characteristic Formula (May Modality)

$$\begin{array}{c}
 \frac{E \vdash_{\text{may}}^* P \blacktriangleright A}{E \vdash_{\text{may}}^* a(k).P \blacktriangleright \langle\langle a(k) \rangle\rangle A} \quad \frac{E \vdash_{\text{may}}^* P}{E \vdash_{\text{may}}^* \bar{a}(k).P \blacktriangleright \langle\langle \bar{a}(k) \rangle\rangle A} \quad \text{Acc,Req} \\
 \\
 \frac{E \vdash_{\text{may}}^* P \blacktriangleright A}{E \vdash_{\text{may}}^* k(x).P \blacktriangleright \forall x. \langle\langle kx \rangle\rangle A} \quad \frac{E \vdash_{\text{may}}^* P \blacktriangleright A}{E \vdash_{\text{may}}^* \bar{k}(e).P \blacktriangleright \langle\langle \bar{k}e \rangle\rangle A} \quad \text{Send, Rcv} \\
 \\
 \frac{E \vdash_{\text{may}}^* P_i \blacktriangleright A_i \quad \forall i \in I}{E \vdash_{\text{may}}^* k \triangleright [l_i : P_i]_{i \in I} \blacktriangleright \bigwedge_{i \in I} \langle\langle k \triangleright l_i \rangle\rangle A_i} \quad \frac{E \vdash_{\text{may}}^* P \blacktriangleright A_j}{E \vdash_{\text{may}}^* k \triangleleft l_j.P \blacktriangleright \langle\langle k \triangleleft l_j \rangle\rangle A_j} \\
 \text{Bra, Sel} \\
 \\
 \frac{E_i \vdash_{\text{may}}^* P_i \blacktriangleright A_i \quad i = 1, 2}{E_1 \cup E_2 \vdash_{\text{may}}^* P_1 \mid P_2 \blacktriangleright A_1 \circ A_2} \quad \frac{E \vdash_{\text{may}}^* P \blacktriangleright A}{E \vdash_{\text{may}}^* (\nu u)P \blacktriangleright \forall x. A[x/u]} \quad \frac{-}{E \vdash_{\text{may}}^* \mathbf{0} \blacktriangleright \top} \\
 \text{Conc, Res, Inact} \\
 \\
 \frac{-}{E, X : (\tilde{x})\underline{X} \vdash_{\text{may}}^* X \langle \tilde{e} \rangle \blacktriangleright \underline{X} \langle \tilde{e} \rangle} \quad \frac{E, X : (\tilde{x})\underline{X} \vdash_{\text{may}}^* P \blacktriangleright A}{E \vdash_{\text{may}}^* (\mathbf{rec} X(\tilde{x}).P) \langle \tilde{e} \rangle \blacktriangleright (\nu \underline{X}(\tilde{x}).A) \langle \tilde{e} \rangle} \\
 \text{Var, Rec} \\
 \\
 \frac{E \vdash_{\text{may}}^* P_1 \blacktriangleright A_1 \quad E \vdash_{\text{may}}^* P_2 \blacktriangleright A_2}{E \vdash_{\text{may}}^* \text{if } e \text{ then } P_1 \text{ else } P_2 \blacktriangleright (e \wedge A_1) \vee (\neg e \wedge A_2)} \quad \text{If}
 \end{array}$$

Derivation Rules (1): May Modality The derivation rules for the May modality are given in Figure 6. The sequent $E \vdash_{\text{may}}^* P \blacktriangleright A$ is calculated purely based on structure of processes, even without using typing environments (except annotation of logical variables). That is, when $E \vdash_{\text{may}}^* P \blacktriangleright A$ is derived from these derivation rules, A (and E) is uniquely determined by P .

The derivation of $\vdash_{\text{may}}^* P \blacktriangleright A$ assigns a property A in which P is the *minimal* element satisfying A with respect to the May preorder \sqsubseteq_{may} (we later validate this claim). Property A gives a positive description of what a process can do using the May modality, rather than what it does not do: for example Inact assigns the least informative specification, \top , to P , since $\mathbf{0}$ cannot do anything.

In Var rule, we assign a formula \underline{X} (an assertion variable) to X (a process variable), assuming this map is injective (cf. (Rec-mix), page Rec-mix). In Rec, we fold both the process and the formula in sync. This folding is done via ν -recursion. We later show it maintains the minimality of the target process inside the property represented by the constructed formula.

Fig. 7 Derivation of Characteristic Formula (Must Modality)

$$\begin{array}{c}
\frac{E \vdash_{must}^* P \blacktriangleright A}{E \vdash_{must}^* a(k).P \blacktriangleright \llbracket a(k), \Gamma \rrbracket A} \quad \frac{E \vdash_{must}^* P}{E \vdash_{must}^* \bar{a}(k).P \blacktriangleright \llbracket \bar{a}(k), \Gamma \rrbracket A} \quad \text{Acc, Req} \\
\frac{E \vdash_{must}^* P \blacktriangleright A}{E \vdash_{must}^* k(x).P \blacktriangleright \forall x. \llbracket kx, \Gamma \rrbracket A} \quad \frac{E \vdash_{must}^* P \blacktriangleright A}{E \vdash_{must}^* \bar{k}(e).P \blacktriangleright \llbracket \bar{k}e, \Gamma \rrbracket A} \quad \text{Send, Rcv} \\
\frac{E \vdash_{must}^* P_i \blacktriangleright A_i \quad \forall i \in I}{E \vdash_{must}^* k \triangleright [l_i : P_i]_{i \in I} \blacktriangleright \bigwedge_{i \in I} \llbracket k \triangleright l_i, \Gamma \rrbracket A_i} \quad \frac{E \vdash_{must}^* P \blacktriangleright A_j}{E \vdash_{must}^* k \triangleleft l_j.P \blacktriangleright \llbracket k \triangleleft l_j, \Gamma \rrbracket A_j} \quad \text{Bra, Sel} \\
\frac{E_i \vdash_{must}^* P_i \blacktriangleright A_i \quad i = 1, 2}{E_1 \cup E_2 \vdash_{must}^* P_1 | P_2 \blacktriangleright A_1 \circ A_2} \quad \frac{E \vdash_{must}^* P \blacktriangleright A}{E \vdash_{must}^* (\nu u)P \blacktriangleright \nu x.A[x/u]} \quad \frac{-}{\vdash_{must}^* \mathbf{0} \blacktriangleright \text{noact}(\Gamma)} \quad \text{Conc, Res, Inact} \\
\frac{-}{X : (\tilde{x})\underline{X} \langle \tilde{x} \rangle \vdash_{must}^* X \langle \tilde{e} \rangle \blacktriangleright \underline{X} \langle \tilde{e} \rangle} \quad \text{Var} \\
\frac{E, X : (\tilde{x})\underline{X} \vdash_{must}^* P \blacktriangleright A}{E \vdash_{must}^* (\text{rec } X(\tilde{x}).P) \langle \tilde{e} \rangle \blacktriangleright (\nu X(\tilde{x}).A) \langle \tilde{e} \rangle} \quad \text{Rec} \\
\frac{E \vdash_{must}^* P_1 \blacktriangleright A_1 \quad E \vdash_{must}^* P_2 \blacktriangleright A_2}{E \vdash_{must}^* \text{if } e \text{ then } P_1 \text{ else } P_2 \blacktriangleright (e \wedge A_1) \vee (\neg e \wedge A_2)} \quad \text{If}
\end{array}$$

Derivation Rules (2): Must Modality The derivation rules of characteristic formulae for the Must modality are given in Figure 7. We list all rules, though Conc, Res and Res are common with Figure 6. In the prefix rules and Inact, we assume Γ is given as in the Must proof system (Figure 3, page 36), using an implicitly given typing environment and following Convention 7.1. Var adds a disjunction with the no-action predicate for Γ (Γ is implicitly assigned to X from the typing of the recursive agent when this process is closed: such Γ can always be chosen uniquely once we have a type derivation of the whole process).

Rec uses a maximal fixed point to obtain a partial correctness property for recursion,¹² giving a sufficiently constrained property as far as the recursion conforms to a semantically non-restrictive syntactic condition. This condition, called *well-guardedness*, is discussed in §7.2 later.

When we derive $E \vdash_{must}^* P \blacktriangleright A$, the formula A is intended to be a property in which P is a maximal element w.r.t. \sqsubseteq_{must} . This claim will be verified later.

¹² A previous version uses a least fixed point and a mix characteristic formula. The new rule gives a better closure property and, in particular, admissibility.

Fig. 8 Derivation of Characteristic Formula (Mixed Modality)

$$\begin{array}{c}
 \frac{E \vdash_{mix}^* P \blacktriangleright A}{E \vdash_{mix}^* a(k).P \blacktriangleright (a(k), \Gamma)A} \quad \frac{E \vdash_{mix}^* P}{E \vdash_{mix}^* \bar{a}(k).P \blacktriangleright (\bar{a}(k), \Gamma)A} \quad \text{Acc, Req} \\
 \\
 \frac{E \vdash_{mix}^* P \blacktriangleright A}{E \vdash_{mix}^* k(x).P \blacktriangleright \forall x.(kx, \Gamma)A} \quad \frac{E \vdash_{mix}^* P \blacktriangleright A}{E \vdash_{mix}^* \bar{k}\langle e \rangle.P \blacktriangleright (\bar{k}e, \Gamma)A} \quad \text{Send, Rcv} \\
 \\
 \frac{E \vdash_{mix}^* P_i \blacktriangleright A_i \quad \forall i \in I}{E \vdash_{mix}^* k \triangleright [l_i : P_i]_{i \in I} \blacktriangleright \bigwedge_{i \in I} (k \triangleright l_i, \Gamma)A_i} \quad \frac{E \vdash_{mix}^* P \blacktriangleright A_j}{E \vdash_{mix}^* k \triangleleft l_j.P \blacktriangleright (k \triangleleft l_j, \Gamma)A_j} \quad \text{Bra, Sel} \\
 \\
 \frac{E \vdash_{mix}^* P_i \blacktriangleright A_i \quad i = 1, 2}{E \vdash_{mix}^* P_1 | P_2 \blacktriangleright A_1 \circ A_2} \quad \frac{E \vdash_{mix}^* P \blacktriangleright A}{E \vdash_{mix}^* (\nu u)P \blacktriangleright \forall x.A[x/u]} \quad \frac{-}{\vdash_{mix}^* \mathbf{0} \blacktriangleright \text{noact}(\Gamma)} \quad \text{Conc, Res, Inact} \\
 \\
 \frac{-}{X : (\tilde{x})\underline{X} \vdash \underline{X}\langle \tilde{e} \rangle \blacktriangleright \underline{X}\langle \tilde{e} \rangle} \quad \frac{E, X : (\tilde{x})\underline{X} \vdash_{mix}^* P \blacktriangleright A}{E \vdash_{mix}^* (\mathbf{rec} X(\tilde{x}).P)\langle \tilde{e} \rangle \blacktriangleright (\nu \underline{X}(\tilde{x}).A)\langle \tilde{e} \rangle} \quad \text{Var, Rec} \\
 \\
 \frac{E \vdash_{mix}^* P_1 \blacktriangleright A_1 \quad E \vdash_{mix}^* P_2 \blacktriangleright A_2}{E \vdash_{mix}^* \text{if } e \text{ then } P_1 \text{ else } P_2 \blacktriangleright (e \wedge A_1) \vee (\neg e \wedge A_2)} \quad \text{If}
 \end{array}$$

Derivation Rules (3): Mixed Modality Finally the derivation rules for the mixed modality are given in Figure 8. These rules are identical with those for the Must derivation rules in Figure 7 except for the prefix rules, which are identical with the proof rules for the Mix modality in Figure 5, page 47; and the pair of the variable rule (Var) and the recursion rule (Rec), which are identical with the corresponding rules for characteristic formulae for May modality in Figure 6.

Rec can construct a characteristic formula which exactly pinpoints a recursion $(\mathbf{rec} X(\tilde{x}).P)\langle \tilde{e} \rangle$ based on the assumption A in the premise is already sufficiently restrictive, pinpointing the behaviour of P . The structure of A follows the structure of P , and this assumption is generally true for a wide class of typable processes. We shall later formally stipulate the shape of processes which can guarantee characterisation. As we shall discuss there, this does not restrict the expressive power of typed processes, since the excluded shape can always be recovered through composition.

We illustrate the use of Rec by a simple example. Consider a process:

$$P \stackrel{\text{def}}{=} k!1.X\langle k \rangle \quad (7.1.1)$$

For this process we can easily build its characteristic formula, for an appropriate Γ such that $\Gamma, k : \mathbf{rect}. \uparrow \text{nat}; t$ types P :

$$A \stackrel{\text{def}}{=} (\bar{k}1, \Gamma)X \quad (7.1.2)$$

If we set $\Gamma = a : (\tau)$, A can be expanded as:

$$[[\!]](\langle \bar{k}1 \rangle \top \wedge [\bar{k}1]X \langle k \rangle \wedge \forall x \neq 1. [\bar{k}1]F \wedge \text{noact}(a : (\tau))) \quad (7.1.3)$$

From P , we build the recursive process $(\mathbf{rec}X(k).P)\langle k \rangle$. For its characteristic formula, we obtain $(\mathbf{v}X(k).A)\langle k \rangle$ by **Rec**. Using the expanded form (7.1.3), we obtain the following series of unfolded fixed points:

$$\begin{aligned} (\mathbf{v}^0X(k).A)\langle k \rangle &\stackrel{\text{def}}{=} \top \\ (\mathbf{v}^1X(k).A)\langle k \rangle &\stackrel{\text{def}}{=} [[\!]](\langle \bar{k}1 \rangle \top \wedge [\bar{k}1]\top \wedge \forall x \neq 1. [\bar{k}1]F \wedge \text{noact}(a : (\tau))) \\ &\dots \\ (\mathbf{v}^{n+1}X(k).A)\langle k \rangle &\stackrel{\text{def}}{=} [[\!]](\langle \bar{k}1 \rangle \top \wedge [\bar{k}1]A^n \wedge \forall x \neq 1. [\bar{k}1]F \wedge \text{noact}(a : (\tau))) \end{aligned}$$

Then $(\mathbf{v}^0X(k).A)\langle k \rangle$ denotes the intersection of the properties specified by these formulae. Since this formula is already a fixed point for A , it in fact coincides with $(\mathbf{v}X(k).A)\langle k \rangle$. By the construction, it specifies that there is only the output of 1 through k initially; and then it returns to exactly the same state. Thus this formula exactly describes the behaviour of $(\mathbf{rec}X(k).P)\langle k \rangle$, and nothing more: if another typed process satisfies $(\mathbf{v}X(k).A)\langle k \rangle$, then that process should be bisimilar to (7.1.2).

As a simplest example of a process for which **Rec** does *not* give the desired single semantic point, consider the trivial recursive process $\mathbf{rec}X.X$, typed under Γ .¹³ This is bisimilar to $\mathbf{0}$, but the corresponding maximal fixed point formula, $\mathbf{v}X.X$, gives the *whole* set of processes typed under Γ . As another example, writing e.g. $a.P$ for $a(k).P$ with k not occurring in P and using the replication notation, consider the recursive process $\mathbf{rec}X.(a.X \mid \bar{a}.0)$. The formula constructed for this process is, with an appropriate Γ :

$$\mathbf{v}X.((\langle a, \Gamma \rangle X \circ (\mathbf{v}Y.(\bar{a}, \Gamma)Y))) \quad (7.1.4)$$

which in fact is equivalent to \top (to see this, expand and check the first approximant of (7.1.4) is \top , hence the chain is already stationary at $\mathbf{0}$).

As before, these rules build judgements of form $E \vdash_{\text{mix}}^* P \blacktriangleright A$ (under implicit typing environments, following Convention 7.1). As noted, under a mild (and semantically non-restrictive) restriction on the shape of the body of the process, the constructed process A pinpoints a single semantic point, P , up to \approx . In other words, A captures the whole behaviour of P , and nothing else.

¹³ This recursion is not guarded but we can easily construct an equivalent guarded (but not well-pointed) version, such as $(\mathbf{v}a)(\mathbf{rec}X.(a.X \mid \bar{a}))$, to which the identical arguments apply.

7.2 Well-Guarded Recursion

The characterisation results for Must and Mix characteristic formulae are stated under a mild syntactic condition on recursive processes called *well-guardedness* (which strengthens the well-pointedness discussed in §6.7). We study this condition in this subsection before entering the next two subsections, showing, among others, the condition is not semantically restrictive.

Intuitively, a recursive process is well-guarded if it is well-pointed and moreover, in its recursion body, any of its prefix can only be invoked from the outside. The formal definition follows. Below the *subject* of a prefix is the initial channel/name of a prefix. A *communication redex pair* is a pair of prefixes which induce a reduction in a process. *Below and henceforth we always assume mentioned transitions, substitutions, etc. are well-typed.*

Definition 7.2 (well-guarded recursion). Let $Q \stackrel{\text{def}}{=} (\mathbf{rec} X(\bar{u}).Q_0)\langle \bar{v} \rangle$ be well-pointed below.

1. Q is a well-guarded recursion if whenever $Q \xrightarrow{s} Q'$ for some action sequence s , the process Q' does not contain two active prefixes at the same subject.
2. Q is a weakly well-guarded recursion if whenever $Q \xrightarrow{s} Q'$ for some action sequence s , and if the subject channel, say u , of the prefix of each recursion in Q' occurs actively (i.e. not under any prefix), then u is never part of a communication redex pair in Q' .
3. We say a typed process is well-guarded (resp. weakly well-guarded) if all recursions occurring in it are well-guarded recursions (resp. weakly well-guarded recursions).

Note a recursion occurring in a well-guarded recursion is always well-guarded since, if not, the property in (i) does not hold. Note also a well-guarded process is immediately weakly well-guarded.

Example 7.3 (well-guarded recursions).

1. (sequential recursion, 1) $P_1 \stackrel{\text{def}}{=} (\mathbf{rec} X.a(k).\bar{k}1.X)$ is well-guarded. Under arbitrary transitions the residual has the form $a(k).\bar{k}1.a(k).\bar{k}1\dots$ hence each $a(k)$ can only interact with the outside.
2. (sequential recursion, 2) A process P is *sequential* if it starts from a prefix and does not contain any parallel composition. For any sequential P , the recursion $(\mathbf{rec} X(\bar{x}).P)\langle \bar{e} \rangle$ is well-guarded.
3. (replication, 1) $P_2 \stackrel{\text{def}}{=} \mathbf{rec} X.(a(k).(\bar{k}1|X))$ is well-guarded. Though its recursion body is not sequential, after arbitrary transitions it has the form $a(k).(\bar{k}1|a(k).(\bar{k}1|a(k).\bar{k}1|\dots))$, so $a(k)$ can only interact with the outside.

4. (replication, 2) A well-typed process of the form $\mathbf{rec}X.(a(k).(P|X))$ (which we may write $!a(k).P$) is well-guarded whenever P does not contain an output shared name a or a variable as a subject.
5. Let P_3 be $\mathbf{rec}X(k).k \triangleright [l_1 : \bar{k}\langle 1 \rangle.X\langle k \rangle, l_2 : \mathbf{0}]$. Then P_2 is well-guarded. In fact, in the present type discipline, any typed process whose recursion has a linearly typed subject not typed \perp in the enclosing context, is well-guarded, since k , by the typing, can never be part of a redex or co-occur inside the recursion body.

Proposition 7.4 (well-guardedness suffices). *For any well-typed $\Gamma \vdash P$, we can algorithmically construct $\Gamma \vdash Q$ such that $P \approx Q$ and Q is well-guarded.*

Proof. By noting that, in the translation into well-pointed processes in the proof of Proposition 6.8 (page 54), each prefixed Q'_i can in fact be expelled to the outermost of the whole process, thus dispensing with the need for nesting. More precisely, assume we have a process which can be written as follows:

$$\mathcal{C}[(\mathbf{rec}X(\tilde{x}\tilde{k}).P)\langle \tilde{e}\tilde{k} \rangle] \quad (7.2.1)$$

where $\mathcal{C}[\]$ is a context, such that the hole of $\mathcal{C}[\]$ is not inside another recursion (so this is an outermost recursion: there can be two or more such outermost recursions in a process, which are treatable in the same way). Then the translation of this recursion into a well-guarded one can be done as follows, using Q'_i given in the proof of Proposition 6.8, 54::

$$(\nu \tilde{b})(\mathcal{C}[\bar{b}_0\langle \tilde{e} \rangle] \mid \Pi_{0 \leq i \leq n} Q'_i) \quad (7.2.2)$$

The translation starts from the outermost recursions in a given process as above, then inductively we apply the translation to each Q'_i above, we do the same, which always allows these prefixed recursions (note that, inside each recursion, there can be a sequence of other recursions) stay in a reduction context. Each generated recursion is prefixed by a fresh channel which can never be communicated hence whose dual can only occur outside of its recursion body. ■

Remark 7.5 (well-guarded processes). The mapping given in the proof of Proposition 7.4 may not lead to naturally specified processes by itself, but often the underlying idea can be used for obtaining a natural translation to externalise an inner recursion. As a simple but typical example, consider

$$\mathbf{rec}X.a(k).(X|\bar{a}(k).\mathbf{0}) \quad (7.2.3)$$

which is not well-guarded. But we can turn it into

$$(\nu b)((\mathbf{rec}X.a(k).(X|\bar{b}(h).\mathbf{0}) \mid (\mathbf{rec}X.b(h).(X|\bar{a}(k).\mathbf{0}))) \quad (7.2.4)$$

which is well-guarded. A similar natural translation may be possible in many classes of typed processes other than what we are treating in the present paper.

Well-guardedness has a sound syntactic characterisation which is useful to check (weak) well-guardedness.

Proposition 7.6 (syntactic well-guardedness).

1. *Suppose P is a well-pointed recursion such that its body never contain a redex under any well-typed substitution, similarly inductively for recursions in P , if any. Then P is a well-guarded recursion.*
2. *Suppose P is a well-pointed recursion such that either its subject is linearly typed or, if not, its subject is free and its dual prefix never occurs in any sub-term of the recursion body under any well-typed substitution, similarly inductively for recursions in P , if any. Then P is a weakly well-guarded recursion.*

Proof. (i) is immediate. For (ii), as we already observed in Example 7.3 (v), no linearly typed channel in such a prefix can participate in a reduction inside a residual of arbitrary transitions. ■

We can easily check that all recursions in Example 7.3 satisfy the condition (i) stated in Proposition 7.6.

A key property of well-guarded processes follows, after a definition.

Definition 7.7. *A well-guarded closed process P is enough unfolded when there is a well-guarded closed process Q such that P is the result of unfolding each recursion occurring in Q more than once.*

Note unfoldings can be performed through \equiv . Below ℓ range over τ and visible actions. s is a sequence of such actions.

Proposition 7.8 (strong transition in well-guarded process). *Let P be a closed, well-guarded recursion and $P \xrightarrow{s} Q$ for a sequence of actions s . Then the following two clauses hold.*

1. *Q is well-guarded.*
2. *If $R \equiv Q$ such that R is enough unfolded, and $R \xrightarrow{\ell} R'$, then this transition is inferred without the recursion rule in Figure 2.*

Proof. (1) is immediate because well-guardedness is closed under substitutions and multi-step transitions, by Definition 7.6. For (2) notice that, by being enough unfolded, each recursion is now (possibly indirectly) under some prefix. Thus if there is any transition including τ -action, it can be inferred without reaching the recursion body of any recursion. ■

Corollary 7.9 (weak transition in well-guarded process). *Let P be a closed, well-guarded recursion and $P \xrightarrow{s} \equiv R$ for a sequence of actions s such that R is enough unfolded. Let $R \xrightarrow{\ell} R'$ with ℓ being τ -action or a visible action. Then this weak transition, consisting of zero or more strong transitions, is inferred without applying the recursion rule in Figure 2.*

Proof. Immediate from Proposition 7.8. ■

Observe that the proof of Proposition 7.8 above only uses weak well-guardedness: thus this Proposition and Corollary 7.9 hold for weakly well-guarded processes, though in the following we only use these properties for well-guarded processes.

7.3 Properties of Derivation Systems

7.3.1 May characteristic formulae The first two derivation systems introduced above respectively derive, for a given typed process, what correspond to total correctness properties (\vdash_{may}^*) and partial correctness properties (\vdash_{must}^*) in Hoare logic, following [54]: total correctness specifies positive information about actions, i.e. what the process *can do*, whereas partial correctness specifies negative information, i.e. what actions a process *does not do*, inductively at each state.

Lemma 7.10 (May characteristic formulae). *Let P and Q be closed and typable under the same typing and $\vdash_{may}^* P \blacktriangleright A$. Then $\models P \blacktriangleright A$ and, for each Q such that $\models Q \blacktriangleright A$, we have $P \sqsubseteq_{may} Q$.*

We first prove the soundness.

Proposition 7.11 (May characteristic formula: soundness). *$E \vdash_{may}^* P \blacktriangleright A$ implies $\models P \blacktriangleright A$.*

Proof. By rule induction. We only show the case for Rec, leaving the remaining cases to Appendix A.1. Suppose

$$E, X : (\tilde{x})\underline{X} \models P \blacktriangleright A. \quad (7.3.1)$$

Now consider the following chain:

$$\begin{aligned} (\mathbf{v}^0 X(\tilde{x}).A)\langle \tilde{e} \rangle &\stackrel{\text{def}}{=} \top \\ &\dots \\ (\mathbf{v}^{n+1} X(\tilde{x}).A)\langle \tilde{e} \rangle &\stackrel{\text{def}}{=} A[(\mathbf{v}^n X(\tilde{x}).A)/X][\tilde{e}/\tilde{x}] \end{aligned}$$

By induction and using (7.3.1), we obtain:

$$E \models (\mathbf{rec} X(\tilde{x}).P)\langle\tilde{e}\rangle \blacktriangleright (\mathbf{v}^n X(\tilde{x}).A)\tilde{e} \quad (7.3.2)$$

(which is immediately true for $n = 0$; and if it is true for $n = m$, then apply (7.3.1) and note $P[\mathbf{rec} X(\tilde{x}).P/X][\tilde{e}/\tilde{x}] \equiv (\mathbf{rec} X(\tilde{x}).P)\langle\tilde{e}\rangle$ to obtain the case for $n = m + 1$). Since (7.3.2) entails, under a fixed interpretation environment, each $(\mathbf{v}^n X(\tilde{x}).A)\tilde{e}$ as a property includes $(\mathbf{rec} X(\tilde{x}).P)\langle\tilde{e}\rangle$, the same holds for $(\mathbf{v}^0 X(\tilde{x}).A)\tilde{e}$. For all ordinals the same argument applies, so that we obtain:

$$E \models (\mathbf{rec} X(\tilde{x}).P)\langle\tilde{e}\rangle \blacktriangleright (\mathbf{v} X(\tilde{x}).A)\langle\tilde{e}\rangle \quad (7.3.3)$$

as required. ■

We now turn to the minimality.

Definition 7.12 (May-closed properties). *We say a property is May-closed if it has the minimal element w.r.t. \sqsubseteq_{may} . We say a parametrised property is May-closed if its all well-typed instantiations are May-closed. ξ is May-closed when ξ maps each assertion to a May-closed parametrised property.*

Proposition 7.13 (May characteristic formula: minimality). *If $E \vdash_{\text{may}}^* P \blacktriangleright A$ then, for each May-closed ξ , $\llbracket A \rrbracket \xi$ is also May-closed, and, moreover, $P\xi\sigma$ is a minimal element of $\llbracket A \rrbracket \xi$ w.r.t. \sqsubseteq_{may} whenever σ assigns to each say $X\langle\tilde{e}\rangle$ a minimal element of $\xi(\underline{X})\langle\tilde{e}\rangle$.*

Proof. By induction on the size of the derivation tree, where the size of a derivation tree is given as follows: for each application of a recursion rule, we use the next limit ordinal as its size of the derivation tree (this is enough since the maximal fixed point we shall use is stationary at the first limit ordinal ω , as we shall see soon). We show the case for **Rec**, which is the only non-trivial case, leaving other cases to Appendix A.2. Consider assigning respectively \underline{X} to **T** and $\mathbf{0}$ to X , where immediately **T** is May-closed and $\mathbf{0}$ is its minimum element. This bootstraps approximations in both processes and fixed point formulae, as follows (below the notation $[(\tilde{x})P^n/X]$ denotes the obvious substitution involving instantiation of variables).

- We set P^n as: $P^0 = \mathbf{0}$, $P^{n+1} = P[(\tilde{x})P^n/X]$. We also write P^ω for $(\mathbf{rec} X(\tilde{x}).P)\langle\tilde{x}\rangle$. We write $P^n\langle\tilde{e}\rangle$ for $P^n[\tilde{e}/\tilde{x}]$.
- We set $\mathbf{v}^n.\underline{X}(\tilde{x}).A$ as the standard n -th approximant with n ranging over natural numbers, i.e. $(\mathbf{v}^0.\underline{X}(\tilde{x}).A)\langle\tilde{x}\rangle \equiv \mathbf{T}$ and $(\mathbf{v}^{n+1}.\underline{X}(\tilde{x}).A)\langle\tilde{x}\rangle \equiv A[(\mathbf{v}^n.\underline{X}(\tilde{x}).A)/X]$.

Define the *recursion size* of a formula A as the deepest nesting of fixed point operators in A . We claim:

Claim. Suppose, up to and including the recursion size m , the Proposition above holds. Then for A whose recursion size is m , and for each natural number n and under any May-closed interpretation environment, we have, for each type-conforming parameter \tilde{e} :

1. $P^n \langle \tilde{e} \rangle \blacktriangleright (\mathbf{v}^n.\underline{X}(\tilde{x}).A) \langle \tilde{e} \rangle$.
2. P^n is a minimal element of $(\mathbf{v}^n.X(\tilde{x}).A) \langle \tilde{e} \rangle$ w.r.t. \sqsubseteq_{may} and $(\mathbf{v}^n.\underline{X}(\tilde{x}).A) \langle \tilde{e} \rangle$ is May-closed.

Proof of Claim. Both are obvious when $n = 0$, regardless of the environment. For induction, note that $(\mathbf{v}^{n+1}.\underline{X}(\tilde{x}).A) \langle \tilde{e} \rangle$ is the May characteristic formula of P^{n+1} , and if a fixed point formula occurs in it, its depth is m or less, hence by assumption and other cases of Proposition we directly obtain the result. (end of the proof of Claim)

Now we show the above Claim entails the statement for Rec, by induction of the derivation trees. Assume the rule constructs a fixed point formula whose recursion size is $m + 1$. By induction we can apply the Claim above for the first ω approximants. Now consider the first limit ordinal ω . The traces of $P^\omega \langle \tilde{e} \rangle$ are the union of those of $\{P^0 \langle \tilde{e} \rangle, P^1 \langle \tilde{e} \rangle, \dots, P^n \langle \tilde{e} \rangle, \dots\}$ while $(\mathbf{v}^\omega.\underline{X}(\tilde{x}).A) \langle \tilde{e} \rangle$ is the intersection of $(\mathbf{v}^n.\underline{X}(\tilde{x}).A) \langle \tilde{e} \rangle$ each of which contains $P^n \langle \tilde{e} \rangle$; hence their intersection still contains all the traces. Hence we know $P^\omega \langle \tilde{e} \rangle$ is minimal in $(\mathbf{v}^\omega.\underline{X}(\tilde{x}).A) \langle \tilde{e} \rangle$, the intersection of all the predecessors, by which $(\mathbf{v}^\omega.\underline{X}(\tilde{x}).A) \langle \tilde{e} \rangle$ is May-closed. From here on each approximant always contains $P^\omega \langle \tilde{e} \rangle$, hence $P^\omega \langle \tilde{e} \rangle$ stays to be minimal until we reach $(\mathbf{v}.\underline{X}(\tilde{x}).A) \langle \tilde{e} \rangle$, as required. ■

7.3.2 Must characteristic formulae Next we prove:

Lemma 7.14 (Must characteristic formulae). *Let P and Q be closed and well-guarded, and be typable under the same typing and $\vdash_{\text{must}}^* P \blacktriangleright A$. Then $\models P \blacktriangleright A$ and, for each Q such that $\models Q \blacktriangleright A$, we have $Q \sqsubseteq_{\text{must}} P$.*

Note we restrict processes to well-guarded ones. As noted in Proposition 7.4, this does not lose generality semantically. We start from soundness.

Proposition 7.15 (Must characteristic formula: soundness). *$E \vdash_{\text{must}}^* P \blacktriangleright A$ implies $\models P \blacktriangleright A$.*

Proof. By rule induction. Rec is reasoned precisely as in the proof of the corresponding case for May characteristic formulae, in Proposition 7.15. The remaining cases are found in Appendix A.3. ■

For maximality we use the following property using \sqsubseteq_{must} .

Definition 7.16 (Must-closed properties). *We say a property is Must-closed if it has the maximal element w.r.t. \sqsubseteq_{must} . We say a parametrised property is Must-closed if its all well-typed instantiations are Must-closed. We say an interpretation environment ξ is Must-closed when ξ assigns a Must-closed parametrised property to each assertion variable.*

The following statement mentions well-guardedness (cf. §7.2), though the property holds with weakly well-guarded processes (in fact the proof only uses the latter).

Proposition 7.17 (Must characteristic formula: maximality). *If $E \vdash_{must}^* P \blacktriangleright A$ and P is well-guarded, then, for each Must-closed ξ , $\llbracket A \rrbracket \xi$ is also Must-closed, and, moreover, $P\xi\sigma$ is the maximal element of $\llbracket A \rrbracket \xi$ whenever σ assigns to each process variable say X the maximal element of $\xi(X)$.*

Proof. By rule induction. Below we focus on the recursion rule Rec, relegating other cases to Appendix A.4. Suppose by induction that, in

$$E \vdash_{must}^* P \blacktriangleright A \tag{7.3.4}$$

that, for each ξ and σ as noted in Proposition above, that

1. $P\sigma$ is a maximal element of $A\xi$ w.r.t. \sqsubseteq_{must} .
2. $A\xi$ is admissible.

We show the same conditions hold for the process and formula in conclusion. We recall the approximants to the maximal fixed point formula:

$$\begin{aligned} (\mathbf{v}^0 X(\tilde{x}).A)\langle \tilde{e} \rangle &\stackrel{\text{def}}{=} \top \\ &\dots \\ (\mathbf{v}^{n+1} X(\tilde{x}).A)\langle \tilde{e} \rangle &\stackrel{\text{def}}{=} A[(\mathbf{v}^n X(\tilde{x}).A)/X][\tilde{e}/\tilde{x}] \end{aligned}$$

We now show that, if a process, say Q , is such that

$$E \models Q \blacktriangleright (\mathbf{v}^i X(\tilde{x}).A)\langle \tilde{e} \rangle \tag{7.3.5}$$

for each ordinal i , then we have

$$Q \sqsubseteq_{must} (\mathbf{rec} X(\tilde{x}).P)\langle \tilde{e} \rangle \tag{7.3.6}$$

For the proof, suppose Q passes a must test O with a success flag at fresh c :

$$Q|O \Longrightarrow R \quad \text{implies} \quad R \xrightarrow{\bar{c}} \tag{7.3.7}$$

and suppose for each i we have

$$E \models Q \blacktriangleright (v^i X(\tilde{x}).A)\langle\tilde{e}\rangle \quad (7.3.8)$$

Now suppose the Q has the following multi-step visible transition corresponding to the first part of (7.3.7): (which as always we assume to be well-typed):

$$Q \xRightarrow{\hat{s}} Q' \quad (7.3.9)$$

By tracking this transition w.r.t. approximants by (7.3.8), and by (weak) well-guardedness and Corollary 7.9, we can infer, without going into infinite regression, that:

By satisfying the n -th approximant, the first n weak visible one-step transitions of Q are inside the transition tree given by $(v^n X(\tilde{x}).A)\langle\tilde{e}\rangle$.

This is because, by Corollary 7.9, an unfolding of the recursion always demands a visible action. Thus the recursion can simulate this transition, i.e.

$$(\mathbf{rec} X(\tilde{x}).P)\langle\tilde{e}\rangle \xRightarrow{\hat{s}} P' \quad (7.3.10)$$

where the visible weak actions from Q' is similarly constrained by those of P' , in that it is a sub-tree of that of P' . Hence the same property as the second part of (7.3.7) holds, as required. \blacksquare

We have now proved Lemma 7.14.

7.3.3 Mix characteristic formulae. This subsection proves:

Lemma 7.18 (Mix characteristic formulae). *Let P and Q be closed, be typable under the same typing and be well-guarded. Assume $\vdash_{mix}^* P \blacktriangleright A$. Then $\models Q \blacktriangleright A$ if and only if $P \approx Q$.*

Remark. The result holds under weak well-guardedness.

We first prove the soundness part (“if” direction) of lemma 7.18.

Notation 7.19. Given a well-typed $\Gamma \vdash P$, we write $\mathcal{L}_{\approx}(\Gamma \vdash P)$ for the formula A such that $\Gamma; E \models P \blacktriangleright A$ is derivable (such A is determined uniquely once given Γ and P , by Convention 7.1). We often leave the typing implicit, writing $\mathcal{L}_{\approx}(P)$.

Proposition 7.20 (mix characteristic formulae, soundness). *If $\Gamma; E \vdash_{mix}^* P \blacktriangleright A$ then $\Gamma; E \models P \blacktriangleright A$. In particular, when $\Gamma \vdash P$ is closed, we always have $P \in \llbracket \mathcal{L}_{\approx}(P) \rrbracket$.*

Proof. By rule induction. Rec is the same as before, while other cases are combination of the previous two modalities. ■

For completeness, we use a co-induction over processes and formulae, in contrast to the use of induction for May and Must characterisations. We use the following behavioural characterisations of extended mixed modalities (for the notion of extended mixed modalities, see Definition 6.2, page 47).

Lemma 7.21 (extended mix modalities and transitions). *Assume below Γ in each extended mixed modality is given from the typing of the process P as before, i.e. in $(\ell, \Gamma)A$, Γ is the typing of P minus the capability of ℓ . Under this assumption, we leave the typing of processes implicit. Further, for each clause, we assume that $E \vdash \sigma \prec \xi$, and that $\llbracket A\xi \rrbracket$ is not empty.*

1. *Suppose $E \models P \blacktriangleright (\bar{a}(k), \Gamma)A$. Then $P\xi\sigma \approx \bar{a}(k).Q$ such that $Q \in \llbracket A \rrbracket \xi$.*
2. *Suppose $E \models P \blacktriangleright (a(k), \Gamma)A$. Then $P\xi\sigma \approx a(k).Q$ such that $Q \in \llbracket A \rrbracket \xi$.*
3. *Suppose $E \models P \blacktriangleright \forall x.(kx, \Gamma)A$. Then $P\xi\sigma \approx k(x).Q$ such that $Q \in \llbracket A \rrbracket \xi$.*
4. *Suppose $E \models P \blacktriangleright \bigwedge_{i \in I} (k \triangleright l_i, \Gamma)A_i$ where the typing for k has precisely the $\{l_i\}$ branches. Then $P\sigma \approx k \triangleright [\&l_i.Q_i]$ such that $Q_i \in \llbracket A \rrbracket_i \xi$.*
5. *Suppose $E \models P \blacktriangleright (k \triangleleft l_j, \Gamma)A_j$. Then $P\sigma \approx k \triangleleft l_j.Q$ such that $Q \in \llbracket A \rrbracket \xi$.*
6. *Suppose $E \models P \blacktriangleright A_1 \circ A_2$. Then $P\sigma \approx Q_1 | Q_2$ such that $Q_i \in \llbracket A_i \rrbracket \xi$ ($i = 1, 2$).*
7. *Suppose $E \models P \blacktriangleright \nu x.A[x/u]$. Then $P\sigma \approx (\nu u)Q$ such that $Q \in \llbracket A[x/u] \rrbracket \xi$.*
8. *Suppose $\Gamma \vdash E \models P \blacktriangleright \text{noact}(\Gamma)$. Then $P\sigma \approx \mathbf{0}$.*

Proof. All are direct from the definition. We only outline the first and last clauses. For the first clause, fix σ and ξ and suppose $E \models P \blacktriangleright (\bar{a}(k), \Gamma)A$. Then by definition of the extended mix modality, we have

$$P\sigma \approx P' \xrightarrow{a(k)} Q, \quad Q \blacktriangleright A\xi. \quad (7.3.11)$$

Moreover $P\sigma$ is stable and has no other visible transitions. Thus taking Q above, and taking the obvious bisimulation, we get $P\sigma a(k).Q$, as required. For the last clause, if $\Gamma \vdash E \models P \blacktriangleright \text{noact}(\Gamma)$ then P has no transition, hence again by the obvious bisimulation we get $P \approx \mathbf{0}$. The remaining cases are similar. ■

We now prove the main lemma of this subsection. The statement and its proof below only rely on weakly well-guardedness.

Lemma 7.22. *Let P and R be closed, R be a well-guarded recursion and suppose $P \in \llbracket \mathcal{L}_{\approx}(R) \rrbracket$. Then $P \approx R$.*

In the proof below, the *one-time unfolding* of a fixed point formula is the result of unfolding a fixed point once. For example, given $(\nu X(\bar{x}).A)\langle \bar{e} \rangle$, its one-time unfolding is $A[\bar{e}/\bar{x}][(\nu X(\bar{x}).A)/X]$. Note that, by the standard unfolding axiom [31], a fixed-point formula and its one-time unfolding are logically equivalent.

Proof. Call R a *well-guarded component*, or WGC, if it is a sub-term of a well-guarded recursion, where we include the use of well-typed instantiation of variables in the construction of sub-terms. Note any well-guarded closed term is a combination of finite constructors (without recursion/replication) over WGCs. Now we set \mathcal{S} to be a relation over typed closed processes given by:

$$\mathcal{S} \stackrel{\text{def}}{=} \{(\Gamma \vdash P, \Gamma \vdash R) \mid \Gamma \vdash P \blacktriangleright \mathcal{L}_{\approx}(\Gamma \vdash R), R \text{ WGC and enough unfolded}\} \quad (7.3.12)$$

(Above, for the term “enough unfolded”, see Definition 7.7.) In the following we show \mathcal{S} is a weak bisimulation up to \approx [100], which proves the Proposition. Throughout the proof we often leave typing environments implicit for brevity. We first observe, writing $P \blacktriangleright A$ for P and A both closed and leaving the typing environment left implicit:

Claim. Let $P \blacktriangleright \mathcal{L}_{\approx}(\Gamma \vdash R)$ with P and R closed, typed under Γ and well-guarded. Then there exists Q satisfying the following three conditions:

1. $P \approx Q$.
2. For some context $\mathcal{C}[\cdot]$ and Q_1, \dots, Q_n and R_1, \dots, R_n (each Q_i and R_i closed), we have $Q \stackrel{\text{def}}{=} \mathcal{C}[Q_1] \dots [Q_n]$ and $R \stackrel{\text{def}}{=} \mathcal{C}[R_1] \dots [R_n]$ such that $Q_i \sigma \blacktriangleright \mathcal{L}_{\approx}(R_i \sigma)$ for $1 \leq i \leq n$ and for each well-typed instantiation σ .

Proof of Claim. Immediate from $P \blacktriangleright \mathcal{L}_{\approx}(\Gamma \vdash R)$ and by applying Lemma 7.21 sufficiently many times. \blacksquare

Let PSR . We start from a simulation of R by P . Recall we set R to be enough unfolded. We assume each mentioned transition is well-typed.

Case $R \xrightarrow{\ell} R'$. By Claim above, there exists Q such that $P \approx Q$ and Q has exactly the same syntactic structure as R except for recursions in R . Since R is enough unfolded, we can apply Proposition 7.8 to know this transition does not use any of these recursions. Hence we have:

$$Q \xrightarrow{\ell} Q', \quad Q' \blacktriangleright \mathcal{L}_{\approx}(R') \quad (7.3.13)$$

By $P \approx Q$, we know

$$P \xrightarrow{\hat{\ell}} P', \quad P' \approx Q' \blacktriangleright \mathcal{L}_{\approx}(R') \quad (7.3.14)$$

which means $P' * \mathcal{L}_{\approx}(R')$. Since R' is well-guarded and, through \approx , can always be made enough unfolded, we know $P'SR'$, as required.

Again set PSR . We show a simulation of P by R . The argument crucially relies

on Corollary 7.9, a bound in τ -actions for well-guarded processes.

Case $P \xrightarrow{\ell} P'$. Take Q as the preceding case, given by Claim above. By $P \approx Q$ we have, for some Q' :

$$Q \xrightarrow{\hat{\ell}} Q', \quad P' \approx Q' \blacktriangleright \mathcal{L}_{\approx}(R') \quad (7.3.15)$$

By the construction of Q by which Q has the same syntactic structure as R which is enough unfolded, R has precisely the same sequence of transitions as (7.3.15) using precisely the same inferences, since Corollary 7.9 says that such a sequence of transitions from an enough unfolded R can be inferred without using recursions, which are the only places Q and R differ (by the construction of R in Claim above). Thus we know:

$$R \xrightarrow{\hat{\ell}} R' \quad (7.3.16)$$

such that the inferences for (7.3.16) precisely follows those for (7.3.15). By Proposition 7.8 R' is well-guarded and by \equiv we can make R' well unfolded. Since each of R_i and Q_i in Claim above is not touched (except instantiation of variables to values if any), we obtain, by the construction of Q' and of characteristic formulae in the derivation rules of Figure 8, for R' above:

$$Q' \blacktriangleright \mathcal{L}_{\approx}(R') \quad (7.3.17)$$

By $P' \approx Q'$ this also means $P' \blacktriangleright \mathcal{L}_{\approx}(R')$, as required.

By the initial observation that any well-guarded process is combination of WGCs by finite combinators (without using recursion), we are done. \blacksquare

Combined with Proposition 7.20, this proves Lemma 7.18.

7.4 Soundness

Let us write $\Gamma; E \vdash_{\text{may}} P \blacktriangleright A$, $\Gamma; E \vdash_{\text{must}} P \blacktriangleright A$ and $\Gamma; E \vdash_{\text{mix}} P \blacktriangleright A$ for the respective provability in the three proof systems of the May/Must/Mixed modalities in § 5 and § 6. In this section we prove:

Theorem 7.23 (soundness). $\Gamma; E \vdash_{\text{may}} P \blacktriangleright A$ implies $\Gamma; E \models P \blacktriangleright A$, similarly for $\Gamma; E \vdash_{\text{must}} P \blacktriangleright A$ and $\Gamma; E \vdash_{\text{mix}} P \blacktriangleright A$.

Notation 7.24. In the following proof, whenever we write $E \models P \blacktriangleright A$, we always assume Γ is the typing of the sequent. When we interpret say $E \models P \blacktriangleright A$ as $P\sigma \in \llbracket A \rrbracket_{\xi}$, we often fix ξ and σ such that $E \vdash \sigma \prec \xi$ (cf. (4.3.1), page 32).

Proof. By rule induction of the proof rules of each proof system. The proofs for prefixes, inaction, parallel composition and hiding are shared with the soundness proofs for characteristic formulae, see Appendix A.2 and A.4. Here we only list the proofs of soundness for lf (which has a slightly different shape than the derivation rule for characteristic formulae), Var (for May and Must proof rules), Rec-ind , Rec-adm , Rec-mix and Conseq .

Case lf . By induction hypothesis we have $E \models_{P_1} \blacktriangleright e \supset A$ and $E \models_{P_2} \blacktriangleright \neg e \supset A$. In the environment where e is true, i.e. if $\llbracket e \supset A \rrbracket \xi = \top$, then

$$\text{if } e \text{ then } P \text{ else } Q \sigma \approx P \sigma \in \llbracket e \supset A \rrbracket \xi = \llbracket A \rrbracket \xi. \quad (7.4.1)$$

Symmetrically for the case when $\llbracket e \supset A \rrbracket \xi = \text{F}$, done.

Case Var . By definition having $X : (\tilde{x})A$ in the assumption means that if we instantiate \tilde{x} to \tilde{e} then we only assign to X the property coming from $A[\tilde{e}/\tilde{x}]$.

Case Rec-ind . In the rule, i and j range over values of some well-founded order. Since any well-founded order is isomorphic to an initial segment of ordinals, below we regard as such without loss of precision. Suppose by induction hypothesis we have:

$$E, X : (\tilde{x})(\forall j \preceq i.A(j)) \models P \blacktriangleright A(i) \quad (7.4.2)$$

By definition, (7.4.2) means that, for each i and for each ξ and σ such that $E \vdash \sigma \prec \xi$, we have, writing $A'(i)$ for $\forall j \preceq i.A(j)$

$$P \sigma \cdot X \mapsto (\tilde{x})Q \in \llbracket A(i) \rrbracket \xi \quad (Q \in \llbracket A'(i) \rrbracket \xi) \quad (7.4.3)$$

When $i = 0$, we can instantiate (7.4.3) as, noting $A'(0) = \top$:

$$(P[(\mathbf{rec} X(\tilde{x}).P)/X])\sigma \equiv P(\sigma \cup \{X \mapsto (\mathbf{rec} X(\tilde{x}).P)\sigma\}) \in \llbracket A(0) \rrbracket \xi \quad (7.4.4)$$

Observing

$$P[(\mathbf{rec} X(\tilde{x}).P)/X][\tilde{e}/\tilde{x}] \approx (\mathbf{rec} X(\tilde{x}).P)\langle \tilde{e} \rangle, \quad (7.4.5)$$

we obtain

$$(\mathbf{rec} X(\tilde{x}).P)\langle \tilde{e} \rangle \sigma \in \llbracket A(0) \rrbracket \xi \quad (7.4.6)$$

This bootstraps a hierarchy of satisfactions by repeatedly applying (7.4.3), obtaining $(\mathbf{rec} X(\tilde{x}).P)\langle \tilde{e} \rangle \sigma \in \llbracket A(i) \rrbracket \xi$ for each ordinal i , reaching:

$$(\mathbf{rec} X(\tilde{x}).P)\langle \tilde{e} \rangle \sigma \in \llbracket \forall i.A(i) \rrbracket \xi \quad (7.4.7)$$

that is $E \models (\mathbf{rec} X(\tilde{x}).P)\langle \tilde{e} \rangle \blacktriangleright \forall i.A(i)$, as required.

Case Rec-adm . Assume A is admissible. Then we have $\mathbf{0} \in \llbracket A \rrbracket \xi$, from which

(as we have done for Rec-ind above, but starting from the substitution of $\mathbf{0}$ for X), all finite unfoldings of $(\mathbf{rec} X(\tilde{x}).P)\langle \tilde{e} \rangle$ inhabits A under any ξ and σ such that $E \vdash \sigma \prec \xi$. Hence again by admissibility we know $(\mathbf{rec} X(\tilde{x}).P)\langle \tilde{e} \rangle$ inhabits A , as required.

Case Rec-mix. As already proved for the corresponding rule in the May derivation rule, in the proof of Proposition 7.11, 65.

Case Conseq. If $P\sigma \in \llbracket A \rrbracket \xi$ and $A \supset B$ then $P\sigma \in \llbracket A \rrbracket \xi \subset \llbracket B \rrbracket \xi$ hence done. ■

7.5 Completeness

The completeness results for May/Must proof systems are given with respect to the following subclasses of properties.

Definition 7.25 (May/Must-saturated properties). *A property p is May-saturated when it is May-closed and moreover $P \in p$ and $P \sqsubseteq_{\text{may}} Q$ implies $Q \in p$. A property p is Must-saturated when it is Must-closed and moreover $P \in p$ and $Q \sqsubseteq_{\text{must}} P$ implies $Q \in p$. Their counterparts for parametrised properties, and formulae, are obviously given.*

May/Must saturated properties are those properties which do not care behaviours except with respect to having at least some positive behaviours, or at most some negatively specified (lack of) behaviours. These properties naturally extend total and partial correctness in the standard sense to the richer interactional setting, which in particular include those for higher-order functions we explored in [54] (also see the end of the section for further discussions on these properties).

We now establish one of the basic results of the present paper. We use well-guardedness, although the completeness for Mix only needs weak well-guardedness and the completeness for May holds for arbitrary well-typed processes.

Theorem 7.26 (completeness). *Let $\Gamma \vdash P$ and let P be well-guarded and closed and A be closed.*

1. $\models P \blacktriangleright A$ and A May-saturated imply $\vdash_{\text{may}} P \blacktriangleright A$.
2. $\models P \blacktriangleright A$ and A Must-saturated imply $\vdash_{\text{must}} P \blacktriangleright A$.
3. $\models P \blacktriangleright A$ implies $\vdash_{\text{mix}} P \blacktriangleright A$.

Let P be closed. Observe that the least May-saturated property including P is the \sqsubseteq_{may} -upper set of P . Similarly the least Must-saturated property including P is the $\sqsubseteq_{\text{must}}$ -down set of P . Further the former can be given by saturating a \sqsubseteq_{may} -closed whose minimum element is P ; similarly the latter by saturating a $\sqsubseteq_{\text{must}}$ -closed set whose maximum element is P . Thus in order to prove Theorem 7.26, it suffices to prove the following.

Proposition 7.27 (completeness w.r.t. characteristic formulae). *Let P be closed and well-guarded below.*

1. $\vdash_{may}^* P \blacktriangleright A$ implies $\vdash_{may} P \blacktriangleright A$.
2. $\vdash_{must}^* P \blacktriangleright A$ implies $\vdash_{must} P \blacktriangleright A'$
3. $\vdash_{mix}^* P \blacktriangleright A$ implies $\vdash_{mix} P \blacktriangleright A$.

Note we do not have the assertion environment E since P is closed. The proof is by showing the same result for open processes through rule induction on the derivation systems given in (respectively) Figure 6, Figure 7 and Figure 8, given respectively in Page 58, Page 59, and Page 60, . showing provability w.r.t. the proof systems given in Figures 3, 4 and 5 (pages 36, 45, 47). For Must derivation, we use the derivation system which uses the same rules as in Figure 7 except Var and Rec are replaced by the following two rules:

$$\frac{}{X : (\tilde{x})A \vdash_{must}^* X^A \langle \tilde{\rho} \rangle \blacktriangleright A[\tilde{\rho}/\tilde{x}]} \quad \frac{E, X : (\tilde{x})A \vdash_{must}^* P \blacktriangleright A}{E \vdash_{must}^* (\mathbf{rec} X(\tilde{x}).P) \langle \tilde{\rho} \rangle \blacktriangleright A[\tilde{\rho}/\tilde{x}]}$$

Var-Close, Rec-close

Above A is a ν -recursion (a maximal fixed point) of the form $(\nu X(\tilde{x}).A_0) \langle \tilde{x} \rangle$ which is later to be bound to X , in an enclosing target closed process. That is, we assume that we first annotate the occurrences of each process variable in a closed process, say P , by going through the derivation of Figure 7 for P (the annotation is done after unfolding a ν -recursion for each occurring process variable). For example, if a process is

$$(\mathbf{rec} X(k).\bar{k}\langle 1 \rangle.X \langle k \rangle) \langle k \rangle \tag{7.5.1}$$

Then we annotate X of “ $X \langle k \rangle$ ” in the process above by, assuming it is typed under $k : \tau, \Gamma$ for an appropriate τ and Γ :

$$(\nu X(k).\llbracket \bar{k}\langle 1 \rangle, \Gamma \rrbracket X \langle k \rangle) \langle k \rangle \tag{7.5.2}$$

Note the resulting derivation coincides for closed typed processes. We denote the derivability in this refined derivation as $E \vdash_{must}^{**} P \blacktriangleright A$. Note A does not contain a free assertion variable, by construction, though it may contain a value variable. By well-guardedness, such A is constructed from extended Must modalities, conjunction, disjoint composition, i.e. parallel composition without a potential redex (under well-typed substitutions), hiding and ν -recursion (for the formal construction see Definition A.1 in Appendix A.5). We call such formulae, *well-guarded Must-modality formulae*. We note:

Lemma 7.28. *Let A be well-guarded Must-modality formulae. Then A is admissible.*

Proof. By showing that, for the establishment of satisfaction of such a formula by a process which has only visible transitions (as in a well-guarded recursive process), it suffices to check its sub-trees of its synchronisation tree at all finite depths satisfy the formula. We call this property, *finite visible property*. This property is proved through inductive construction of such formulae, using the standard approximants for the v-recursion. See Appendix A.5 for the proof. ■

We believe that the result here holds for its “weak” version (defined in correspondence with weak well-guardedness in processes), through the justification of the encoding of weak well-guardedness to well-guardedness discussed in §8.1, though details are to be checked.

We can now prove Proposition 7.27. Since, in each pair of the derivation system and the proof system, the rules for prefixes, composition and hiding are identical, we only treat the following derivation rules: If (common to three derivation systems), Rec for May derivation rule, Rec for Must derivation rule, and Rec for Mix derivation rule.

If. This rule is common to May/Must/Mix derivation systems. The reasoning is also common to all these systems, using the common proof rules Conseq and If (the latter being the proof rules in Fig. 3). By induction hypothesis we have $E \vdash P_1 \blacktriangleright A_1$ and $E \vdash P_2 \blacktriangleright A_2$. We now observe:

$$A_1 \supset e \supset A_1 \quad (7.5.3)$$

$$\supset e \supset (e \wedge A_1) \quad (7.5.4)$$

$$\supset e \supset (e \wedge A_1 \vee \neq e \wedge A_2) \quad (7.5.5)$$

Symmetrically we have the deduction:

$$A_2 \supset (\neg e \supset (e \wedge A_1 \vee \neq e \wedge A_2)). \quad (7.5.6)$$

Hence by applying the proof rules Conseq and If, we obtain the conclusion of the derivation rule If.

Var. Immediate.

Rec (**May**). We consider the derivation rule Rec for May characteristic formula against the proof rule Rec-ind. We first set:

$$A(n) \stackrel{\text{def}}{=} (\mathbf{v}^n \underline{X}(\tilde{x}).A)\langle \tilde{e} \rangle \quad (7.5.7)$$

We also let $A'(n) \stackrel{\text{def}}{=} \forall j \prec n. A(j)$. By induction hypothesis we have $E, X : (\tilde{x}) \underline{X} \vdash P \blacktriangleright A$. By substitution we obtain:

$$E, X : (\tilde{x}) A'(n) \vdash P \blacktriangleright A[(\tilde{x}) A'(n) / \underline{X}] \quad (7.5.8)$$

We now show, by transfinite induction on n , that

$$A[(\tilde{x})A'(n)/\underline{X}] \supset A(n). \quad (7.5.9)$$

This is immediate when $n = 0$. For $n = m + 1$ with m being an ordinary ordinal, we get:

$$\begin{aligned} A[(\tilde{x})A'(m+1)/\underline{X}] &\supset A[(\tilde{x})A(m)/\underline{X}] \\ &\stackrel{\text{def}}{=} A(m+1) \quad \stackrel{\text{def}}{=} A(n) \end{aligned}$$

If $n = \lambda$ is a limit ordinal, since in this case $A'(\lambda) = A(\lambda)$, we reason:

$$\begin{aligned} A[(\tilde{x})A'(\lambda)/\underline{X}] &\stackrel{\text{def}}{=} A[(\tilde{x})A(\lambda)/\underline{X}] \\ &\supset A(\lambda) \end{aligned}$$

as required.

Rec (Must). Working with the extended derivation \vdash_{must}^{**} as discussed above, suppose

$$E, X : (\tilde{x})A \vdash_{must}^{**} P \blacktriangleright B \quad (7.5.10)$$

occurs as the premise of Rec. By construction the premise B in fact coincides with A . Further by construction A is a well-guarded Must modal formula (for the precise syntatic construction of this formula, see Appendix A.5) hence, by Lemma 7.28 we know it is admissible. Hence we can apply Rec-adm to each the conclusion of Rec.

Rec (Mix). This is identical with the proof rule Rec for Mix, Figure 8.

We have now proved Proposition 7.27, hence Theorem 7.26. \blacksquare

Remark 7.29 (Extensions of Completeness Results). These results can extend to many other typed π -calculi without general sums (as far as they obey the algebras in [51] such as partial commutativity and associativity in parallel composition, which, as far as we know, are satisfied by all known typed π -calculi). It should be noted that the May characteristic formulae are not May-saturated, similarly for the Must characteristic formulae. This suggests there are broader classes of properties the May and Must proof systems in fact cater for, extending the completeness results.

8 Reasoning Examples

8.1 Reasoning about Synchronised Stateful Interaction

We now turn to stateful agents, extending the simple ATM discussed in § 2 and 3 to the three-party interactions among User, ATM and Bank. Our purpose in this example is two-fold: first, it shows a typical case, in specification and reasoning, how the composition of stateful communicating systems exhibit both (inevitable global) non-determinism and (local) determinism. Second, its natural specification demands us to capture *transfer of state* induced by (repeated) synchronised actions among multiple parties, which needs be captured as a single modal behaviour obeying a certain invariance. In the present case, User interacts with ATM and ATM interacts with Bank, so the behaviour of ATM for User is in fact a composite of ATM and Bank, which requires certain consistency.

The scenario extends the previous one (given in § 2.1) by adding withdraw among the options ATM offers. Further ATM now asks Bank each time it receives a request, and forwards its answer to User. Below we list the π -calculus term for this behaviour, which we henceforth denote as *ATM*.

$$\begin{aligned}
 & a(k).\bar{b}(k').\mathbf{rec}Y.(\\
 & \quad k \triangleright [\text{balance} : k' \triangleleft \text{balance}.k'(z).\bar{k}(z).Y, \\
 & \quad \quad \text{withdraw} : k(n).k' \triangleleft \text{withdraw}.\bar{k}'(n).k' \triangleright \begin{cases} \text{ok} : k \triangleleft \text{ok}.Y \\ \text{no} : k \triangleleft \text{no}.Y \end{cases}, \\
 & \quad \text{quit} : k' \triangleleft \text{quit})
 \end{aligned}$$

Thus ATM itself is now stateless: it does not retain any state on its own. The stateful part is now performed by Bank, whose specification is presented later.

By interacting with Bank, the state change in Bank is reflected onto the actions of ATM, so that ATM will *behave to User* as if it were stateful. To this behaviour User may as well demand the following consistency criteria:

If User first looks at the balance then withdraws money one or more times, the balance after each withdrawal should be the immediately preceding one minus the withdrawn amount; and the withdrawal succeeds if it is within the current balance.

Even if the bank account itself is shared, User may expect this consistency criteria to be obeyed within a single session. Thus we consider the following specification, $\text{ATMSpec}(a, x)$, for ATM,

$$\begin{aligned}
 \text{ATMSpec}(a, x) &= (a(k))(\nu Z(z).A)\langle x \rangle \\
 \text{with } A &= (k \triangleright \text{withdraw}) \forall n.(kn) \wedge \begin{pmatrix} z \geq n \supset (k \triangleleft \text{ok})Z\langle z-n \rangle \\ z < n \supset (k \triangleleft \text{no})Z\langle z \rangle \end{pmatrix}
 \end{aligned}$$

where x denotes its initial balance. We shall later slightly refine this specification. For now our purpose is to show:

$$ATM \models \text{BankSpec}(b, 300) \triangleright \text{ATMSpec}(a, 300)$$

where $\text{BankSpec}(b, x)$ is a specification for Bank which is given as (again focusing on withdrawal):

$$\begin{aligned} \text{BankSpec}(b, x) &= \langle b(k') \rangle (\nu Z(z). B) \langle x \rangle \\ \text{with } B &= \langle k' \triangleright \text{withdraw} \rangle \forall n. \langle k'n \rangle \wedge \left(\begin{array}{l} z \geq n \supset \langle k' \triangleleft \text{ok} \rangle Z \langle z - n \rangle \\ z < n \supset \langle k' \triangleleft \text{no} \rangle Z \langle z \rangle \end{array} \right) \end{aligned}$$

Note both $\text{BankSpec}(b, x)$ and $\text{ATMSpec}(a, x)$ have state. To derive the target assertion, we start from the assertion directly derivable by applying \vdash_{mix}^* to ATM and weakening a little, which we call $\text{ATMSpec}_0(a, b)$, given as:

$$\begin{aligned} \text{ATMSpec}_0(a, b) &= \langle a(k) \rangle \langle \bar{b}(k') \rangle \nu Y. A_0 \\ \text{with } A_0 &= \langle k \triangleright \text{withdraw} \rangle \langle k' \triangleleft \text{withdraw} \rangle \forall n. \langle kn \rangle \langle \bar{k}'n \rangle \wedge \left(\begin{array}{l} \langle \bar{k}' \triangleright \text{ok} \rangle \langle k \triangleleft \text{ok} \rangle . Y \\ \langle \bar{k}' \triangleright \text{no} \rangle \langle k \triangleleft \text{no} \rangle . Y \end{array} \right) \end{aligned}$$

It thus suffices to show:

$$\text{ATMSpec}_0(a, b) \circ \text{BankSpec}(b, 300) \supset \text{ATMSpec}(a, 300)$$

Note $\text{ATMSpec}_0(a, b)$ is stateless, so that we need to realise the stateful specification $\text{ATMSpec}(a, 300)$ by its composition with $\text{BankSpec}(b, 300)$. We use the following axioms (all are from Section B). We observe that the second axiom cannot take off $\langle \rangle$. This point is further discussed at the end of this subsection.

Proposition 8.1. *Below we assume formulae are appropriately typed.*

1. Assume ℓ is linear. Then we have $(\langle \ell \rangle A) \circ B \equiv \langle \ell \rangle (A \circ B)$ (assuming both sides have the same type) and $\langle \ell \rangle A \circ \langle \bar{\ell} \rangle B \equiv \nu \text{bn}(\ell). (\langle \langle \rangle \rangle A \circ \langle \langle \rangle \rangle B)$.
2. $\langle \ell \rangle A \circ \langle \bar{\ell} \rangle B \supset (\langle \langle \ell \rangle \rangle (A \circ \langle \bar{\ell} \rangle B) \wedge \langle \langle \rangle \rangle (A \circ B) \wedge \langle \langle \bar{\ell} \rangle \rangle (\langle \ell \rangle A \circ B))$
3. (state merging axiom) Assume $A \circ B \supset C[X \langle \tilde{e} \rangle \circ Y \langle \tilde{g} \rangle]_i$ is valid. Then

$$(\nu X(\tilde{x}). A) \langle \tilde{e} \rangle \circ (\nu Y(\tilde{y}). B) \langle \tilde{g} \rangle \supset (\nu Z(\tilde{x}\tilde{y}). C[Z \langle \tilde{e}\tilde{g} \rangle]_i) \langle \tilde{e}\tilde{g} \rangle$$

where $C[X \langle \tilde{e} \rangle_i \circ Y \langle \tilde{g} \rangle_i]_{i \in I}$ denotes a formula with multiple holes indexed by I , assuming all occurrences of X , Y and Z are thus exhausted.

Proof. We outline the proof of the last axiom (state-merging axiom). Others are easier. For simplicity we assume I is a singleton (i.e. there is only one occurrence of the stipulated variable) so that we can write our assumption:

$$A \circ B \supset C[X \langle \tilde{e} \rangle \circ Y \langle \tilde{g} \rangle]_i \tag{8.1.1}$$

We use the induction using the standard unfoldings of maximum fixed points, by which the left-hand side is the conjunction of all unfoldings:

$$(\mathbf{v}^k X(\tilde{x}).A)\langle \tilde{e} \rangle \circ (\mathbf{v}^k Y(\tilde{y}).B)\langle \tilde{g} \rangle \quad (8.1.2)$$

and the right-hand side is the conjunction of all unfoldings:

$$(\mathbf{v}^k Z(\tilde{x}\tilde{y}).C[Z\langle \tilde{e}\tilde{g} \rangle]_i)\langle \tilde{e}\tilde{g} \rangle \quad (8.1.3)$$

Write α^k , β^k and θ^k for these unfolding formulae. We also write (under the standard bound name condition) σ for the substitution $[\tilde{e}\tilde{g}/\tilde{x}\tilde{y}]$.

The base case is trivial. For the successor case, the right-hand side is:

$$A\tilde{x}[\alpha^k/X]\sigma \circ B[\beta^k/Y]\sigma \quad (8.1.4)$$

By (8.1.1), and writing σ_X , σ_Y and σ_Z for the substitutions replacing (respectively) X , Y and Z with α^k , β^k and γ^k , (8.1.4) entails:

$$C\sigma_X\sigma_Y\sigma \quad (8.1.5)$$

By induction and because of the positive occurrences this also means:

$$C\sigma_Z\sigma \quad (8.1.6)$$

as required. The limit case is immediate. Thus at each step the entailment holds, hence their conjunctions are related in the same way, as required. ■

We now calculate:

$$A_0 \circ B \supset A$$

Note the interactions involve only strong linear actions so that all dual actions at k' silently compensate by Axiom (3) in Proposition 8.1:

$$\begin{aligned} & \{ (k' \triangleleft \text{withdraw}), (\bar{k}' n), (\bar{k}' \triangleright \text{ok}), (\bar{k}' \triangleright \text{no}) \} \text{ in } A_0 \text{ and} \\ & \{ (k' \triangleright \text{withdraw}), (k' n), (k' \triangleleft \text{ok}), (k' \triangleleft \text{no}) \} \text{ in } B \end{aligned}$$

are cancelled, preserving causality. This and Axiom (6) in Proposition 8.1 give us:

$$\mathbf{v}Y.A_0 \circ (\mathbf{v}Z(z).B)\langle x \rangle \supset (\mathbf{v}Z(z).A)\langle x \rangle$$

All involved actions have strong modality: once a session has started, we do not want it to be interrupted nor its state getting inconsistent. We have successfully transferred the bank's state to ATM, assuring the required consistency.

We now finish, combining all the specifications:

$$\begin{aligned}
& \langle \langle b(k') \rangle \rangle . (\nu Z(z).B) \langle 300 \rangle \circ \langle \langle a(k) \rangle \rangle \langle \langle \bar{b}(k') \rangle \rangle (\nu Y.A_0) \\
& \supset \langle \langle a(k) \rangle \rangle (\langle \langle b(k') \rangle \rangle (\nu Z(z).B) \langle 300 \rangle \circ \langle \langle \bar{b}(k') \rangle \rangle (\nu Y.A_0)) \\
& \supset \langle \langle a(k) \rangle \rangle \langle \langle \rangle \rangle (\nu Z(z).A) \langle 300 \rangle
\end{aligned}$$

But the final conclusion above has a subtle difference from our original specification, $\text{ATMSpec}(a, 300)$: now $\langle \langle \rangle \rangle$ is inserted after the first action, derived by Axiom (5) in Proposition 8.1. Indeed, $\text{ATMSpec}(a, 300)$ is impossible to realise as far as b is a shared channel, since there can be interference at b . Thus we insert $\langle \langle \rangle \rangle$, saying “it can happen but it may not”. In this way, the proposed framework can efficiently infer fine-grained mixture of determinism and non-determinism involving recursive state transfer.

In the example above, we have used the basic fact that two compensating “strong” actions represented by $\langle \langle \ell \rangle \rangle A$ do not generally induce another strong action. In fact, we have the following result.

Fact 8.2. Suppose $B \circ C \supset \langle \langle \ell \rangle \rangle A$. Then there is a counter-example to the following assertion:

$$\langle \langle \ell_0 \rangle \rangle B \circ \langle \langle \bar{\ell}_0 \rangle \rangle C \supset \langle \langle \ell \rangle \rangle A.$$

Proof. Take $P = \bar{b}$, $Q = b.\bar{a}$ where P satisfies $\langle \langle \bar{b} \rangle \rangle \text{noact}(a, b, c)$ and Q satisfies $\langle \langle b \rangle \rangle \langle \langle \bar{a} \rangle \rangle \text{noact}(a, b)$, and that $\text{noact}(a, b, c) \circ \langle \langle \bar{a} \rangle \rangle \text{noact}(a, b)$ does entail $\langle \langle \bar{a} \rangle \rangle \top$ but $P|Q$ does not satisfy $\langle \langle \bar{a} \rangle \rangle \top$ since the involved τ -action is at a shared channel. ■

This is why the use of *typed* actions plays the essential role in the present system: when a channel is linear or replicated, the induced τ -action is always stateless, mediating the reductum and residual with \approx .

8.2 Capturing Imperative Variables: Sequential and Concurrent

The way we have captured imperative computation in the previous subsection can be adapted to the reasoning principles for standard sequential computation as well as shared variable concurrency. In particular, such reasoning principles as presented by Floyd and Hoare (for sequential computation) and Cliff Jones (for shared variable concurrency), as well as those which involve locks.

In the present paper we shall not go into detailed specification, but just discuss the core idea. First, we consider the following standard specification of an imperative variable:

$$\text{Var} \langle xv \rangle = \left(\text{rec } X \langle v \rangle . x(k). k \triangleright \left[\begin{array}{l} \text{read} : \bar{k} \langle v \rangle . X \langle xv \rangle, \\ \text{write} : k(w) . X \langle xw \rangle \end{array} \right] \right) \langle v \rangle$$

We can then infer the following specification for this “variable agent”.

$$\text{VarSPec}(xv) = vX\langle v \rangle . (x(k)) \left(\begin{array}{c} (k \triangleright \text{read}) (\bar{k}v) X\langle v \rangle \\ \wedge \\ (k \triangleright \text{write}) \forall w. (kw) X\langle w \rangle \end{array} \right)$$

Now the agent which interacts with the variable agent may be encoded as:

$$[[x := 2]]_u \stackrel{\text{def}}{=} \bar{x}(k) . k \triangleleft \text{write} . \bar{k}\langle 2 \rangle . \bar{u} :$$

Under the sequential typing [17], we can then infer:

$$[[x := 2]]_u \models \exists pre. \text{VarSPec}(x, pre) \triangleright [[\bar{u}]] \text{VarSPec}(x, 2)$$

Exactly the same framework can be used for encoding the rely-guarantee method by Jones [65]. It would also be interesting to extend this method to object-oriented programming incorporating method invocations (along the line of [64]), using the encoding of objects into processes as found in [115].

When an imperative variable is used in combination with a lock, as in many concurrent programs with shared memory, a straightforward encoding is obtained through combining a lock with a variable, as follows.

$$LVar\langle xv \rangle = \left(\text{rec } X\langle v \rangle . x(k) . \text{rec } Y\langle v \rangle . k \& \left[\begin{array}{l} \text{read} . \bar{k}v . Y\langle v \rangle, \\ \text{write} . k(w) . Y\langle w \rangle, \\ \text{unlock} . X\langle xv \rangle \end{array} \right] \right) \langle v \rangle$$

The specification for this agent can be given as:

$$vX\langle v \rangle . (x(k)) \forall Y\langle y \rangle . \left(\begin{array}{c} (k \triangleright \text{read}) (\bar{k}y) Y\langle y \rangle \wedge \\ (k \triangleright \text{write}) \forall w. (kw) Y\langle w \rangle \wedge \\ (k \triangleright \text{unlock}) X\langle y \rangle \end{array} \right) \langle v \rangle$$

which is close to the bank account we discussed in §8.1, suggesting how we can reason about locked variables. A refined way to use a locked variable is to use delegation of a linear handle to another program, which limits certain idioms (such as shared read lock) but which allows a tractable reasoning on state. More concrete reasoning examples will be discussed in future expositions.

8.3 Extending Logic for Server-Client Types

In the next two examples, we use a type discipline in which a name can be used by one agent for input repeatedly often, while the remaining agents will use it for outputs, just as a server and clients interact. Such a discipline is presented

and studied by Sangiorgi as *uniform receptiveness* [98]. Its introduction to the present system is simple.

First we introduce the following two linear types:

$$\alpha ::= \dots \mid (\alpha)^! \mid (\alpha)^?$$

The first one is *server type*, while the second one is *client type*. We set $\overline{(\alpha)^!} = (\overline{\alpha})^?$. The algebra over these types is given as follows:

$$(\alpha)^! \odot (\overline{\alpha})^? = (\alpha)^!$$

which extends to the environments as before. Then we stipulate the following type rules:

$$\text{Ser} \frac{\Gamma, k: \tau \vdash P}{\Gamma, a: (\tau)^! \vdash !a(k).P} \quad \text{CReq} \frac{\Gamma \vdash a: (\tau)^? \quad \Gamma, k: \tau \vdash P}{\Gamma \vdash \bar{a}(k).P}$$

Note we use the replication for typing. The same subject reduction and other properties hold. The semantics is given precisely as before.

For proof rules, we have the following rules for May:

$$\text{Ser} \frac{E \vdash P \blacktriangleright A}{E \vdash !a(k).P \blacktriangleright \langle\langle a(k) \rangle\rangle A} \quad \text{CReq} \frac{E \vdash P \blacktriangleright A}{E \vdash \bar{a}(k).P \blacktriangleright \langle\langle \bar{a}(k) \rangle\rangle A}$$

For Must modality, we get:

$$\text{Ser} \frac{E \vdash P \blacktriangleright A}{E \vdash !a(k).P \blacktriangleright \llbracket a(k), \Gamma \rrbracket A} \quad \text{CReq} \frac{E \vdash P \blacktriangleright A}{E \vdash \bar{a}(k).P \blacktriangleright \llbracket \bar{a}(k), \Gamma \rrbracket A}$$

under the same typing convention as before. Finally for mixed modality we set:

$$\text{Ser} \frac{E \vdash P \blacktriangleright A}{E \vdash !a(k).P \blacktriangleright \langle\langle a(k), \Gamma \rangle\rangle A} \quad \text{CReq} \frac{E \vdash P \blacktriangleright A}{E \vdash \bar{a}(k).P \blacktriangleright \langle\langle \bar{a}(k), \Gamma \rangle\rangle A}$$

In these rules, we do not use fixed point: while we can do so, interactions following the uniform receptive discipline, where one stateless server will receive many clients' request, leads to the following axioms (shown in the case of mixed modality):

$$\langle\langle a(k) \rangle\rangle A \circ \langle\langle \bar{a}(k) \rangle\rangle B \equiv \langle\langle a(k) \rangle\rangle A \circ \nu k. (A \circ B) \quad (8.3.1)$$

$$\langle\langle a(k) \rangle\rangle A \circ \langle\langle \ell \rangle\rangle B \equiv \langle\langle a(k) \rangle\rangle A \circ \langle\langle \ell \rangle\rangle (\nu a. (\langle\langle a(k) \rangle\rangle A \circ B)) \quad (8.3.2)$$

which are instances of the equivalence between $\langle\langle a(k) \rangle\rangle A$ and $\nu X. \langle\langle a(k) \rangle\rangle (A \circ X)$ for fresh X , and which dispense with the explicit specification of fixed points.

The same completeness and soundness results, through the derivation of characteristic formulae, are satisfied for the augmented type discipline.

8.4 Data Type Encoding: from Untyped to Typed

First we present a small example, treating a typed version of Milner's encoding of natural numbers [82]. Essentially the same agent is treated by Dam [31] in the untyped setting. In its conciseness the encoding captures many of the essential elements of the π -calculus. Our purpose is to pinpoint the effect of having types in a modal specification of processes. The encoding is defined inductively as follows.

Definition 8.3 (natural number encoding). *For each natural number n , we define $\llbracket n \rrbracket_u$ by the following induction:*

$$\begin{aligned}\llbracket 0 \rrbracket_u &= !u(k).k \triangleleft \text{zero} \\ \llbracket n + 1 \rrbracket_u &= (\nu p)(\text{Succ}(up) \mid \llbracket n \rrbracket_p)\end{aligned}$$

where $\text{Succ}(u, p) = !u(k).k \triangleleft \text{succ}.\bar{k}(p)$.

For its typing we observe:

Proposition 8.4. *Let $\text{nattype} = \mathbf{rect}.\left(\oplus\{\text{zero} : \text{end}, \text{succ} : \uparrow t; \text{end}\}\right)!$. Then we can check $u : \text{nattype} \vdash \llbracket n \rrbracket_u$ for each n .*

Thus all natural number encodings are given the same types. Note nattype already gives a rough specification, focusing on the skeletal interaction structures including directions of communication. Upon this backbone we can specify their behavioural properties as a logical assertion, as we shall do so.

Example 8.5 (addition). As an example of operators on natural numbers, for example we can define addition as follows.

$$\begin{aligned}\text{Add}(k, b, c) &= \bar{b}(k).k \triangleright [\\ &\quad \text{zero} : \bar{k}(n), \\ &\quad \text{succ} : k(x).(\nu k')(\text{Add}(k', p, c) \mid k'(x).(\nu y)\bar{k}(y).\text{Succ}(yx))]\end{aligned}$$

Then we can check

$$(\nu m, n)(!a(k).\text{Add}(k, b, c) \mid \llbracket n \rrbracket_b \mid \llbracket m \rrbracket_c) \approx \llbracket m + n \rrbracket_a$$

We can further define multiplication, subtraction and other operations in the same way, which we omit.

We now consider specifications for natural numbers. As one of the natural specifications, we consider the following assertion.

Definition 8.6 (specification for natural numbers). Under $u : \text{nattype}$:

$$\text{NatSpec}(u) = \nu X(x). (\langle x(k) \rangle ((k \triangleleft \text{zero}) \text{noact}(k) \vee (k \triangleleft \text{succ}) \exists p. (\langle \bar{k}p \rangle \text{noact}(k) \circ X(p)))) \langle u \rangle$$

where $\text{noact}(k)$ is given as in § 6.2 (by which, indeed, it is defined to be \top).

The assertion $\text{NatSpec}(u)$ is conformant to nattype (as it should be): but it now tries to capture the whole of the behaviour, with the help of strong modality and the least fixed point, on the basis of the use of strong typing.

We first show soundness, i.e. that all encodings satisfy this specification. For this purpose it suffices to show that the judgement $\vdash \llbracket n \rrbracket_u \blacktriangleright \text{NatSpec}(u)$ holds under $u : \text{nattype}$, for each n . We use the following logical laws.

Lemma 8.7. Under an appropriate typing the following assertions are valid.

- (1) $(A \wedge B) \supset A$.
- (2) $(\nu X(\bar{x}).A) \langle \bar{e} \rangle \equiv A[\nu X(\bar{x}).A/X][\bar{e}/\bar{x}]$.
- (3) $(A \wedge B \equiv F) \supset ((\hat{\ell})(A \vee B) \equiv (\hat{\ell})A \vee (\hat{\ell})B)$.
- (4) $A \circ (B \vee C) \equiv (A \circ B) \vee (A \circ C)$ assuming $B \wedge C \equiv F$ is valid.
- (5) $A \circ \exists i. B(i) \equiv \exists i. (A \circ B(i))$ where we assume $\forall i \neq j. (B(i) \wedge B(j)) \equiv F$ is valid and i does not occur in A .

Proof. (1) is a law of predicate calculus. The rest are instances of the axioms in Section B. ■

Note that by Lemma 8.7 (2) we have:

$$\text{NatSpec}(u) \equiv \langle u(k) \rangle ((k \triangleleft \text{zero}) \text{noact}(k) \vee (k \triangleleft \text{succ}) \exists p. (\langle \bar{k}p \rangle \text{noact}(k) \circ \text{NatSpec}(p)))$$

We use this unfolding (the r.h.s. of the equivalence above) for our proof of:

Proposition 8.8. $u : \text{nattype} \models \llbracket n \rrbracket_u \blacktriangleright \text{NatSpec}(u)$.

Proof. By induction on n we show

$$u : \text{nattype} \vdash \llbracket n \rrbracket_u \blacktriangleright \text{NatSpec}(u).$$

in the proof system for mixed modalities in Section 6.4.

Case $n = 0$. Note that $\text{noact}(k : \text{end}) = \top$.

$$\begin{array}{c} \frac{k : \text{end} \vdash 0 \blacktriangleright \top}{k : k \triangleleft \text{zero}; \text{end} \vdash k \triangleleft \text{zero} \blacktriangleright (k \triangleleft \text{zero}) \top} \text{(Inact)} \\ \frac{k : k \triangleleft \text{zero}; \text{end} \vdash k \triangleleft \text{zero} \blacktriangleright (k \triangleleft \text{zero}) \top}{u : \text{nattype} \vdash \llbracket 0 \rrbracket_u \blacktriangleright \langle u(k) \rangle (k \triangleleft \text{zero}) \top} \text{(Sel)} \\ \frac{u : \text{nattype} \vdash \llbracket 0 \rrbracket_u \blacktriangleright \langle u(k) \rangle (k \triangleleft \text{zero}) \top}{u : \text{nattype} \vdash \llbracket 0 \rrbracket_u \blacktriangleright \text{NatSpec}(u)} \text{(Ser)} \\ \frac{}{u : \text{nattype} \vdash \llbracket 0 \rrbracket_u \blacktriangleright \text{NatSpec}(u)} \text{(Conseq)} \end{array}$$

Case $n > 0$.

$$\frac{\frac{u : \text{natype}, p : \text{natype} \vdash \text{Succ}(up) \blacktriangleright \langle u(k) \rangle \langle k \triangleleft \text{succ} \rangle \langle \bar{k}p \rangle \top \quad (\text{Par})}{u : \text{natype}, p : \text{natype} \vdash \text{Succ}(up) \mid \llbracket n \rrbracket_p \blacktriangleright \langle u(k) \rangle \langle k \triangleleft \text{succ} \rangle \langle \bar{k}p \rangle \top \circ \text{NatSpec}(p) \quad (\text{Res})}}{u : \text{natype} \vdash (\forall p)(\text{Succ}(up) \mid \llbracket n \rrbracket_p) \blacktriangleright \forall p. (\langle u(k) \rangle \langle k \triangleleft \text{succ} \rangle \langle \bar{k}p \rangle \top \circ \text{NatSpec}(p)) = C}$$

Since p is a server type, we can change \forall to \exists in C . Also by $A \supset A \vee B$, we know C entails:

$$\langle u(k) \rangle \langle k \triangleleft \text{zero} \rangle \top \vee \langle u(k) \rangle \langle k \triangleleft \text{succ} \rangle \exists p. (\langle \bar{k}p \rangle \top \circ \text{NatSpec}(p)) \quad (8.4.1)$$

Thus we conclude, for each n :

$$\vdash \llbracket n \rrbracket_u \blacktriangleright \text{NatSpec}(u)$$

which, by Theorem 7.23 implies the required statement. \blacksquare

Thus all encodings are inhabitants of $\text{NatSpec}^n(u)$. We now show the converse: all processes which satisfy $\text{NatSpec}^n(u)$ are essentially (up to \approx) the encodings of natural numbers. Thus $\text{NatSpec}(u)$ indeed captures the whole of the behaviour of natural number encodings precisely. To prove this result, use the standard approximants of the least fixed points.

Definition 8.9. Let $\text{NatSpec}^n(u)$ be given by:

$$\begin{aligned} \text{NatSpec}^0(u) &= \text{F} \\ \text{NatSpec}^{n+1}(u) &= \langle u(k) \rangle (\langle k \triangleleft \text{zero} \rangle \text{noact}(k) \vee \\ &\quad \langle k \triangleleft \text{succ} \rangle \exists p. (\langle \bar{k}p \rangle \text{noact}(k) \circ \text{NatSpec}^n(p))) \\ \text{NatSpec}^\omega(u) &= \exists i \in \text{Nat}. \text{NatSpec}^i(u) \end{aligned}$$

Below we write $\llbracket A \rrbracket$ is the interpretation of A given in Definition 4.7 where we omit ξ because A is closed. The typing is always $u : \text{natype}$.

Lemma 8.10. $\llbracket \text{NatSpec}(u) \rrbracket = \llbracket \text{NatSpec}^\omega(u) \rrbracket$, i.e. $\text{NatSpec}(u)$ and $\text{NatSpec}^\omega(u)$ are logically equivalent

Proof. We can syntactically reason, using Lemma 8.7 (5),

$$\text{NatSpec}^\omega(u) \equiv C[\text{NatSpec}^\omega(u)/X\langle x \rangle][u/x]$$

where C is the body of $\text{NatSpec}(u)$. This shows $\llbracket \text{NatSpec}^\omega(u) \rrbracket$ is indeed a fixed point of the underlying function on parametrised properties. Since the chain of approximants is increasing and $\llbracket \text{NatSpec}(u) \rrbracket$ is the least fixed point we know $\llbracket \text{NatSpec}(u) \rrbracket$ and $\llbracket \text{NatSpec}^\omega(u) \rrbracket$ coincide. \blacksquare

We now prove the key lemmas.

Lemma 8.11. *Below (1) is taken up to logical equivalence.*

- (1) $\vdash_{mix}^* \llbracket n \rrbracket_u \blacktriangleright \text{NatSpec}^{n+1}(u) \wedge \neg \text{NatSpec}^n(u)$;
- (2) $\text{NatSpec}^n(u) \equiv \exists i \leq n. (\text{NatSpec}^{i+1}(u) \wedge \neg \text{NatSpec}^i(u))$.
- (3) $\text{NatSpec}^0(u)$ gives a least fixed point.

Proof. We first show (1) and (2) simultaneously. We let

$$A(u, m) = \text{NatSpec}^{n+1}(u) \wedge \neg \text{NatSpec}^n(u).$$

Below we write $\mathcal{L}_{\approx}(\llbracket n \rrbracket_u)$ for the characteristic formula of $\llbracket n \rrbracket_u$ derived by \vdash_{mix}^* . Thus we are to prove $\mathcal{L}_{\approx}(\llbracket n \rrbracket_u) \equiv A(u, n)$. The base case ($n = 0$) is obvious. For the inductive case, we have:

$$\begin{aligned} \exists i \leq n+1. A(u, i) &\equiv \exists i \leq n. A(u, i) \vee A(u, n+1) \\ &\equiv \exists i \leq n. A(u, i) \vee (\text{NatSpec}^{n+1}(u) \wedge \neg \text{NatSpec}^n(u)) \\ &\equiv \text{NatSpec}^n(u) \vee (\text{NatSpec}^{n+1}(u) \wedge \neg \text{NatSpec}^n(u)) \\ &\equiv \text{NatSpec}^n(u) \vee \text{NatSpec}^{n+1}(u) \\ &\equiv \text{NatSpec}^{n+1}(u) \end{aligned}$$

As well as, through the axiom (4) in Lemma 8.7:

$$\begin{aligned} \mathcal{L}_{\approx}(\llbracket n+1 \rrbracket_u) &= \langle u(k) \rangle \langle k \triangleleft \text{succ} \rangle \vee p. (\langle \bar{k}p \rangle \top \circ \mathcal{L}_{\approx}(\llbracket n \rrbracket_u)) \\ &\equiv \langle u(k) \rangle \langle k \triangleleft \text{succ} \rangle \vee p. (\langle \bar{k}p \rangle \top \circ A(u, n)) \\ &\equiv \exists i \leq n+1. \langle u(k) \rangle \langle k \triangleleft \text{succ} \rangle \vee p. (\langle \bar{k}p \rangle \top \circ A(u, i-1)) \\ &\quad \wedge \neg \exists i \leq n. \langle u(k) \rangle \langle k \triangleleft \text{succ} \rangle \vee p. (\langle \bar{k}p \rangle \top \circ A(u, i-1)) \\ &\equiv \text{NatSpec}^{n+1}(u) \wedge \neg \text{NatSpec}^n(u) \end{aligned}$$

(b) is by Lemma 8.7 (1) and (4). ■

We can now establish the desired inhabitation result.

Proposition 8.12. $u : \text{natype} \models P \blacktriangleright \text{NatSpec}(u)$ iff $P \approx \llbracket n \rrbracket_u$ for some n .

Proof. By Lemma 8.11 (1), we have:

$$\llbracket \text{NatSpec}^n(u) \rrbracket = \llbracket \exists i \leq n. A(u, i) \rrbracket = \llbracket \exists i \leq n. \mathcal{L}_{\approx}(\llbracket i \rrbracket_u) \rrbracket = \cup_{i \leq n} \llbracket i \rrbracket_u. \quad (8.4.2)$$

We now reason:

$$\begin{aligned} \llbracket \text{NatSpec}(u) \rrbracket &= \llbracket \text{NatSpec}^0(u) \rrbracket \quad (\text{Lemmas 8.11}) \\ &= \cup_n \llbracket \text{NatSpec}^n(u) \rrbracket \quad (\text{definition}) \\ &= \cup_n \llbracket n \rrbracket_u \quad (\text{by (8.4.2) above}) \end{aligned}$$

as desired. ■

8.5 Comparison with Dam's Specification

There are three comments in comparison with Dam's specification in [31, Example 4.4]. First, Dam's assertion is more complex than $\text{NatSpec}(u)$ using a parametrisation of formulae (which can be unfolded), because he had to specify what natype has already ruled out, such as when at k a natural number does either $k \triangleleft \text{zero}$ or $k \triangleleft \text{succ}$ there are no other actions: no value output, no value input. All these actions are prevented by *typed* transition defined in § 2.3, hence we do not need to specify them.

Second, $\text{NatSpec}(u)$ is specified using the weak semantics, recapturing strong modality inside the weak regime through mixed modalities. This aids not only the conciseness of specifications but also reasoning. In contrast, Dam's logic is based on strong modalities. The use of weak modalities gives much more efficient reasoning in many encodings (the correctness of most encodings is proved up to weak semantics in the literature). Nevertheless, axioms for strong modality have more definite forms, as we have seen in Section B. Thus, from a foundational viewpoint it is important to consider strong transitions.

The ease of reasoning for the Mixed modality is not only limited to functional computation: this is because many systems exhibit locally deterministic behaviour. We shall see such an example in the next section.

As further note on the significance of the use of types in specifications and reasoning, we observe that $\text{NatSpec}(u)$ in fact entails mathematical induction. For this purpose we define predicates $\text{isZero}(x)$ and $\text{isSucc}(xy)$ as follows:

$$\begin{aligned} \text{isZero}(x) &\stackrel{\text{def}}{=} \mathcal{L}_{\approx}(\llbracket 0 \rrbracket_x) \\ \text{isSucc}(xy) &\stackrel{\text{def}}{=} \text{succ}\langle xy \rangle \end{aligned}$$

Now let

$$\Phi(A) \stackrel{\text{def}}{=} \forall x. (\text{isZero}(x) \supset A(x)) \wedge \quad (8.5.1)$$

$$\forall xp. ((\text{isSucc}(xp) \wedge \text{NatSpec}(p) \wedge A(p)) \supset A(x)) \quad (8.5.2)$$

Then we can show

$$\Phi(A) \supset \forall x. (\text{NatSpec}(x) \supset A(x)) \quad (8.5.3)$$

Further using an axiom which substantiates observability, we can also infer:

$$(\text{isZero}(x) \wedge \exists p. \text{isSucc}(yp)) \supset x \neq y$$

In this way many of the axioms for natural numbers can be obtained through syntactic reasoning.

9 Determinism and Elimination Results

This section demonstrates how our logic can be used for another, and significantly different, type discipline for the π -calculus, with essentially no change in proof rules. Our target, the affine π -calculus, was first introduced in [17] to give a fully abstract encoding of call-by-name and call-by-value PCF, which (for the call-by-value PCF) we shall outline at the end. Later, in §10, we show that the full-abstraction result of [17] can be lifted to the level of axiomatic semantics.

The interest in studying this type discipline is two-fold. First, as we just noted, this type discipline is quite different, indeed its calculus is now based on asynchronous communication. That all constructions in the previous sections extend to this type discipline is of technical interest.

Another interest is the application of the logic for a deterministic type discipline. In communicating systems, many of its fundamental parts are deterministic, combined with non-deterministic sharing [79, 124]. Thus it is useful to consider how we can treat specifications and reasoning for such behaviour: the following discussions give one starting point for such inquiry. This aspect is carried over to the next section, where we show a logic for call-by-value higher-order functions can be fully abstractly embedded in this logic.

Indeed, as we shall discuss at the end of the next section, the embedding results for deterministic languages cleanly extend to imperative features, suggesting the potential for uniform treatment of the diversity of deterministic and non-deterministic features in a single technical framework, thus substantiating our initial programs discussed in Introduction.

9.1 Types and Affine Processes

We have four *modes*: \downarrow for *affine input*, \uparrow the *affine output*, $!$ standing for *replicated input* and $?$ which is *replicated output*. Each mode p has a *dual* mode \bar{p} , given by $\bar{\downarrow} \stackrel{\text{def}}{=} ?$, $\bar{\uparrow} \stackrel{\text{def}}{=} \downarrow$ and $\bar{!} \stackrel{\text{def}}{=} p$. Types are now given by the following grammar.

$$\alpha ::= \text{nat} \mid \text{bool} \mid (\bar{\alpha})^! \mid (\bar{\alpha})^? \mid (\bar{\alpha})^\downarrow \mid (\bar{\alpha})^\uparrow \mid \&\{l_i : (\bar{\alpha}_i)\}_{i \in I}^\downarrow \mid \oplus\{l_i : (\bar{\alpha}_i)\}_{i \in I}^\uparrow$$

Adding recursive types is straightforward (cf. §2) and has been omitted for brevity. We extend modes with a new one for constant, denoted s , for *simple*. We define the *mode* of a type α , written $\text{md}(\alpha)$, with the following clauses.

- $\text{md}(\text{nat}) \stackrel{\text{def}}{=} s$, $\text{md}(\text{bool}) \stackrel{\text{def}}{=} s$.
- $\text{md}((\bar{\alpha})^p) \stackrel{\text{def}}{=} p$, $\text{md}(\&\{l_i : (\bar{\alpha}_i)\}_{i \in I}^\downarrow) \stackrel{\text{def}}{=} \downarrow$, and $\text{md}(\oplus\{l_i : (\bar{\alpha}_i)\}_{i \in I}^\uparrow) \stackrel{\text{def}}{=} \uparrow$.

Dualisation of types $\bar{\alpha}$ is also given as an extension of dualisation of type variables by the following clauses.

- $\overline{\text{nat}} \stackrel{\text{def}}{=} \text{nat}, \overline{\text{bool}} \stackrel{\text{def}}{=} \text{bool}.$
- $\overline{(\tilde{\alpha})^p} \stackrel{\text{def}}{=} (\tilde{\alpha})^p, \&\{l_i : (\tilde{\alpha}_i)\}_{i \in I}^\downarrow \stackrel{\text{def}}{=} \oplus\{l_i : (\tilde{\alpha}_i)\}_{i \in I}^\uparrow;$ and
 $\oplus\{l_i : (\tilde{\alpha}_i)\}_{i \in I}^\uparrow \stackrel{\text{def}}{=} \&\{l_i : (\tilde{\alpha}_i)\}_{i \in I}^\downarrow.$

A type α is *sequential* if

- If $\alpha = (\alpha_1, \dots, \alpha_n)^\downarrow$, then there is a unique i such that $\text{md}(\alpha_i) = \downarrow$ and for all $j \neq i$: $\text{md}(\alpha_j) = !$ or $\alpha_j \in \{\text{nat}, \text{bool}\}$. In addition α_i must be sequential for all i .
- If $\alpha = (\alpha_1, \dots, \alpha_n)^\uparrow$, then $\text{md}(\alpha_i) = !$ or $\alpha_i \in \{\text{bool}, \text{nat}\}$ for all i . In addition α_i must be sequential for all i .
- If $\text{md}(\alpha) \in \{!, \downarrow\}$ then $\bar{\alpha}$ is sequential.

From now on we assume all occurring types to be sequential.

An *extended sequential type* is a sequential type or \perp . We shall usually simply refer to extended sequential types as types. *Composition of types*, $\alpha \odot \beta$ is defined as the least partial, associative and commutative binary function on types such that the following requirements are met.

- $\text{nat} \odot \text{nat} = \text{nat}$ and $\text{bool} \odot \text{bool} = \text{bool}$
- $(\tilde{\alpha})^\downarrow \odot (\tilde{\alpha})^\uparrow = \perp, (\tilde{\alpha})^! \odot (\tilde{\alpha})^? = (\tilde{\alpha})^!,$ and $(\tilde{\alpha})^? \odot (\tilde{\alpha})^? = (\tilde{\alpha})^?$
- $\&\{l_i : (\tilde{\alpha}_i)\}_{i \in I}^\downarrow \odot \oplus\{l_i : (\tilde{\alpha}_i)\}_{i \in I}^\uparrow = \perp$

A (*typing-*)*environment*, Γ, Γ', \dots , is now a finite map from names and variables to extended sequential types. Such an environment is *simple* if $\Gamma(x)$ is simple for all $x \in \text{dom}(\Gamma)$.

Two environments Γ_1, Γ_2 are *compatible*, written $\Gamma_1 \asymp \Gamma_2$, if for all $a \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2)$: $\Gamma_1(a) \odot \Gamma_2(a)$ is defined. We extend \odot to environments as follows: $\Gamma_1 \odot \Gamma_2$ is defined exactly when $\Gamma_1 \asymp \Gamma_2$, and in this case:

$$(\Gamma_1 \odot \Gamma_2)(a) \stackrel{\text{def}}{=} \begin{cases} \Gamma_1(a) & a \in \text{dom}(\Gamma_1) / \text{dom}(\Gamma_2) \\ \Gamma_2(a) & a \in \text{dom}(\Gamma_2) / \text{dom}(\Gamma_1) \\ \Gamma_1(a) \odot \Gamma_2(a) & \text{else} \end{cases}$$

An environment Γ is *simple* if $\Gamma(a)$ is simple for all x . We set $\text{md}(\perp) \stackrel{\text{def}}{=} \perp$.

The grammar of processes is given next. Names, $a, b, c, \dots, k, k', \dots, x, y, \dots$ (note we do not have any distinction between session names, shared names and variables). Expressions, e, e', \dots are exactly those of §2.

$$\begin{aligned} P ::= & \mathbf{0} \mid \bar{k}(\bar{e}) \mid k \triangleleft l \langle \bar{e} \rangle \mid k(\bar{x}).P \mid !a(\bar{x}).P \mid k \triangleright [l_i(\bar{x}_i).P_i]_{i \in I} \mid P|Q \\ & \mid \text{if } e \text{ then } P \text{ else } Q \mid (\nu a)P \end{aligned}$$

Fig. 9 Typing rules for the affine π -calculus.

$$\begin{array}{c}
\frac{}{\Gamma, x : \alpha \vdash x : \alpha} \quad \frac{\Gamma \vdash e_i : \alpha \quad i = 1, 2 \quad \text{md}(\alpha) \neq !}{\Gamma \vdash e_1 = e_2 : \text{bool}} \quad \frac{\Gamma \vdash e_i : \text{nat} \quad i = 1, 2}{\Gamma \vdash e_1 + e_2 : \text{nat}} \\
\\
\frac{\Gamma \text{ simple}}{\Gamma \vdash 0} \quad \frac{\Gamma_i \vdash P_i \quad i = 1, 2 \quad \Gamma_1 \asymp \Gamma_2}{\Gamma_1 \odot \Gamma_2 \vdash P_1 | P_2} \quad \frac{\Gamma^{x:\alpha} \vdash P \quad \text{md}(\alpha) \in \{\perp, !\}}{\Gamma/x \vdash (\nu x^\alpha)P} \\
\\
\frac{\Gamma^{x:\alpha} \vdash P \quad \text{md}(\alpha) \in \{\perp, ?\}}{\Gamma, x : \alpha \vdash P} \quad \frac{\tilde{v} : \tilde{\alpha}, \uparrow ? \Gamma^{x:\alpha} \vdash P}{x : (\tilde{\alpha})^\downarrow, \Gamma \vdash x(\tilde{v}).P} \\
\\
\frac{\tilde{v}_i : \tilde{\alpha}_i, \uparrow ? \Gamma^{x:\alpha} \vdash P_i}{x : \& \{l_i : (\tilde{\alpha}_i)_{i \in I}^\downarrow, \Gamma \vdash x \triangleright [l_i(\tilde{v}_i).P_i]_{i \in I}\}} \quad \frac{\Gamma \text{ simple} \quad \Gamma^{x:\alpha} \vdash e_i : \alpha_i \quad i = 1, \dots, n}{x : (\tilde{\alpha})^\uparrow, \Gamma \vdash \bar{x} \langle \tilde{e} \rangle} \\
\\
\frac{\Gamma \text{ simple} \quad \Gamma^{x:\alpha} \vdash e_{i_j} : \alpha_{i_j}}{x : \oplus \{i : (\tilde{\alpha}_i)_{i \in I}^\uparrow \vdash x \triangleleft l \langle \tilde{e} \rangle\}}
\end{array}$$

We write $\bar{a}(\tilde{v})P$ for $(\nu \tilde{v})(\bar{a} \langle \tilde{v} \rangle | P)$ and $\bar{x}(nr).P$ for $(\nu r)(\bar{x} \langle nr \rangle | P)$, assuming that n is of type nat . The main difference between these processes and those of previous sections is that we use replication instead of recursion. Both are mutually definable, but replication is more convenient for embedding functional languages. We use the standard structural congruence \equiv .

Typing judgements for expressions are of the form $\Gamma \vdash e : \alpha$ where all names and variables occurring freely in e must be in $\text{dom}(\Gamma)$. *Typing judgements* for processes are of the form $\Gamma \vdash P$ with a similar restriction of free names and variables. Figure 9 gives the key rules, omitting most of the straightforward typing rules for expressions.

Remark 9.1. The rules in Figure 9 prohibit comparing $!$ -moded names to be compared for equality or inequality (we assume that $e \neq e'$ is a shorthand for $\neg(e = e')$). This is a key difference from the system in previous sections, where names could be compared for equality regardless of type. The advantage of imposing this mild restriction is that it enables an elegant form of semantics where elimination results can be proved easily.

Reductions are given by the following clauses.

- $\bar{x} \langle \tilde{e} \rangle | x(\tilde{v}).P \longrightarrow P[\tilde{w}/\tilde{v}]$, assuming that $\tilde{e} \downarrow \tilde{w}$
- $x \triangleleft l_i \langle \tilde{e} \rangle | x \triangleright [l_i(\tilde{v}_i).P_i]_{i \in L} \longrightarrow P_i[\tilde{w}/\tilde{v}_i]$, assuming that $\tilde{e} \downarrow \tilde{w}$.
- $\bar{x} \langle \tilde{e} \rangle | !x(\tilde{v}).P \longrightarrow P[\tilde{w}/\tilde{v}] | !x(\tilde{v}).P$, assuming that $\tilde{e} \downarrow \tilde{w}$

- If $P \longrightarrow P'$ then also $P|Q \longrightarrow P'|Q$ and $(\nu x)P \longrightarrow (\nu x)Q$
- If $P \equiv P' \longrightarrow Q' \equiv Q$ then $P \longrightarrow Q$.

The *contextual precongruence* is the least binary typed relation \lesssim_Γ such that $\Gamma \vdash P_i (i = 1, 2)$ implies: for every closing context $C[\cdot]$ such that $x : ()^\dagger \vdash C[P_i] (i = 1, 2)$ we have:

$$C[P_1] \Downarrow \text{ implies } C[P_2] \Downarrow.$$

The *contextual congruence* \cong_Γ is obtained as: $\lesssim_\Gamma \cap \lesssim_\Gamma^{-1}$. We usually write \lesssim for \lesssim_Γ and \cong for \cong_Γ .

9.1.1 Determinism of the Affine π -Calculus One interesting property of the affine π -calculus is that although a process might have reductions to different terms that are not structurally congruent, these different terms are nevertheless contextually indistinguishable. Here is an example:

$$P \stackrel{\text{def}}{=} (\nu x) \left(\underbrace{(\nu r)(\bar{x}\langle 0r \rangle | r(n).\bar{a}\langle n \rangle)}_{P_0} \mid \underbrace{(\nu r)(\bar{x}\langle 1r \rangle | r(n).\bar{b}\langle n \rangle)}_{P_1} \mid \underbrace{!x(nr).\bar{r}\langle n+2 \rangle}_{P'} \right)$$

P has the following reductions:

$$P \longrightarrow^* (\nu x) \left(\underbrace{\bar{a}\langle 1 \rangle | P_1 | P'}_{Q_0} \right) \quad P \longrightarrow^* (\nu x) \left(\underbrace{\bar{b}\langle 2 \rangle | P_0 | P'}_{Q_1} \right)$$

As no well-typed observer can input on x , Q and Q' are indistinguishable: $Q \cong Q'$. In addition we have:

$$Q_0 \longrightarrow^* \bar{a}\langle 1 \rangle | \bar{b}\langle 2 \rangle \quad Q_1 \longrightarrow^* \bar{a}\langle 1 \rangle | \bar{b}\langle 2 \rangle.$$

More generally:

Proposition 9.2. *If $P \longrightarrow^* Q$ and $P \longrightarrow^* Q'$ then:*

- $Q \cong Q'$.
- There is R such that $Q \longrightarrow^* R$ and $Q' \longrightarrow^* R$.

9.2 Semantics of Assertions Based on \cong

Now we interpret the logic of §3, but with a twist: instead of \approx we use \cong to quotient properties. When constructing models in §4, we defined the notion of *properties* in our logic. There, a property is a set of processes modulo \approx (cf. Definition 4.2, page 29), i.e. it is an equivalence class of \approx . But proving the elimination result for the ν operator in §9.4 later, and, more significantly, establishing

the logical full abstraction result in §10, is not feasible with \approx -based properties: this is because a basic logical axiom for equality such as $(\bar{a}\langle b \rangle \wedge b = c) \supset \bar{a}\langle c \rangle$, with b and c representing the equal *behaviour*, is not consistent with the direct transition-based reasoning in \approx , since outputting b and outputting c can never be equal in a transition-based equivalence such as \approx . This issue is however eliminated when we use \cong for defining properties, where, for example, an equation such as

$$\bar{a}\langle b \rangle !b(k).\bar{k}\langle 1 \rangle !c(k).\bar{k}\langle 1 \rangle \approx \bar{a}\langle c \rangle !b(k).\bar{k}\langle 1 \rangle !c(k).\bar{k}\langle 1 \rangle$$

makes sense. This is why we use \cong -based semantics of logics in the following.¹⁴

9.2.1 Semantics First, *properties* are now defined to be the sets of processes of the affine π -calculus up to \cong , rather than up to \approx . Similarly for *parametrised properties*. $\max p^\Gamma.\mathcal{P}$ is the maximum property (in the sense above) which satisfies \mathcal{P} . We set:

$$\begin{aligned} p|q &= \bigcup_{P \in p, Q \in q} [P|Q]_{\cong} \\ (\nu u)p &= \bigcup_{P \in p} [(\nu u)P]_{\cong} \\ \langle\!\langle \ell \rangle\!\rangle p' &= \{P \mid P \Longrightarrow P' \in p'\} \\ \langle \ell \rangle p' &= \{P \mid P \cong P_0 \xrightarrow{\ell} P' \in p'\} \\ \langle\!\hat{\ell}\!\rangle p' &= \{P \mid P \cong P_0 \xrightarrow{\hat{\ell}} P' \in p'\} \end{aligned}$$

In the last three clauses above, for illustration, we first present the semantic operations corresponding to the pure weak transition and the strong transition, then finally the clause for the weak transition. Note that if $P \Longrightarrow \cong \xrightarrow{\ell} P'$ and $Q \cong P$ then surely $Q \Longrightarrow \cong \xrightarrow{\ell} P'$ by reduction closure, so that the last clause is the result of combining the preceding two clauses (in other words, we do not need this clause if we are only to use $\langle\!\langle \ell \rangle\!\rangle A$ and $\langle \ell \rangle A$). We note:

Proposition 9.3. *The operations $p|q$, $(\nu u)p$, $\langle\!\hat{\ell}\!\rangle p'$ and $\langle \ell \rangle p'$ are well-defined, i.e. the results of these operations are again properties. Further these operations are closed under injective renaming.*

Proof. For the firsts two operations this is obvious. We consider the third operation. Assume $P \in \langle\!\hat{\ell}\!\rangle p'$ and $P \approx Q$. From the former, by definition, there is some P_0 such that $P_0 \xrightarrow{\hat{\ell}} P' \in p'$. Since $\equiv_{\text{sym}} \subset \approx$ from the construction we have $P_0 \approx Q$ hence, for ℓ which is not a linear output or even if it is so if it is not a server-typed name, we have $Q \xrightarrow{\ell} Q \approx P'$. If it is a linear output of a server-typed name then there are two cases:

¹⁴ Such “extensional” nature of name equality is more telling in the typed setting, but can also be observed in the untyped π -calculus [56].

- $\ell = \bar{k}a$. In this case if Q does not have the corresponding transition then it has the “renamed” transition say $\bar{k}b$ or $\bar{k}(b)$. In the first case we can surely find \equiv_{sym} -equivalent Q_0 such that $Q_0 \xrightarrow{\bar{k}b} Q' \approx P'$. In the second case again we can do the renaming under the prefix by the construction of \equiv_{sym} and we can find such Q_0 .
- $\ell = \bar{k}(a)$. This is the symmetric to the case above.

Hence as required. ■

We now present the defining clauses for the semantics of formulae, which are as in Section 4 except, for equality and quantification, we set:

$$\llbracket \Gamma \vdash e_1^\alpha = e_2^\alpha \rrbracket \xi = \begin{cases} \{ \Gamma \vdash P \mid \xi(e_1) = \xi(e_2) \} & (\text{md}(\alpha) \neq !) \\ \{ \Gamma \vdash P \mid P[\xi(e_1)\xi(e_2)/\xi(e_2)\xi(e_1)] \cong P \} & (\text{md}(\alpha) = !) \end{cases}$$

$$\llbracket \Gamma \vdash \forall x^\alpha . A \rrbracket \xi = \begin{cases} \max p^{\Gamma^\xi} . (\forall v : \alpha . p \subset \llbracket \Gamma, x : \alpha \vdash A \rrbracket (\xi \cdot x \mapsto v)) & (\text{md}(\alpha) \neq !) \\ \max p^{\Gamma^\xi} . (\forall q^{u:\alpha} . p \mid q \subset \llbracket \Gamma, x : \alpha \vdash A \rrbracket (\xi \cdot x \mapsto u)) & (\text{md}(\alpha) = !) \end{cases}$$

The second clause in each definition uses symmetries [52, 121] for server-typed channels, whose significance is discussed next.

9.3 Names, Distinctions and Symmetries

A *symmetry on P* [52] is a type-preserving injective renaming σ on P of server channels s.t. $P\sigma \approx P$. We generate $P \equiv_{\text{sym}} Q$ by all rules of \equiv including congruence rules together with: $P|R \equiv_{\text{sym}} P|R'$ if R' substitutes zero or more occurrences of a for b in R where $[ab/ba]$ is a symmetry on P . For example, $P|\bar{h}\langle a \rangle \equiv_{\text{sym}} P|\bar{h}\langle b \rangle$ when $P \equiv !a(k).\bar{k}\langle 1 \rangle \mid !b(k).\bar{k}\langle 1 \rangle$ with a and b typed ($\uparrow \text{nat}$)[!]. Since a and b have the same functionality their difference is not detected (unless names are matched).

9.3.1 Equality over server-typed channels The observation behind the clause for equation of server typed channels, which use symmetries, is that these “channels” can be treated as if they are “variables” for behaviours, as far as processes do not use the channel matching operation. For example, consider the following two processes:

$$P \stackrel{\text{def}}{=} \bar{k}\langle a \rangle \tag{9.3.1}$$

$$Q \stackrel{\text{def}}{=} \bar{k}\langle b \rangle \tag{9.3.2}$$

$$R \stackrel{\text{def}}{=} !a(k).\bar{k}\langle 1 \rangle \mid !b(k).\bar{k}\langle 1 \rangle \tag{9.3.3}$$

$$P' \stackrel{\text{def}}{=} P|R \tag{9.3.4}$$

$$Q' \stackrel{\text{def}}{=} Q|R \tag{9.3.5}$$

Now consider interacting with P' and Q' through k . P' will emit a , at which location we have the constant behaviour of emitting one: and Q' will emit b , at which location we have the same constant behaviour. So from the viewpoint of the users of a and b , there is no difference between them, except the identities of channels a and b . This identity is however insignificant as far as processes do not compare these channels directly. Thus logically speaking we may as well equate these a and b . If we have some assertions mentioning a , then the same assertions should be possible mentioning b instead. This means in effect we are treating a and b as embodiment of the underlying behaviours. This idea has been present in our preceding works on logics for higher-order functions [54].

9.3.2 Universal quantification over server-typed channels The preceding discussion also motivates the treatment of quantification over variables of server typed channels. The use of parallel composition in the defining clause may look strange: however it is necessary from the viewpoint of types. Suppose we wish to range x over all server typed names, including, as is standard, fresh names. But when u is fresh, the result of assigning u to x in A is typed as:

$$\Gamma, u:\alpha \vdash A[u/x] \quad (9.3.6)$$

which however can *never* be an assertion satisfied by P such that $\Gamma \vdash P$ since α is a server type and in this case we need a replicated process at u by the type discipline. For this reason we need to add a process (which may as well be a property) in parallel to the process in question. In effect we treat “processes” as if they were “values” (since they are stateless and never change the behaviour), so that one may consider the composition with r above as an assignment of a “value” r to x .

Formally we have the following basic properties of the quantifiers and substitutions (we state the results for all types).

Proposition 9.4 (equality, quantifier). *The following formulae are valid (we omit type annotations assuming any appropriate typing).*

1. (equality) $x = x$, $x = y \supset y = x$, and $x = y \wedge y = z \supset x = z$.
2. (quantification, 1) $\forall x.A \supset A[u/x]$, for each u with the same type as x .
3. (quantification, 2) $\forall x.A$, if A is valid with x free.
4. (quantification, 3) $\forall x.(A \supset B) \supset (A \supset \forall x.B)$, if x does not occur free in A .
5. (substitution) $\forall x.(x = y \supset A) \equiv A[y/x] \equiv \exists x.(A \wedge x = y)$.

Proof. All statements are standard for types other than server-typed names. Below we discuss only this case. (1) is immediate since the collection of all symmetries in a process form a group. (2) says that choosing any replicated process

of type $u : \alpha$ is as good as choosing it in another (existing) name, which holds because of symmetries and because the assertion for these processes are identical (except the initial subjects which symmetries cater for). ■

It is also notable that the current clause for universal quantification is equivalent to the following one where we let x range over arbitrary (similarly typed) channels in Γ and u .

$$\max p^{\Gamma\xi}. \forall q^{u:\alpha}, v:\alpha. p | q \subset [[\Gamma, x:\alpha \vdash A]](\xi \cdot x \mapsto v) \quad (9.3.7)$$

That is ranging x over all channels in $\Gamma\xi, u : \alpha$ of type α is the same thing as setting x to be u . This is because we are anyway letting u “ranging over” all possible processes of type $u : \alpha$, and because in the case of a server type, whether the process is situated at u or at say u' does not make any semantic difference.

9.4 Elimination of ν and \circ

Elimination of ν and \circ is closely related with basic axioms in the present logic. The results listed in this subsection all hold for the logic with strong transitions, with the strong versions of the respective pre-orders and bisimilarity. We need some preparation.

- P is *linearly receptive* [17, 98] if $\Delta \vdash P$ is derived by restricting Γ in the prefix rules in Figure 1 so that it does not contain input or branching session channels. Then P is *functional* if all shared names in Δ are server types and it is linear receptive.
- P is *acyclic* if it is derived without $\mu t. \tau$ and does not contain circular dependency among names induced by prefixing (e.g. $!a.\bar{b}|!b.\bar{c}|!c.\bar{a}$ has circular dependency among a, b and c , see [120] for a formal treatment).

Linear-receptiveness makes all linear output actions to be consumable: whereas acyclicity leads to strong normalisation, as studied in [120]. These conditions make processes essentially those which compute functions. In particular their reductions become Church-Rosser. We also assume:

Convention 9.5. In the rest of this section, we always assume the semantics of assertions based on \cong , stipulated in §9.2.

We note the following elimination results on \circ hold for both the \approx -based semantics and the \cong -based semantics: only the proof of the elimination result for ν uses the \cong -based semantics. As noted in §9.2, the key difference between these two semantics is the existence of symmetries on server-typed channels.

The same proof systems as given in §5 and §6 are complete for the \cong -based semantics, similarly all axioms we have listed so far are sound under the semantics with \cong .

We shall use the following axioms for the elimination results.

Proposition 9.6 (axioms for elimination).

1. Assume a is server-typed.

$$\begin{aligned} \langle a(k), \Gamma \rangle A \circ \langle \bar{a}(k), \Delta \rangle B &\equiv \langle a(k), \Gamma \rangle A \circ \nu k. (A \circ B) \\ \langle a(k) \rangle A \circ \langle \ell \rangle B &\equiv \langle a(k) \rangle A \circ \langle \ell \rangle \nu a. (\langle a(k) \rangle A \circ B) \\ \langle a(k) \rangle A \circ \langle \bar{a}(k) \rangle B &\equiv \langle a(k) \rangle A \circ \nu k. (A \circ B) \end{aligned}$$

Below assume (1) the typing of the whole formulae only use server types and no variables for channel types occur, and (2) $\text{fn}(A) \cap \text{fn}(B) = \emptyset$.

$$A \circ B \supset A \wedge B$$

2. Assume each of ℓ, ℓ' is either linear or client typed.

$$\begin{aligned} \langle \ell \rangle A \circ \langle \ell' \rangle B &\equiv \langle \ell \rangle (A \circ \langle \ell' \rangle B) \wedge \langle \ell' \rangle (\langle \ell \rangle A \circ B) \\ \langle \ell \rangle A \circ \langle \ell \rangle B &\equiv \langle \ell \rangle ((A \circ \langle \ell \rangle B) \vee (\langle \ell \rangle A \circ B)) \wedge \\ &\quad \langle \ell \rangle (A \circ \langle \ell \rangle B) \wedge \langle \ell \rangle (\langle \ell \rangle A \circ B) \\ &\equiv \langle \ell \rangle (A \circ \langle \ell \rangle B) \wedge \langle \ell \rangle (\langle \ell \rangle A \circ B) \\ \langle \ell \rangle A \circ \langle \ell \rangle B &\equiv \langle \ell \rangle (A \circ \langle \ell \rangle B) \wedge \langle \ell \rangle (\langle \ell \rangle A \circ B) \end{aligned}$$

3. Assume ℓ is linearly typed.

$$\begin{aligned} \langle \ell \rangle A \circ \langle \bar{\ell} \rangle B &\equiv \nu \text{bn}(\ell). (A \wedge B) \\ \langle \ell \rangle A \circ \langle \bar{\ell} \rangle B &\equiv \langle \tau \rangle (A \circ B) \end{aligned}$$

Below A is \circ -free (resp. ν -free) if \circ (resp. ν) does not occur in A . A is *approximately* \circ -free if \circ occurs only in fixed point formulae whose finite unfoldings are \circ -free.

Theorem 9.7 (elimination of ν and \circ).

1. Assume for each $\nu x^\rho . B$ which appears in A , ρ is a server type. Then there is an algorithm to find ν -free C s.t. $A \equiv C$.
2. Let $\vdash_{\text{mix}}^* P \blacktriangleright A$ for a closed functional process P typed under Γ whose range contains only server types. Then there is an algorithm to find ν -free and approximately ν -free B s.t. $A \equiv B$. Further if P is acyclic, there is an algorithm to find ν -free and ν -free B s.t. $A \equiv B$.

Proof. (1) is by the following sequence of logical equivalences. Let the (hidden) x be typed with a server type.

$$P \models \nu x.A \equiv \exists Q.(P \approx (\nu a)Q, Q \models A[a/x]) \quad (9.4.1)$$

$$\equiv \exists Q.(Q \approx !a(k).R \mid P, Q \models A[a/x]) \quad (9.4.2)$$

$$\equiv \exists Q, Q_0.(Q \approx Q_0 \mid P, \text{fn}(Q_0) = \{a\}, Q \models A[a/x]) \quad (9.4.3)$$

$$\equiv \exists Q_0.(P \mid Q_0 \models A[a/x], \text{fn}(Q_0) = \{a\}) \quad (9.4.4)$$

$$\equiv P \models \exists x.A \quad (9.4.5)$$

For (2) we only list an outline. A later version will provide full details. First if a process is not acyclic then by Proposition 9.6 we can eliminate each “redex” one by one, the procedure which we can write $A \rightsquigarrow B$. Then we can show \rightsquigarrow precisely corresponds to the extended reduction in [120], hence it is strongly normalising, erasing all composed names. Finally we can use the final axiom in Proposition 9.6 (1). For the non-acyclic case, it is possible we cannot apply the elimination of server channels in which case we resolve circularity through the formula:

$$\Psi \stackrel{\text{def}}{=} \nu x.((a(k).A) \circ [x \mapsto a]^A) \quad (9.4.6)$$

where x occurs in A and $[x \mapsto a]^A$ is a “copy-cat” specification induced from A , copying each input to an output and back. We then construct the sequence of the following formulae:

$$\begin{aligned} \Psi(a, 0) &\stackrel{\text{def}}{=} \nu x.((a(k).A) \circ !x(k).\text{noact}(k, a)) \\ \Psi(a, n+1) &\stackrel{\text{def}}{=} \nu x.((a(k).A) \circ \Psi(x, n)) \end{aligned}$$

whose limit can be easily represented by the least fixed point. Each of these formulae can be resolved into \circ -free formula, thus we obtain approximately \circ -free formulae as a whole, as required. ■

Remark 9.8 (\circ -elimination for stateful processes). We also outline how \circ may be eliminated for stateful processes (processes which are typed under the typing in Section 2). We use the following typed processes:

- P is *simple* if $\Delta \vdash P$ is derived by restricting Γ in **Rec** in Figure 1 contains a single input shared type; and Γ in all prefix rules in Figure 1 is empty.
- P is *first-order* if α in $\downarrow \alpha; \tau$ and $\uparrow \alpha; \tau$ are restricted to nat or bool .

The simpleness condition means a recursive process starts from a shared input and after a session, again recurs to the same input name, possibly with a different state; and interactions in-between is never interfered with another non-deterministic interactions and two sessions never interleave. We also use the

same acyclicity condition on stateful processes. Now let P be a simple, acyclic, and first-order process and $\vdash_{may}^* P \blacktriangleright A$. Then there is a natural algorithm to calculate B which is \circ -free from the May characteristic formula of A , through the application of some of the basic May axioms. We believe these axioms preserve the May preorder so that we have $\Gamma \models Q \blacktriangleright B$ implies $P \sqsubseteq_{may} Q$ (details are to be checked). The result can be strengthened to Must preorder.

9.5 Examples

Consider the following simple process.

$$P \stackrel{\text{def}}{=} \bar{f}(nr)r(x).\bar{f}\langle mr'\rangle r'(y).\bar{h}\langle x \text{ div } y \rangle \quad (9.5.1)$$

The process P invokes f with a number n and a return channel r . On r it receives f 's result x . Then P sends a second number to f and awaits the result y . Then it divides the first answer by the second and emits the result via h . Let Γ be $f : (\text{nat}(\text{nat})^\dagger)^?$, $h : (\text{nat})^\dagger$. Then P is typed with Γ .

We wish to let this process compute a binomial coefficient, $\binom{n}{m}$, so we assert for P as follows (we omit universal quantification over n and m and omit the condition $n \geq m$).

$$\langle\langle \bar{f}(nr) \rangle\rangle \langle\langle rn! \rangle\rangle \langle\langle \bar{f}\langle mr' \rangle \rangle\rangle \langle\langle r' m! \rangle\rangle \langle\langle \bar{h}\langle \binom{n}{m} \rangle \rangle\rangle \top \quad (9.5.2)$$

This assertion assumes that the environment – the process situated at f – calculates a factorial, and ensures the property of P only in that case.

Now consider the process given as:

$$Q \stackrel{\text{def}}{=} !f(nr).\bar{r}\langle n! \rangle \quad (9.5.3)$$

which is typed with $f : (\text{nat}(\text{nat})^\dagger)^\dagger$. We can assert for Q as:

$$\langle\langle f(nr) \rangle\rangle \langle\langle \bar{r}n! \rangle\rangle \top \quad (9.5.4)$$

Now consider composing P and Q . The corresponding assertion is $A \circ B$. Using the axioms we discuss later, we indeed obtain:

$$(9.5.5) \circ (9.5.4) \supset \forall n^{\text{nat}}, m^{\text{nat}} : n \geq m. \langle\langle \bar{h}\langle \binom{n}{m} \rangle \rangle\rangle \top \quad (9.5.5)$$

That is, if we combine these two processes, then there can be an output $\langle\langle \bar{h}\langle \binom{n}{m} \rangle \rangle\rangle$ at h .

9.6 Call-by-value PCF in the Affine π -Calculus

Call-by-value PCF, which we shall hereafter write PCFv for brevity, is an idealised functional language with call-by-value evaluation order. In [17] a fully abstract compositional embedding of PCF (using call-by-name) into the affine π -calculus was presented, whose encoding we outline below. First we summarise the syntax of PCFv. PCFv-types, values (V, W, \dots) and terms (M, N, \dots) are given by the following grammar.

$$\begin{aligned} \alpha &::= \text{nat} \mid \text{bool} \mid \alpha \Rightarrow \beta \\ V &::= x \mid c \mid \lambda x^\alpha. M \mid \mu x^{\alpha \Rightarrow \beta}. \lambda y^\alpha. M \\ M &::= V \mid \text{op}(\tilde{M}) \mid MN \mid \text{if } M \text{ then } N \text{ else } N' \end{aligned}$$

We assume the standard evaluation relation over closed terms, $M \Downarrow V$, induced from the reduction relation $M \longrightarrow M'$, again defined over closed terms. Using the evaluation relation (or equivalently reduction), we let \cong_λ denote the standard contextual congruence for typed PCFv-terms.

The embedding of PCFv into the affine π -calculus is denoted $\llbracket M \rrbracket_u$ on terms and α° on types. We begin with the latter.

$$\text{nat}^\circ \stackrel{\text{def}}{=} \text{nat} \quad \text{bool}^\circ \stackrel{\text{def}}{=} \text{bool} \quad (\alpha \Rightarrow \beta)^\circ \stackrel{\text{def}}{=} (\overline{\alpha^\circ}(\beta^\circ)^\dagger)^\dagger$$

The call-by-value term is in fact given a type $(\alpha^\circ)^\dagger$, which represents an output of a value of type α , following the preceding encoding by Milner.

The translation of terms follows that of the typing.

$$\begin{aligned} \llbracket x \rrbracket_u &\stackrel{\text{def}}{=} \bar{u}(x) \\ \llbracket \lambda x. M \rrbracket_u &\stackrel{\text{def}}{=} \bar{x}(a)!a(xm). \llbracket M \rrbracket_m \\ \llbracket \mu g. \lambda x. M \rrbracket_u &\stackrel{\text{def}}{=} \bar{u}(a)(\nu g)(!a(xm). \llbracket M \rrbracket_m | g(\nu r). \bar{a}(\nu r)) \\ \llbracket MN \rrbracket_u &\stackrel{\text{def}}{=} (\nu m)(\llbracket M \rrbracket_m | m(a). (\nu n)(\llbracket N \rrbracket_n | n(b). \bar{a}(bu))) \\ \llbracket n \rrbracket_u &\stackrel{\text{def}}{=} \bar{u}(n) \\ \llbracket \text{if } M \text{ then } N \text{ else } N' \rrbracket_u &\stackrel{\text{def}}{=} (\nu m)(\llbracket M \rrbracket_m | m(a). \text{if } a \text{ then } \llbracket N \rrbracket_u \text{ else } \llbracket N' \rrbracket_u) \\ \llbracket M + N \rrbracket_u &\stackrel{\text{def}}{=} (\nu m)(\llbracket M \rrbracket_m | m(a). (\nu n)(\llbracket N \rrbracket_n | n(b). \bar{u}(a+b))) \end{aligned}$$

We can now show that the following holds.

$$\Gamma \vdash M : \alpha \quad \text{implies} \quad \overline{\Gamma^\circ}, u : (\alpha^\circ)^\dagger \vdash \llbracket M \rrbracket_u.$$

Here Γ° is the pointwise extension of $(\cdot)^\circ$ to environments, and dualisation $\overline{(\cdot)}$ is extended to environments in a pointwise manner too. It is possible to prove the

following *equational full abstraction theorem* which says that the embedding is precise, i.e. reflects the semantics of PCFv in the affine π -calculus without losing information.

In Section 10, we use this encoding for the embedding of the program logic for call-by-value PCF into that for the π -calculus.

10 Logical Full Abstraction of PCFv

10.1 From Equational to Logical Embedding

A notable effects of type disciplines in the π -calculus is to enhance the semantic precision of embeddings of diverse calculi and languages as processes. When a typing discipline is sufficiently powerful, the embedding enjoys *equational full abstraction* [17], that is, the embedding preserves all equational properties of the original calculus/language through the compositional decomposition of its constructs into name passing. §9.6 presented an example of an equationally fully abstract embedding of PCFv into the affine π -calculus. In the following we demonstrate that the presented process logic inherits this precise embeddability at the logical level, using a relatively-complete logic for call-by-value PCF studied in our preceding work [54]. We show that this program logic is embeddable, in terms of validity and provability, in the process logic through the encoding of formulae, which naturally arise from those of types and terms.

In this section we shall use \cong as the basis of the semantics of the process logic, which we introduced in §9, since in this way we obtain the semantics of quantifiers which precisely correspond to those in the program logic in [54].

Convention 10.1. *Throughout the technical development in the rest of the section, we use the logic in §9 with its semantics based on \cong .*

For typed processes as given in §9 with the semantics based on \cong , exactly the same proof systems as given in Sections 5 and 6 are sound and complete. This is because the semantic difference due to the use of \cong is absorbed by the consequence rule: all compositional rules can capture precisely the interactional content, while the syntactic variance up to the logical equivalence is captured by the consequence rule.

10.2 Logic for PCFv

As a target logic, we use the relatively complete total-correctness program logic for call-by-value PCFv (cf. §9.6), studied in [54]. This logic has the following syntax (types are those of §9.6).

$$\begin{aligned} e &::= c \mid x \mid \text{op}(\tilde{e}) \\ A &::= e = e' \mid A \wedge B \mid \forall x^\alpha. A \mid \neg A \mid x \bullet y \searrow z \end{aligned}$$

We use the same meta-symbols ranging over expressions and formulae of the logic for PCFv as we do for the target logic (for the affine π -calculus). In the first line (expressions), $\text{op}(\tilde{M})$ denotes standard first-order operations. For brevity we

omit products and sums as well as typing rules. In the second line (formulae), $x \bullet y \searrow z$, called *evaluation formula*, specifies that a function x , when applied an argument y , converges and results in a value z . The semantics of these formulae follows exactly [54], whose key clauses are given as follows. Below we let ξ, ξ', \dots range over finite maps from variables to closed values.

$$\xi \models e_1 = e_2 \equiv \llbracket e_1 \rrbracket \xi \cong_\lambda \llbracket e_2 \rrbracket \xi \quad \xi \models x \bullet y \searrow z \equiv \xi(x)\xi(y) \Downarrow \xi(z)$$

Logical connectives and quantifiers are interpreted as usual, following the semantics of (classical) predicate logic with equality, cf. [74].

Sequent in the program logic is written:

$$[A]M :_u [B],$$

We write

$$\models [A]M :_u [B],$$

This last judgement says that if the free variables in M satisfy A , the program M terminates and whose result, named by u , satisfies B . The definition of the relation $\models [A]M :_u [B]$ is formally defined as follows, precisely following [54].

$$\xi \models A \quad \supset \quad \exists V. (M\xi \Downarrow V \wedge \xi, u : V \models B)$$

There are simple proof rules which can infer, with the help of the consequence rule (hence an oracle to infer validity of assertions, in particular entailment), precisely the set of valid judgements for formulae which are saturated upwards with respect to the standard semantic ordering [54], see Remark 10.7 later. We shall write this provability relation $\vdash [A]M :_u [B]$.

10.3 Embedding PCFv into the π -Calculus

The embedding of PCFv into the affine π -calculus is given in §9.6. For convenience of presentation we add the following additional translation of values. For closed values V of function type $\langle\langle V \rangle\rangle_x$ is a process such that $\llbracket V \rrbracket_k \stackrel{\text{def}}{=} (\nu x)(\bar{k}\langle x \rangle \mid \langle\langle V \rangle\rangle_x)$, i.e. $\langle\langle V \rangle\rangle_x$ singles out the replicated part from the encoding. For example $\langle\langle \lambda x.M \rangle\rangle_u = !u(xm). \llbracket M \rrbracket_m$.

We can now show that the following holds.

Proposition 10.2. *Let Γ° , resp. $\bar{\Gamma}$, be the pointwise extension of $(\cdot)^\circ$, resp dualisation $\bar{(\cdot)}$ to environments.*

- $\alpha \vdash M$ implies $\bar{\Gamma}^\circ, u : (\alpha^\circ)^\dagger \vdash \llbracket M \rrbracket_u$.
- $\alpha \vdash V$ with α a function type implies $u : \alpha^\circ \vdash \langle\langle V \rangle\rangle_u$.

It is easy to show that $\llbracket \cdot \rrbracket$ is sound, even in the absence of typing constraints on the target calculus. The definability result given next shows that the affine typing discipline of §9 constrains the target calculus sufficiently so that typable processes (at translations of PCFv types) are exactly the encodings of PCFv-types (up to observational congruence).

Lemma 10.3 (definability). *If P is typed with $x : \alpha^\circ$ then $P \cong \langle\langle V \rangle\rangle_x$ for some closed value V of functional type α . Similarly if P is typed with $x : (\alpha^\circ)^\dagger$, then $P \cong \llbracket M \rrbracket_x$ for some closed M of type α .*

Proof. We show the latter by considering what corresponds to the encoding of open terms. We show by the induction on the the height of α :

$$\bar{\Gamma}^\circ, h : (\alpha^\circ)^\dagger \vdash P$$

implies P composed with processes under the typing dual to Γ° is equivalent to the encoding of a PCFv-term of the corresponding type. We use a method similar to the one given in [119], showing such P is sequentially typed in the sense of [17], which allows us to reduce the term to the corresponding let-form. The base case is when α is an atomic type and Γ is empty, which is immediate. When α is a function type, say $\alpha_1 \Rightarrow \alpha_2$, then we should show P as above is equivalent to the encoding of $\lambda x.M$. The shape of P is by functional typing:

$$\bar{h}(k)!k(xm).Q$$

assuming we fold all hidden compositions with replicated processes into Q . We then consider the “spine” in Q which contains output at m and neglect all interactions not in the spine. The result is typable by the sequential typing in [17] (except the disregarded interactions which do not affect an output at m), hence by the same argument as in [17] we can translate the intermediate actions into the let-form, together with induction hypothesis. ■

As a direct corollary we obtain the result mentioned in §9.6:

Corollary 10.4 (full abstraction). *Let M and N be closed PCFv-terms of type α . Then $M \cong_\lambda N$ if and only if $\llbracket M \rrbracket_k \cong \llbracket N \rrbracket_k$ under the type $k : (\alpha^\circ)^\dagger$; end. Similarly if V and W are closed values of the same functional type say α , then $V \cong_\lambda W$ if and only if $\langle\langle V \rangle\rangle_x \cong \langle\langle W \rangle\rangle_x$.*

10.4 Logical Embedding

Having embedded PCFv into the affine π -calculus, we now embed the logic for PCFv into the corresponding process logic. We get ideas about how to embed logical formulae, by looking at how functions are translated, using types.

For the encoding of formulae, we use the following mapping:

$$\begin{aligned}
[[e_1 = e_2]] &\equiv e_1 = e_2 \\
[[A \wedge B]] &\equiv [[A]] \wedge [[B]] \\
[[\forall x^\alpha. A]] &\equiv \forall x. [[A]] \\
[[\neg A]] &\equiv \neg [[A]] \\
[[x \bullet y \setminus z]] &\equiv \langle\langle x(ya) \rangle\rangle \langle\langle \bar{a}z \rangle\rangle \top
\end{aligned}$$

The last clause of the translation decomposes an evaluation formula into a modal formula with a May-modality (which, in deterministic computation, corresponds to total correctness). We can now translate the satisfiability $\xi \models A$ by mapping ξ and A individually, i.e.:

$$[[\xi \models A]] \stackrel{\text{def}}{=} [[\xi]] \models [[A]] \quad (10.4.1)$$

where we set $[[x_1 : V_1, \dots, x_n : V_n]] \stackrel{\text{def}}{=} [[V_1]]_{x_1} | \dots | [[V_n]]_{x_n}$. We now observe:

Lemma 10.5. *Let A be an assertion in the PCFv-logic. Then A is valid in the PCFv-logic if and only if $[[A]]$ is valid in the process logic.*

Proof. We show, by induction, that $\xi \models A$ and $[[\xi \models A]]$ coincide. The induction in fact proves the coincidence of validity simultaneously for both A and its negation.

- (1) $e = e'$ and $e \neq e'$. If their types are atomic types there is no issue. If both have higher-order types then we can use Corollary 10.4.
- (2) $\forall x.A$ and $\exists x.A$. For this we reason:

$$\begin{aligned}
\xi \models \forall x.A &\equiv \forall V. (\xi, x : V \models A) \\
&\equiv \forall V. [[\xi, x : V \models A]] \\
&\equiv \forall x : \alpha^\circ \vdash P. P | [[\xi \models A]] \\
&\equiv [[\xi \models \forall x.A]]
\end{aligned}$$

where the second to the last step is by Lemma 10.3. This also proves the case for existential.

- (3) For evaluation formulae, it suffices to consider the case when the formula only uses variables. We take terms up to \cong_λ and \cong . Let us assume:

$$\begin{aligned}
\xi(x) &= V \\
\xi(y) &= W \\
\xi(z) &= U.
\end{aligned}$$

We write $M \Downarrow \cong N$ to mean that some value L exists such that $M \Downarrow L$ and $L \cong N$.

$$\begin{aligned}
\xi \models x \bullet y \searrow z &\equiv VW \Downarrow \cong_\lambda U \\
&\equiv (\mathbf{v}x)(\langle\langle V \rangle\rangle_x \mid (\mathbf{v}y)(\bar{x}\langle yu \rangle . \langle\langle W \rangle\rangle_y)) \cong \llbracket U \rrbracket_u \\
&\equiv \llbracket \xi \rrbracket \models \langle\langle x \langle ya \rangle \rangle \langle\langle \bar{a} \langle z \rangle \rangle \top
\end{aligned}$$

which also proves for its negation.

This exhausts all cases. ■

Theorem 10.6 (logical full abstraction). *For each typable PCFv-term M , we have $\models [A]M :_u [B]$ if and only if $\llbracket M \rrbracket_k \models \llbracket A \rrbracket \triangleright \exists x. (\langle\langle \bar{k}x \rangle \rangle \top \wedge \llbracket B \rrbracket [x/u])$.*

Proof. First $\models [A]M :_u [B]$ is the same thing as

$$\forall \xi \models A. (M\xi \Downarrow V \wedge \xi, u : V \models B) \quad (10.4.2)$$

By Lemma 10.5 this is in turn equivalent to, when M is typed with a higher-order type (the case when M is typed with a first-order type, where the encoding simply has a value output, is simpler and omitted),

$$\forall P \models \llbracket A \rrbracket. ((\llbracket M \rrbracket_k \mid P) \xrightarrow{\bar{k}(u)} R \mid P \wedge R \mid P \models \llbracket B \rrbracket) \quad (10.4.3)$$

Through the elimination of \mathbf{v} (cf. Theorem 9.7) we obtain:

$$\llbracket M \rrbracket_k \models \llbracket A \rrbracket \triangleright \exists x. (\langle\langle \bar{k}x \rangle \rangle \top \wedge \llbracket B \rrbracket [x/u]), \quad (10.4.4)$$

Hence as required. ■

Remark 10.7 (extension to provability). The completeness result in Theorem 10.6 immediately extends to provability. We use the completeness result for the PCFv logic [54]. This result holds for the class of total correctness properties, called *upper-closed formulae*. First, we say a formula A of PCFv-logic with $\text{fv}(A) = \{u\}$ is *upper-closed with respect to u* [54] if, whenever V named u satisfies A , and if $V \lesssim W$ (where \lesssim is the pre-congruence given in §9), then W named u also satisfies A . Through the translation, the upper-closed formulae in PCFv-logic precisely correspond to May-saturated formulae in §7, hence by the completeness on both sides, we are done.

10.5 Further Logical Embeddings

The translation of partial correctness for PCFv, as given in [54], can be done using Must modality, leading to logical full abstraction for the partial correctness logic in [54]. In this case, the assertion $\{A\}M :_k \{B\}$ is translated into

$$\llbracket M \rrbracket_k \blacktriangleright \llbracket A \rrbracket \triangleright \exists x. (\llbracket \bar{k}x \rrbracket \top \wedge \llbracket B \rrbracket [x/u]) \quad (10.5.1)$$

that is by exchanging the May modality with the Must modality.

This result directly extends to imperative features: starting from Hoare logic, for the first-order imperative functions, the extensions for higher-order procedures, aliasing and local state are similarly embeddable. Below we show the case for Hoare logic, for both total and partial correctness. The framework follows the same principle as we discussed above for the logic for PCFv. We first map each number theoretic assertion say $A(\tilde{x})$, where $A(\tilde{x})$ indicates an assertion whose free (program) variables are covered by \tilde{x} , as follows.

$$\text{Var}(A(\tilde{x})) \stackrel{\text{def}}{=} \exists \tilde{v}. (\bigwedge_{1 \leq i \leq n} \text{VarSPec}(x_i, v_i) \wedge A(\tilde{v})) \quad (10.5.2)$$

where we used the specification for variables $\text{VarSPec}(xv)$ in §8.2. Using this encoding, we translate the partial correctness assertion

$$\{A(\tilde{x})\} C \{B(\tilde{x})\} \quad (10.5.3)$$

as follows, where $\llbracket C \rrbracket_k$ is the standard mapping of imperative programs to processes [115].

$$\llbracket C \rrbracket_k \blacktriangleright \text{Var}(A(\tilde{x})) \triangleright \llbracket \bar{k} \rrbracket \text{Var}(B(\tilde{x})). \quad (10.5.4)$$

Similarly, we translate the total correctness assertion

$$[A(\tilde{x})] C [B(\tilde{x})] \quad (10.5.5)$$

as follows:

$$\llbracket C \rrbracket_k \blacktriangleright \text{Var}(A(\tilde{x})) \triangleright \langle \langle \bar{k} \rangle \rangle \text{Var}(B(\tilde{x})). \quad (10.5.6)$$

which replaces the Must modality in the partial correctness assertion with the May modality. The interest of these mappings is not only that they are encodable, but also the involved process typing as well as logical encodings are simple and tractable, in a way to allow the extension of the original assertion method to a richer realm of interacting processes.

As one of such points, observe that the assumption part and the conclusion part of (10.5.4) and (10.5.6) mention *all* program variables which may occur in a program. This is because we cannot assume, in the present logic, that they do

behave as “variables”: in Hoare logic we can omit them, since we are implicitly assuming the memory (store, heap, etc.) all behave as variables: but in the present setting it may behave as almost anything as far as the typing is correct! Such an “implicit assumption” may as well be left implicit as induced by types. As a simple use of this idea, we may consider that the behaviour can be either a variable, or a diverging (or failing) one, as an implicit assumption. This is the case when we consider a resource sensitive logic such as separation logic. This suggests the wide spectrum of logical specifications the present framework suggests.

Thus we have obtained a framework where we can reason about the diversity of behaviours on a uniform technical footing. In principle we can specify and reason about arbitrary sequential, concurrent, communications, shared variables, functional, and imperative behaviours as far as they are embeddable into the π -calculus. Underlying is the operational and type structures studied for a long time by researchers in programming languages and process theories, which articulate the extremely rich semantic structures hidden in the π -calculus. This interplay offers us a rich basis for future investigations, in both applications and basic theories.

References

1. Coq home page. <http://coq.inria.fr>.
2. Hol home page. <http://hol.sourceforge.net>.
3. Isabelle home page. <http://isabelle.in.tum.de>.
4. The Java Modeling Language (JML) home page. <http://www.jmlspecs.org/>.
5. Pvs home page. <http://pvs.csl.sri.com>.
6. Web Services Choreography Description Language: Primer 1.0. <http://www.w3.org/TR/ws-cdl-10-primer/>.
7. S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51, 1991.
8. S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS'98*, pages 334–344, 1998.
9. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.
10. R. Amadio and M. Dam. A modal theory of types for the π -calculus. In *FTRTFT'96*, volume 1135 of *LNCS*, pages 347–365, 1996.
11. H. R. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal μ -calculus. In *LICS*, pages 144–153, 1994.
12. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
13. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, pages 203–213, 2001.
14. M. Berger. A program logic for sequential higher-order control (1): stateless case. type script, 50pp, October 2007.
15. M. Berger. Program logics for sequential higher-order control. In *FSEN'09*, LNCS, 2009.
16. M. Berger, K. Honda, and N. Yoshida.
17. M. Berger, K. Honda, and N. Yoshida. Sequentiality and the π -calculus. In *TLCA'01*, volume 2044 of *LNCS*, pages 29–45, 2001.
18. M. Berger, K. Honda, and N. Yoshida. Genericity and the π -calculus. In *Proc. FoSSaCS'03*, number 2620 in LNCS, 2003.
19. M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing for higher-order imperative functions. In *ICFP'05*, pages 280–293, 2005. Full version is available at: www.dcs.qmul.ac.uk/~kohei/logics.
20. J. A. Bergstra and J. W. Klop. Algebra of communicating processes. *Theoretical Computer Science*, 37:77–121, 1985.
21. M. M. Bonsangue and A. Kurz. Pi-calculus in logical form. In *LICS*, pages 303–312, 2007.
22. M. Boreale and D. Sangiorgi. Some congruence properties for π -calculus bisimilarities. Technical Report RR-2870, INRIA-Sophia Antipolis, 1996. Available electronically as <ftp://ftp.dcs.ed.ac.uk/pub/sad/congruence96.ps.gz>.
23. L. Caires. Behavioral and spatial observations in a logic for the pi-calculus. In *FoSSaCS*, volume 2987 of *LNCS*, pages 72–89, 2004.
24. L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *I & C*, 186(2):194–235, 2003.
25. L. Caires and L. Monteiro. Verifiable and executable logic specifications of concurrent objects in $\mathbb{1}_{\pi}$. In *ESOP*, pages 42–56, 1998.
26. D. Callahan, B. L. Chamberlain, and H. P. Zima. The cascade high productivity language. In *HIPS*, pages 52–60, 2004.
27. L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *POPL*, pages 365–377, 2000.

28. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. Special Issue on Coordination, D. Le Métayer Editor.
29. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA'05*. ACM Press, 2005.
30. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.
31. M. Dam. Proof systems for pi-calculus logics. In *Logic for Concurrency and Synchronisation*, R. de Queiroz (ed.), Trends in Logic, Studia Logica Library, pages 145–212. Kluwer, 2003.
32. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. CUP, 2001.
33. R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In *ESOP*, pages 204–218, 2004.
34. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, , and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In W. Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.
35. M. P. Fiore and K. Honda. Recursive types in games: Axiomatics and process representation. In *LICS*, pages 345–356, 1998.
36. R. W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.
37. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
38. J.-Y. Girard. Linear logic. *TCS*, 50:1–102, 1987.
39. M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. LNCS 78, Springer Verlag, 78.
40. S. Graf and J. Sifakis. A modal characterization of observational congruence on finite terms of ccs. In *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, pages 222–234, London, UK, 1984. Springer-Verlag.
41. D. Harel, J. Tiuryn, and D. Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.
42. M. Hennessy and R. Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.
43. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
44. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
45. C. A. R. Hoare. Programs are predicates. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 141–155, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.
46. C. A. R. Hoare and H. Jifeng. *Unified Theories of Programming*. Prentice-Hall International, 1998.
47. C. A. R. Hoare and P. W. O’Hearn. Separation logic semantics for communicating processes. *Electr. Notes Theor. Comput. Sci.*, 212:3–25, 2008.
48. T. Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
49. T. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
50. G. J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
51. K. Honda. Composing Processes. In G. L. Steele, editor, *POPL’96*, pages 344–357. ACM Press, 1996.

52. K. Honda. Elementary Structures for Process Theory (1): Sets with Renaming. *MSCS*, pages 50–54, 2001.
53. K. Honda. From process logic to program logic. In *ICFP'04*, pages 163–174. ACM Press, 2004.
54. K. Honda, M. Berger, and N. Yoshida. Descriptive and relative completeness for logics for higher-order functions. In *ICALP'06*, volume 4052 of *LNCS*, pages 360–371, 2006.
55. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138, 1998.
56. K. Honda and N. Yoshida. On reduction-based process semantics. *TCS*, 151(2):437–486, 1995.
57. K. Honda and N. Yoshida. Game theoretic analysis of call-by-value computation. In *ICALP*, pages 225–236, 1997.
58. K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *Proc. PPDP'04, 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 191–202, 2004.
59. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. LICS'05*, pages 270–279, 2005. Full version is available at: www.dcs.qmul.ac.uk/~kohei/logics.
60. R. Hu, N. Yoshida, and K. Honda. Type-safe Communication in Java with Session Types. <http://www.doc.ic.ac.uk/~rh105/sessiondj.html>, March 2007.
61. J. M. E. Hyland and C. H. L. Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.
62. J. M. E. Hyland and C. H. L. Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.
63. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.
64. C. Jones. Constraining interference in an object-based design method. In *Proceedings of TAPSOFT'93*, LNCS 668, pages 136–150. Springer Verlag, 1993.
65. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
66. N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247, 2006.
67. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. *ACM TOPLAS*, 21(5):914–947, Sept. 1999.
68. D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.
69. J. Laird. A fully abstract game semantics of local exceptions. In *LICS*, pages 105–114, 2001.
70. K. G. Larsen. Proof systems for satisfiability in hennessy-milner logic with recursion. *Theor. Comput. Sci.*, 72(2&3):265–288, 1990.
71. K. R. M. Leino. Verifying object-oriented software: Lessons and challenges. In *TACAS*, page 2, 2007.
72. J. Longley and G. Plotkin. Logical full abstraction and pcf. In *Tbilisi Symposium on Logic, Language and Information*. CLSI, 1998.
73. G. McCusker. Games and full abstraction for fpc. In *LICS*, pages 174–183, 1996.
74. E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
75. D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, 2005.

76. R. Milner. Processes, a mathematical model of computing agents. In *Proc Logic Colloquium*, pages 157–174. North-Holland Pub Co, 1973.
77. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Berlin, 1980.
78. R. Milner. Lectures on a calculus for communicating systems. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 197–221, Berlin, 1985. Springer. Lecture Notes in Computer Science Vol. 197.
79. R. Milner. *Communication and Concurrency*. Prentics Hall, 1989.
80. R. Milner. Functions as processes. In *Proc. of ICALP'90*, volume 443 of *LNCS*, pages 167–180. Springer, 1990.
81. R. Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.
82. R. Milner. The polyadic π -calculus: A tutorial. In *Proceedings of the International Summer School on Logic Algebra of Specification*. Marktoberdorf, 1992.
83. R. Milner. Speech on receiving an Honorary Degree from the University of Bologna, ICALP'97, 1997.
84. R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
85. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1989.
86. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Info. & Comp.*, 100(1), 1992.
87. R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *TCS*, 114:149–171, 1993.
88. A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ads in hoare type theory. In *ESOP*, pages 189–204, 2007.
89. A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.
90. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
91. B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 1996.
92. A. M. Pitts. Existential Types: Logical Relations and Operational Equivalence. In *Proc. ICALP'98*, number 1443 in *LNCS*, 1998.
93. A. M. Pitts. Alpha-structural recursion and induction (extended abstract). In J. Hurd and T. Melham, editors, *Proc. TPHOLs 2005*, *LNCS*, to appear.
94. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *HOOTS'98*, CUP, pages 227–273, 1998.
95. J. H. Reppy. CML: A higher-order concurrent language. In *PLDI*, pages 293–305, 1991.
96. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. LICS'02*, 2002.
97. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.
98. D. Sangiorgi. The name discipline of uniform receptiveness. *Theor. Comput. Sci.*, 221(1-2):457–493, 1999.
99. D. Sangiorgi. Types, or: Where's the Difference Between CCS and pi? In *CONCUR*, volume 2421 of *LNCS*, pages 76–97, 2002.
100. D. Sangiorgi and R. Milner. The problem of “Weak Bisimulation up to”. In W. Cleveland, editor, *Proc. CONCUR '92*, volume 630 of *LNCS*, pages 32–46. Springer, 1992.
101. D. S. Scott. Domains for denotational semantics. In *ICALP*, pages 577–613, 1982.

102. M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. ICFP 2003*, pages 263–274, Aug. 2003.
103. A. Simpson. Sequent calculi for process verification: Hennessy-Milner logic for an arbitrary GSOS. *J. Log. Algebr. Program.*, 60-61:287–322, 2004.
104. J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: high-throughput stream programming in java. In *OOPSLA*, pages 211–228. ACM, 2007.
105. B. Steffen. Characteristic formulae. In *ICALP'89*, pages 723–732. Springer-Verlag, 1989.
106. B. Steffen and A. Ingolfsdottir. Characteristic formulae for processes with divergence. *Inf. Comput.*, 110(1):149–163, 1994.
107. C. Stirling. A complete compositional model proof system for a subset of CCS. In *ICALP*, LNCS, pages 475–486, 1985.
108. C. Stirling. Modal logics for communicating systems. *Theor. Comput. Sci.*, 49:311–347, 1987.
109. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
110. A. F. Tiu. Model checking for pi-calculus using proof search. In *CONCUR*, volume 3653 of *LNCS*, pages 36–50. Springer, 2005.
111. UNIFI. International Organization for Standardization ISO 20022 UNiversal Financial Industry message scheme. <http://www.iso20022.org>, 2002.
112. V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.
113. R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In J. C. M. Baeten and J. W. Klop, editors, *CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990.
114. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.
115. D. Walker. Objects in the pi-calculus. *Inf. Comput.*, 116(2):253–271, 1995.
116. P. Welch and F. Barnes. Communicating Mobile Processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *LNCS*, pages 175–210. Springer, 2005.
117. G. Winskel. *The formal semantics of programming languages*. MIT Press, 1993.
118. N. Yoshida. Graph types for monadic mobile processes. In *FSTTCS*, volume 1180 of *LNCS*, pages 371–386, 1996.
119. N. Yoshida, M. Berger, and K. Honda. Strong Normalisation in the π -Calculus. In *Proc. LICS'01*, pages 311–322. IEEE, 2001. The full version in *Journal of Inf. & Comp.*, 191 (2004) 145–202, Elsevier.
120. N. Yoshida, M. Berger, and K. Honda. Strong Normalisation in the π -Calculus. *Information and Computation*, 191(2004):145–202, 2004.
121. N. Yoshida, K. Honda, and M. Berger. Logical reasoning for higher-order functions with local state. In *FoSSaCS*, volume 4423 of *LNCS*, pages 361–377. Springer, 2007.
122. N. Yoshida, K. Honda, and M. Berger. Logical reasoning for higher-order functions with local state. *CoRR*, abs/0806.2448, 2008.
123. N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *ENTCS*, 171(4):73–93, 2007.
124. S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *CSFW*, pages 29–, 2003.

A Proofs for Section 7

A.1 Remaining Cases for Proposition 7.11

The case for Rec-ind is given in the main section. For other cases:

Case Conc. Suppose $\Gamma; E \Vdash P \blacktriangleright A$ and $\Delta; E \Vdash Q \blacktriangleright B$ such that $\Theta = \Gamma \odot \Delta$. Then we have

$$(P|Q)\xi\sigma \in \llbracket \Gamma \vdash A \rrbracket \xi \mid \llbracket \Delta \vdash B \rrbracket \xi \quad (\text{A.1.1})$$

noting the right-hand side is typed under Θ , hence done.

Case Res. Assume $\Gamma, u : \alpha; E \Vdash P \blacktriangleright A$ and $\Gamma \vdash (\nu u)P$ is typable. We infer:

$$(\nu u)P\xi\sigma \in (\nu u)\llbracket \Gamma \vdash A \rrbracket \xi \quad (\text{A.1.2})$$

$$= (\nu u)\llbracket \Gamma, x : \alpha \vdash A[x/u] \rrbracket (\xi, x \mapsto u) \quad (\text{A.1.3})$$

$$= \llbracket \Gamma \vdash \nu x.A[x/u] \rrbracket \xi \quad (\text{A.1.4})$$

as required.

Case Acc. By $a(k).P \xrightarrow{a(k)} P$ and because $P \in \llbracket A \rrbracket \xi$ by induction hypothesis we have $(a(k).P)\sigma \in \llbracket a(k) \rrbracket \llbracket A \rrbracket \xi$ hence done.

Case Req. Similar to Acc.

Case Send. Similar to Acc.

Case Rcv. Assume x in $k(x).P$ is typed as a channel name (the case when it is an atomic type is simpler). We show, assuming $E \Vdash P \blacktriangleright A$:

$$(k(x).P)\sigma \in \llbracket \forall x. \langle\langle kx \rangle\rangle A \rrbracket \xi \quad (\text{A.1.5})$$

We have:

$$\llbracket \forall x. \langle\langle kx \rangle\rangle A \rrbracket \xi = \bigcap_v \llbracket \Gamma, x : \alpha \vdash \langle\langle kx \rangle\rangle A \rrbracket (\xi, x \mapsto v) \quad (\text{A.1.6})$$

(The right hand side is equivalent to what is given in Definition 4.7 since a process belonging to such an interpretation can only have channels in $\Gamma\xi$). We now show the process $k(x).P$ satisfy this specification. First when v is inside Γ , surely we have

$$(k(x).P)\sigma \xrightarrow{kv} P[v/x] \in \llbracket A \rrbracket (\xi, x \mapsto v) \quad (\text{A.1.7})$$

On the other hand if v is fresh then we have, under $\Gamma\xi, v : \alpha$:

$$(k(x).P)\sigma \xrightarrow{kv} P[v/x] \in \llbracket A \rrbracket (\xi, x \mapsto v) \quad (\text{A.1.8})$$

(Note this transition corresponds to bound input if the process is typed under $\Gamma\xi$.) In both cases we have shown that the process satisfies the specification.

Case Bra, Sel. Similar to Acc.

Case Inact. Immediate.

A.2 Remaining Cases for Proposition 7.13

The minimality of formulae resulting from Conc and Res are by the precongruency of \sqsubseteq_{may} .

For prefix rules, we only show the case of Acc. Suppose $E \vdash_{may}^* P \blacktriangleright A$ and P is minimal in A . Now consider $E \vdash_{may}^* a(k).P \blacktriangleright \langle\langle a(k) \rangle\rangle A$. First this is clearly sound. Second assume $E \vdash_{may}^* Q \blacktriangleright \langle\langle a(k) \rangle\rangle A$. Note all visible traces of $a(k).P$ starts from $a(k).P \xrightarrow{a(k)} P$. Now Q has this initial transition $Q \xrightarrow{a(k)} Q'$ such that $E \models Q' \blacktriangleright A$ which, by induction hypothesis, satisfies $P \sqsubseteq_{may} Q'$. Thus any trace P has is also owned by Q' , hence as required.

Other prefix rules are essentially similar.

For Var, under the given conditions on σ and ξ , X under $\xi\sigma$ is certainly a minimal element of \underline{X} under ξ .

A.3 Remaining Cases for Proposition 7.15

We prove those cases other than Rec.

Case Acc. For Acc-Must, suppose $a(k).P \xrightarrow{a(k)} Q$. Then $Q \equiv P$. Since the satisfaction is closed under \approx we have $Q \in \llbracket A \rrbracket(\xi, \xi(E))$. Since there are no actions except this action the inaction clause is also immediate. For mix modality, we in addition have $\llbracket \langle\langle a(k) \rangle\rangle T \rrbracket$ which is obviously satisfies.

Case Req. Similar to Acc.

Case Send, Rcv, Bra, Sel. Similar to Acc.

Case Inact. The assertion is a single point property which is $\mathbf{0}$ itself up to \approx . (Note there are other agents which are Must-equivalent to $\mathbf{0}$: they are not in this assertion.)

A.4 Remaining Cases for Proposition 7.17

For maximality we prove those cases other than Rec.

Case Inact. The given assertion represents $\mathbf{0}$ and only that semantic point so all stated conditions are immediate.

Case Acc. We take this rule as a representative of the prefix rule. Assume $E \vdash_{must}^* P \blacktriangleright A$ satisfies the stated properties and consider $E \vdash_{must}^* a(k).P \blacktriangleright \llbracket a(k), \Gamma \rrbracket A$. Suppose Q satisfies $\llbracket a(k), \Gamma \rrbracket A$. Then either it does not have a visible transition or if it ever has one then it can only be $a(k)$. In the former case $Q \approx \mathbf{0}$ so it surely is smaller than $a(k).P$ w.r.t. \sqsubseteq_{must} . If the latter is the case then $Q \approx \xrightarrow{a(k)} Q'$ implies $Q' \models A$ for which we know $Q' \sqsubseteq_{must} P$. Hence each must test of Q can also be met by P , that is $Q \sqsubseteq_{must} a(k).P$, as required.

Other prefix rules are similar.

Case Conc. For maximality, suppose we have derived (omitting E uniformly) $\Gamma \odot \Delta \vdash_{must}^* P | Q \blacktriangleright A \circ B$ from $\Gamma \vdash_{must}^* P \blacktriangleright A$ and $\Gamma \vdash_{must}^* P \blacktriangleright A$ by this rule and that we have

$$\models R \blacktriangleright A \circ B. \quad (\text{A.4.1})$$

We first observe that A contains all free channel names and free channel name variables in Γ , by construction. Hence when we form $A \circ B$, it precisely prescribes the typing of individual processes. Now by (A.4.1) we know $R \approx S | T$ such that $\Gamma \models S \blacktriangleright A$ and $\Delta \models T \blacktriangleright B$. Hence by induction hypothesis and pre-congruency of \sqsubseteq_{must} we are done.

Case Res. We again use the pre-congruency of \sqsubseteq_{must} , as above.

Case Var. Immediate from the construction. ■

A.5 Proof of Lemma 7.28

We first make explicit inductive construction of well-guarded Must-modality formulae.

Definition A.1. The set \mathcal{W} of *well-guarded Must-modality formulae* is generated by the following induction, assuming typability at each step under an implicit typing.

- \top , the non-action predicate $\text{noact}(\Gamma)$ at the typing Γ , and equations on Boolean and integer values are in \mathcal{W} .
- $A \wedge B$ and $A \vee B$ are in \mathcal{W} if A and B are.
- $\llbracket \ell, \Gamma \rrbracket A$ (covering the whole of the implicit typing by Γ and ℓ) is in \mathcal{W} if A is, similarly for branching prefixes and value input prefixes.
- $(\nu X(\vec{x}).A)\langle \vec{e} \rangle$ is in \mathcal{W} if A is.
- $A \circ B$ is in \mathcal{W} if under any well-typed closing substitutions, A and B do not have any pair of dual occurrences of a common channel, up to unfoldings of ν -recursion.
- $\nu x.A[x/u]$ is in \mathcal{W} if A is.

We call a well-typed formula in \mathcal{W} a *well-guarded Must formula* for brevity.

Proposition A.2. *If $E, X : (\tilde{x})A \vdash_{must}^{**} P \blacktriangleright A$ is derived and if P is the recursion body of a well-guarded recursion, then A is a well-guarded Must formula.*

Proof. Immediate by construction, noting the condition for \circ comes from well-guardedness. ■

The following characterises transitions of well-guarded recursions and their finite approximants.

Definition A.3 (visible process). We say P is *visible* if, after any typed transitions, it never has a τ -action except those by the evaluation of conditionals, and, moreover, $C[\cdot]$ does not contain two active prefixes with the same channel as their subjects, after any typed transitions, cf. Definition 7.2.

Note that if P is visible, then (because the reduction due to a conditional evaluation is deterministic) we can cut off all τ -actions from its transitions. Thus, without loss of generality, we can assume:

Convention A.4. Hereafter we always consider that a visible process only has strong transitions.

We now show the satisfaction of a well-guarded Must formula by a visible process is determined by the satisfaction of the same formula by its approximants. Below and henceforth we often treat a visible process as its induced strong synchronisation tree. We first generalise the notion of finite unfoldings of a recursion with contexts.

Definition A.5. Let P be of the form $C[R]$ where C is a context whose whole is not under a recursion and R itself is a recursion. Suppose $\{R_i\}$ are the standard finite foldings of R . Then we call $\{C[R_i]\}$ *finite approximants of P w.r.t. $C[\cdot]$* or, often simply, *finite approximants of P* .

Note that when P itself a recursion, then it has only one kind of finite approximants, its finite unfoldings.

Definition A.6 (strong admissibility). A property p is *strongly admissible* if for any approximants $\{P_i\}$ of visible P , if they satisfy p , then P also satisfies p .

We also use the following lemma.

Lemma A.7. *Suppose A is a well-guarded Must characteristic formula under Γ , and u does not occur in A . Then with x fresh, we have $\forall x.A[x/u] \equiv A$ under Γ .*

Proof. Suppose $P \models A$ under Γ . Since u is fresh for A , it is also fresh for P , hence $(\nu u)P \approx P$, hence $P \models \nu x.A[x/u]$. Next suppose $P \models \nu x.A[x/u]$ under Γ . By definition $P \approx (\nu u)Q$ such that $Q \models A$. Since A is a well-guarded Must characteristic formula, A only specifies what actions are possible at each stage, including interleavings of actions (via \circ). Since $(\nu u)Q$ has no more actions than specified in A at each step (noting all actions at u by Q are truncated and Q cannot have an action which extrudes u), $(\nu u)Q$ surely satisfies A , hence $P \models A$. ■

We now show the main result of this subsection.

Proposition A.8. *Let A be a well-guarded Must characteristic formula and assume P is visible. Assume a family of approximants $\{P_i\}$ satisfies A . Then P satisfies A under any environment mapping an assertion variable to a strongly admissible property.*

Proof. We again use induction on the structure of A .

Base cases. \top , non-action predicates and first-order equations under valuations are immediate.

Conjunction and disjunction. Suppose A and B are well-guarded Must formulae. Suppose approximants of P satisfy $A \wedge B$. Then they satisfy A as well as B . By induction we are done. Similarly for disjunction.

Prefixes. Consider a well-guarded Must formula $B \stackrel{\text{def}}{=} \llbracket a(k), \Gamma \rrbracket A$. Suppose approximants of P satisfy B . Hence each P_i with $i \geq 1$ (which cannot be the inaction, since it contains an unfolding) can be written $a(k).Q_i$ with Q_i approximating Q and satisfying A . By induction we are done.

Maximal fixed point. Suppose A is a well-guarded Must formula and let $B \stackrel{\text{def}}{=} (\nu X(\tilde{x}).A)\langle \tilde{z} \rangle$. Suppose each P_i from approximants $\{P_i\}$ satisfies B . Hence each P_i satisfies each approximant to B . Since \top itself is strongly admissible, each approximant to B is strongly admissible. Hence P satisfies each approximant to B , i.e. by definition P satisfies B .

Parallel composition. Let $C \stackrel{\text{def}}{=} A \circ B$ with A and B being well-guarded Must formulae, and they are disjoint at subjects. Suppose approximants $\{P_i\}$ of P satisfy C . Take P_n for some $n \geq 2$ and suppose it satisfies C . By definition $P_n \approx Q'_n | R'_n$ s.t. Q'_n satisfies A and R'_n satisfies B . Again by disjointness we can write $P_n \equiv Q_n | R_n$ such that $Q_n \approx Q'_n$ and $R_n \approx R'_n$ (including the case either has no action). In this

way we know we have a hierarchy of processes:

$$\begin{aligned}
P_0 &\equiv Q_0|R_0, & \text{where } Q_0 \blacktriangleright A, & R_0 \blacktriangleright B \\
P_1 &\equiv Q_1|R_1, & \text{where } Q_1 \blacktriangleright A, & R_1 \blacktriangleright B \\
&\vdots \\
P_n &\equiv Q_n|R_n, & \text{where } Q_n \blacktriangleright A, & R_n \blacktriangleright B \\
&\vdots
\end{aligned}$$

Observe that, since a recursion is always prefixed, for each $i \geq 1$, we know either Q_i or R_i contains the i -th unfolding of the recursion. Suppose it is in Q_i . Then by the definition of the chain $\{P_i\}$, we can set $\{Q_i\}$ giving the chain of finite unfoldings under some context (with Q_0 containing or equal to the 0-th unfolding, the inaction), and each R_i being identical to each other, which we set R . That is, for each i we can set:

$$P_i \equiv Q_i|R \quad \text{where } Q_i \blacktriangleright A, \quad R \blacktriangleright B \quad (\text{A.5.1})$$

where $\{Q_i\}$ approximates Q . By induction Q satisfies A , hence P satisfies C .

Hiding. Let $B \stackrel{\text{def}}{=} \nu x.A[x/u]$ be a well-guarded Must formula. Note B under an environment is a collection of trees each of which is the result of cutting off some branches from one of the trees from A as well as inducing bound outputs. Suppose $\{P_i\}$ approximating P satisfy B . There are the following cases.

- (1) u does not occur in A .
- (2) u occurs in A and only as the subjects of Must modal actions.
- (3) u occurs in A as the object of a Must modal action, as well as possibly as the subjects/objects of other actions.

For (1), since each P_i satisfies B , by Lemma A.7, it also satisfies A . By induction P satisfies A . Again by Lemma A.7 P satisfies B .

For (2), B is satisfied by all and only synchronisation trees without u satisfying A . By assumption, P_n satisfies B . Since the hiding by u on A only cuts off those u -occurring subtrees and because a visible Must-characteristic formula only specifies *allowance* of actions inductively, we know P_i also satisfies A , hence by induction P satisfies A . Since $P \approx (\nu u)P$, P satisfies B .

Finally for (3), if P (hence $\{P_i\}$) does not have one of those bound output extruding u , as specified in A , then this reduces to (1) or (2). If P does have such a bound output, then, by visibility and because $\{P_i\}$ is an unfolding chain, this should start at least from P_1 . Take P_n such that $n \geq 1$. Since P_n satisfies B , there is one or more bound outputs in P_n extruding u conforming to B (hence

A). Observe that the specification of A follows a visible synchronisation tree by construction: in particular:

- If it allows two or more potential actions at different subjects, then it also allows their interleavings. And it never allows two or more potential actions at the same subject.
- The allowance of an initial action at some subject can be strictly mapped to, syntactically, a position under a specific prefix or if not at the top of the whole configuration.
- By well-guardedness, a prefix of the same subject can only occur in a sequence of nested prefixes (except in two different branches of a conditional), so that there is an outermost prefix with that subject, possibly as part of a recursion.
- Therefore we can check that A specifies, after say $\llbracket \bar{k}u, \Delta \rrbracket$, whether a subject u becomes active only after that action (if a prefix at u_i is exactly below such an output, we call it is in a *strict position*), or it is so already before that action (in which case we call the outermost prefix, *pre-positioned prefix*: if it is just under the corresponding bound output), though this difference however gets blurred by hiding at u , where actions at u become possible only after it is extruded.
- Two bound outputs extruding u as specified in A can be realised by P_n as two distinct bound outputs, as far as it conforms to the specification, and as far as there is Q_n such that $P_n \approx (\nu u)Q_n$ satisfying A .

Taking these points into consideration, we construct Q_n such that $P_n \approx (\nu u)Q_n$ and $Q_n \blacktriangleright A$ through the following steps:

- (Step 1) We pinpoint all prefixes of P_n which perform bound outputs extruding u (this can be deeply nested inside a recursion).
- (Step 2) Let $u_1..u_m$ ($m \geq 1$) be bound names extruded by P_n specified as u in B . Let take off their bindings from P_n , and set the result to be Q'_n .
- (Step 3) These u_i may occur initially as pre-positioned prefixes, occurring even before A allows it (thus violating A). If so we move this outermost pre-positioned prefix with subject u_i to the strict position (in the sense above, i.e. immediately under the output prefix which extrudes u_i), without changing the behaviour under its hiding. We then coalesce $u_1..u_m$ into u , and call the resulting process Q_n .

We now observe:

Claim. Q_n constructed as above satisfies $P_n \approx (\nu u)Q_n$ and $Q_n \blacktriangleright A$, and is well-guarded and visible.

PROOF: $(\nu u)Q_n$ has the synchronisation tree identical to the result of adding bindings to Q'_n , i.e. identical to the process P , hence $P_n \approx (\nu u)Q_n$ holds. Next, the synchronisation tree of Q_n conforms to A since, apart from turning bound outputs into free outputs, its transitions precisely mimic those of P_n , by the removal of all pre-positioned prefixes to the strict positions (note if A allows a pre-positioned prefix then it also allows the same prefix in the strict position). Finally Q_n is visible and well-guarded since, by conforming to A , no two subject occurrences of u can become active at the same time under any typed transitions. For uniform construction, observe that either there is a unique name, say u , which is extruded in P , or more than one, and asymptotically P_i will cover all of them (assuming a fixed valuation of variables). **(End of the proof of Claim)**

Let Q be the result of applying the same procedure to P , for which we again have $(\nu u)Q \approx P$ through the same argument. We now construct the chain:

$$\begin{aligned}
P_0 &\approx (\nu u)Q_0, & \text{where } Q_0 &\blacktriangleright A \\
P_1 &\approx (\nu u)Q_1, & \text{where } Q_1 &\blacktriangleright A \\
&\vdots \\
P_{n+1} &\approx (\nu u)Q_{n+1}, & \text{where } Q_{n+1} &\blacktriangleright A \\
&\vdots
\end{aligned}$$

where $\{Q_i\}$ are finite approximants of Q , since the structural change (including renaming and removal of prefixes) acts precisely in the same way for each Q_i as well as Q , so that the unfolding structures are preserved in $\{Q_i\}$ and Q from $\{P_i\}$ and P . By induction, Q satisfies A , hence P satisfies B , as required. ■

Remark A.9 (the use of well-guardedness). Semantically, the closure under admissibility for Must characteristic formulae comes from the nature of these formulae as “specifications for negative information”. For this reason, we believe that these proofs can be carried out with weak well-guardedness, though details are to be seen.

We have now reached Lemma 7.28.

Corollary A.10. *A well-guarded Must formula A is admissible.*

Proof. Immediate from Proposition A.8 and noting that all well-guarded recursions are visible. ■

B Axioms: Informal Presentation

B.1 Logical Axioms

Because of the introduction of \circ and \triangleright , it becomes important how logical laws can extract behavioural (modal) content from assertions which use these connectives. In this Appendix we explore some of the axioms which we may be able to use for enabling such endeavour. **The presentation is not systematic, and some of the axioms are presented informally** (a systematic, formal presentation is planned for a later version). However these axioms may elucidate the nature of the new connectives. One convention before our inquiry begins:

Convention B.1. In the following presentation of axioms, we assume all formulae are well-typed. In particular, when we have an axiom of the form $A \triangleright B$ (saying A entails B) or $A \equiv B$ (saying A and B are logically equivalent) we assume both A and B should be typable under the same typing environment.

B.1.1 Logical Axioms (1): par The following laws hold directly by the semantics of \circ .

$$B \triangleright A \triangleright (A \circ B) \quad (\text{B.1.1})$$

$$(A \triangleright B) \circ A \triangleright B \quad (\text{B.1.2})$$

$$A \triangleright (B \triangleright C) \equiv (A \circ B) \triangleright C \quad (\text{B.1.3})$$

$$A \circ B \equiv B \circ A \quad (\text{B.1.4})$$

$$(A \circ B) \circ C \equiv A \circ (B \circ C) \quad (\text{B.1.5})$$

$$A \circ \text{F} \equiv \text{F} \quad (\text{B.1.6})$$

$$A \circ \text{T} \equiv A \quad (\text{B.1.7})$$

It is notable that the final law does *not* say that $A \circ \text{T}$ can always be simplified into A : rather it says that *if* A is typable in the given environment then we can thus simplify. Similarly, assuming typedness of both sides, we have:

$$(A \triangleright B) \circ C \triangleright (A \triangleright (B \circ C)) \quad (\text{B.1.8})$$

which has a useful corollary:

$$(A \triangleright B) \circ (B \triangleright C) \triangleright A \triangleright C \quad (\text{B.1.9})$$

The next rules relate \circ with the standard logical connectives.

$$(A \wedge B) \circ C \triangleright (A \circ C) \wedge (B \circ C) \quad (\text{B.1.10})$$

$$(A \circ C) \vee (B \circ C) \triangleright (A \vee B) \circ C \quad (\text{B.1.11})$$

$$(A \circ C) \vee (B \circ C) \equiv (A \vee B) \circ C \quad (A \wedge B \equiv \text{F}) \quad (\text{B.1.12})$$

These have the exact analogues that use quantifiers:

$$(\forall x.A) \circ B \supset \forall x.(A \circ B) \quad (x \notin \text{fn}(B)) \quad (\text{B.1.13})$$

$$\exists x.(A \circ B) \supset (\exists x.A) \circ B \quad (x \notin \text{fn}(B)) \quad (\text{B.1.14})$$

In the following law assume $A(i) \wedge A(j) \equiv \text{F}$ whenever $i \neq j$.

$$\exists x.(A(x) \circ B) \equiv (\exists x.A(x)) \circ B \quad (x \notin \text{fn}(B)) \quad (\text{B.1.15})$$

The following law allows us to turn \circ into conjunction, assuming: (1) the typing of the whole formulae only use server types and no variables for channel types occur, and (2) $\text{fn}(A) \cap \text{fn}(B) = \emptyset$.

$$A \circ B \supset A \wedge B \quad (\text{B.1.16})$$

The typing condition makes it possible to decompose processes. Note A and B can only specify behaviour starting from replicated inputs.

B.1.2 Logical Axioms (2): new The ν -quantifier acts as expected with other operators when there is no name conflict.

$$\nu x.\langle\langle\rangle\rangle A \equiv \langle\langle\rangle\rangle \nu x.A \quad (\text{B.1.17})$$

$$\nu x.\langle\ell\rangle A \equiv \langle\ell\rangle \nu x.A \quad (x \notin \text{n}(\ell)) \quad (\text{B.1.18})$$

$$\nu x.\langle\langle\ell\rangle\rangle A \equiv \langle\langle\ell\rangle\rangle \nu x.A \quad (x \notin \text{n}(\ell)) \quad (\text{B.1.19})$$

$$\nu x.[] A \supset [] \nu x.A \quad (\text{B.1.20})$$

$$\nu x.[[\ell]] A \supset [[\ell]] \nu x.A \quad (x \notin \text{n}(\ell)) \quad (\text{B.1.21})$$

$$\nu x.[\ell] A \supset [\ell] \nu x.A \quad (x \notin \text{n}(\ell)) \quad (\text{B.1.22})$$

$$\nu x^\sigma.(A \circ B) \equiv (\nu x^\sigma.A^{x:\sigma}) \circ B \quad (x \notin \text{fn}(B)) \quad (\text{B.1.23})$$

$$\nu x^\sigma.(B \triangleright A) \equiv B \triangleright \nu x^\sigma.A^{x:\sigma} \quad (x \notin \text{fn}(B)) \quad (\text{B.1.24})$$

$$\nu x.\nu y.A \equiv \nu y.\nu x.A \quad (\text{B.1.25})$$

Another basic law is:

$$\nu x.A \supset A \quad (x \notin \text{fn}(A)) \quad (\text{B.1.26})$$

We shall prove later:

$$\nu x^\rho.A \equiv \exists x^\rho.A \quad (\rho \text{ server type}) \quad (\text{B.1.27})$$

It interacts with other connectives as follows.

$$\nu x.(A \wedge B) \supset (\nu x.A) \wedge (\nu x.B) \quad (\text{B.1.28})$$

$$\nu x.(A \vee B) \equiv (\nu x.A) \vee (\nu x.B) \quad (\text{B.1.29})$$

Finally the basic laws which relate this quantifier to the modal actions is:

$$\nu x. \langle\langle \bar{k}x \rangle\rangle A \equiv \langle\langle \bar{k}(a) \rangle\rangle A[a/x] \quad (a \text{ fresh}) \quad (\text{B.1.30})$$

$$\nu x. \langle \bar{k}x \rangle A \equiv \langle \bar{k}(a) \rangle A[a/x] \quad (a \text{ fresh}) \quad (\text{B.1.31})$$

$$\nu x. \llbracket \bar{k}x \rrbracket A \equiv \llbracket \bar{k}(a) \rrbracket A[a/x] \quad (a \text{ fresh}) \quad (\text{B.1.32})$$

$$\nu x. [\bar{k}x] A \equiv [\bar{k}(a)] A[a/x] \quad (a \text{ fresh}) \quad (\text{B.1.33})$$

B.2 Axioms for Strong Modalities

The study of strong modalities we get better ideas how axioms in our logical language relate to the preceding proof rules for parallel composition in the logics for the untyped π -calculus [10, 31] as well as for CCS [108]. Another reason for starting from axioms for strong modality is to have a clear tie with semantics and to assist understanding and justification for the axioms for weak transitions. In this spirit we later relate axioms for strong modalities to those for weak transitions.

B.2.1 May Axioms For axioms on strong modalities, we only consider May axioms and a couple of Must axioms. The logical axioms for \circ and ν are presented in the subsequent Section B.3. We start our inquiry from the axioms for strong May modality. The following three laws are most basic.

$$\langle \ell \rangle A \circ B \supset \langle \ell \rangle (A \circ B) \quad (\text{B.2.1})$$

$$\langle \ell \rangle A \circ \langle \bar{\ell} \rangle B \supset \langle\langle \ell \rangle\rangle (A \circ B) \quad (\text{B.2.2})$$

These hold for any types, though the first law strongly depends on types: for example if the overall typing for the subject of ℓ is a server type, and ℓ has a client type, then it cannot be transposed. For linear and replicated channel, we can strengthen the third axioms:

$$\langle \ell \rangle A \circ \langle \bar{\ell} \rangle B \equiv A \circ B \quad (\ell \text{ linear}) \quad (\text{B.2.3})$$

$$\langle \ell \rangle A \circ \langle \bar{\ell} \rangle B \equiv \langle \ell \rangle A \circ \nu \text{bn}(\ell).(A \circ B) \quad (\ell \text{ replicated}) \quad (\text{B.2.4})$$

Above “replicated” means the subject of ℓ has either a server type or a client type. In these cases we know the only possible interactions they can have are τ -actions.

The way compensating actions (hence τ -actions) are induced demands some care in the case of branching and selection. We need to explicitly stipulate the following law. In the following we assume $l_i \neq l_j$ for $i \neq j$.

$$\bigwedge_{i \in I} \langle k \triangleright l_i \rangle A_i \circ \bigvee_{i \in I' \subset I} \langle k \triangleleft l_i \rangle B_i \equiv \bigvee_{i \in I'} (A_i \circ B_i) \quad (\text{B.2.5})$$

This law cannot be reduced to individual compensation since a May action does not distribute over \wedge . In the following, we treat the l.h.s. as one pair of mutually dual actions (where we often have a single selection action).

One of the basic forms for May modal actions is to collect them by conjunction, $\bigwedge_i \langle l_i \rangle A_i$. Using this form, we can state a general law for May actions, which is essentially the expansion law in logical form. Assume below I and J are disjoint and $l_i \succ l_j$ denotes they are dual actions.

$$\bigwedge_{i \in I} \langle l_i \rangle A_i \circ \bigwedge_{j \in J} \langle l_j \rangle B_j \equiv \bigwedge_{l_i \succ l_j} \langle \langle \rangle \rangle \nu \text{bn}(l_i). (A_i \circ B_j) \wedge C \quad (\text{B.2.6})$$

where C is a term given by, with $I' \subset I$ and $J' \subset J$ which choose typable actions in the whole formula:

$$C \stackrel{\text{def}}{=} \bigwedge_{i \in I'} \langle l_i \rangle (A_i \circ (\bigwedge_{j \in J} \langle l_j \rangle B_j)) \wedge \bigwedge_{j \in J'} \langle l_j \rangle ((\bigwedge_{i \in I} A_i) \circ B_j) \quad (\text{B.2.7})$$

We also have the standard laws from the modal logic, such as $\langle \ell \rangle (A \wedge B) \supset \langle \ell \rangle A \wedge \langle \ell \rangle B$ and $\langle \ell \rangle (A \vee B) \equiv \langle \ell \rangle A \vee \langle \ell \rangle B$.

B.2.2 Must Axioms The following is the well-known law.

$$([\ell]A \wedge A') \circ ([\ell]B \wedge B') \supset [\ell]((A \circ ([\ell]B \wedge B')) \vee (([\ell]A \wedge A') \circ B)) \quad (\text{B.2.8})$$

A corollary of this law is:

$$[\ell]A \circ B \supset [\ell](A \circ B) \quad (B \supset [\ell]F) \quad (\text{B.2.9})$$

which is often useful in the presence of the no-action predicate $\text{noact}(\Gamma)$. Similarly (because types give the equivalent no-action information) we have:

$$([\ell]A) \circ B \supset [\ell](A \circ B) \quad (\ell \text{ linear}) \quad (\text{B.2.10})$$

The axioms for inducting $\llbracket \cdot \rrbracket$ is treated later.

B.2.3 Axioms for Server/Client Types Server/client types dictate that a resource (replicated process) is available without changing behaviour. This leads to the standard replication laws [82] and, in the present context, to the following laws. Below we assume a is server typed (τ) ¹.

$$\langle a(k) \rangle A \circ \langle \bar{a}(k) \rangle B \equiv (a(k))A \circ \langle \tau \rangle \nu k. (A \circ B) \quad (\text{B.2.11})$$

$$\langle a(k) \rangle A \circ \langle \ell \rangle B \equiv \langle a(k) \rangle A \circ \langle \ell \rangle (\nu a. (\langle a(k) \rangle A \circ B)) \quad (\text{B.2.12})$$

$$[a(k)]A \circ [\ell]B \equiv [a(k)]A \circ [\ell] (\nu a. ([a(k)]A \circ B)) \quad (\text{B.2.13})$$

B.2.4 Relating Strong Modality to Weak Modality We give basic laws. *All of the following laws are justified inside the weak semantics.* The first three are particularly useful. Let $\ell \neq \tau$ in these laws.

$$\langle \ell \rangle A \supset \langle \langle \ell \rangle \rangle A \quad (\text{B.2.14})$$

$$[[\ell]]A \supset [\ell]A \quad (\text{B.2.15})$$

$$\langle \langle \ell \rangle \rangle A \equiv \langle \rangle \langle \ell \rangle \langle \rangle A \quad (\text{B.2.16})$$

$$[[\ell]]A \equiv [[\ell]][\ell][[\ell]]A \quad (\text{B.2.17})$$

$$\langle \ell \rangle A \equiv [[\ell]](\langle \ell \rangle A \wedge [\ell]A) \quad (\text{B.2.18})$$

We also note, in the presence of strong $[\tau]$ -modality (defined in the standard way), we have, following [31]:

$$\langle \langle \ell \rangle \rangle A \equiv \mu X.(A \vee \langle \tau \rangle X) \quad (\text{B.2.19})$$

$$[[\ell]]A \equiv \nu X.(A \wedge [\tau]X) \quad (\text{B.2.20})$$

B.3 Axioms for Weak Modalities

B.3.1 May Axioms The following axioms clarify how weak modalities compose, unlike strong ones. The final axiom is the corollary of the preceding two axioms.

$$A \supset \langle \rangle A \quad (\text{B.3.1})$$

$$\langle \rangle \langle \rangle A \equiv \langle \rangle A \quad (\text{B.3.2})$$

$$\langle \rangle \langle \ell \rangle A \equiv \langle \ell \rangle A \quad (\text{B.3.3})$$

$$\langle \ell \rangle \langle \rangle A \equiv \langle \ell \rangle A \quad (\text{B.3.4})$$

$$\langle \rangle \langle \ell \rangle \langle \rangle A \equiv \langle \ell \rangle A \quad (\text{B.3.5})$$

We also have the exact analogue of some of the strong may axioms.

$$\langle \rangle A \circ B \supset \langle \rangle (A \circ B) \quad (\text{B.3.6})$$

$$\langle \ell \rangle A \circ B \supset \langle \ell \rangle (A \circ B) \quad (\text{B.3.7})$$

$$\langle \ell \rangle (A \triangleright B) \supset A \triangleright \langle \ell \rangle B \quad (\text{fn}(l) \cap \text{fn}(A) = \emptyset) \quad (\text{B.3.8})$$

We cannot have the weak analogue of the expansion law for the strong may actions since calculation of composite actions needs to probe deeply into formulae. However for strongly typed actions we have the following analogue, stated in the binary form for brevity. In the first line we assume $\text{sbj}(\ell) \neq \text{sbj}(\ell')$.

$$\langle \ell \rangle A \circ \langle \ell' \rangle B \equiv \langle \ell \rangle (A \circ \langle \ell' \rangle B) \wedge \langle \ell' \rangle (\langle \ell \rangle A \circ B) \quad (\ell, \ell' \text{ linear/client}) \quad (\text{B.3.9})$$

$$\langle \ell \rangle A \circ \langle \ell \rangle B \equiv \langle \ell \rangle (A \circ \langle \ell \rangle B) \wedge \langle \ell \rangle (\langle \ell \rangle A \circ B) \quad (\ell, \ell' \text{ linear/client}) \quad (\text{B.3.10})$$

$$\langle \ell \rangle A \circ \langle \bar{\ell} \rangle B \equiv \langle \tau \rangle (A \circ B) \quad (\ell \text{ linear}) \quad (\text{B.3.11})$$

Below assume a is server typed.

$$\langle\langle a(k) \rangle\rangle A \circ \langle\langle \bar{a}(k) \rangle\rangle B \equiv \langle\langle a(k) \rangle\rangle A \circ \langle\langle \nu k.(A \circ B) \rangle\rangle \quad (\text{B.3.12})$$

$$\langle\langle a(k) \rangle\rangle A \circ \langle\langle \ell \rangle\rangle B \equiv \langle\langle a(k) \rangle\rangle A \circ \langle\langle \ell \rangle\rangle (\nu a.(\langle\langle a(k) \rangle\rangle A \circ B)) \quad (\text{B.3.13})$$

We also list the general laws for May modality (which hold for both weak and strong modalities).

$$\langle\langle \rangle\rangle (A \wedge B) \supset \langle\langle \rangle\rangle A \wedge \langle\langle \rangle\rangle B \quad (\text{B.3.14})$$

$$\langle\langle \ell \rangle\rangle (A \wedge B) \supset \langle\langle \ell \rangle\rangle A \wedge \langle\langle \ell \rangle\rangle B \quad (\text{B.3.15})$$

$$\forall x. \langle\langle \rangle\rangle A \supset \langle\langle \rangle\rangle \forall x.A \quad (\text{B.3.16})$$

$$\forall x. \langle\langle \ell \rangle\rangle A \supset \langle\langle \ell \rangle\rangle \forall x.A \quad (\text{bn}(\ell) \cap \text{fn}(A) = \emptyset) \quad (\text{B.3.17})$$

$$\langle\langle \rangle\rangle (A \vee B) \equiv (\langle\langle \rangle\rangle A) \vee (\langle\langle \rangle\rangle B) \quad (\text{B.3.18})$$

$$\langle\langle \ell \rangle\rangle (A \vee B) \equiv (\langle\langle \ell \rangle\rangle A) \vee (\langle\langle \ell \rangle\rangle B) \quad (\text{B.3.19})$$

$$\exists x. \langle\langle \rangle\rangle A \equiv \langle\langle \rangle\rangle \exists x.A \quad (\text{B.3.20})$$

$$\exists x. \langle\langle \ell \rangle\rangle A \equiv \langle\langle \ell \rangle\rangle \exists x.A \quad (\text{bn}(\ell) \cap \text{fn}(A) = \emptyset) \quad (\text{B.3.21})$$

B.3.2 Must Axioms The following laws are symmetric to those of the weak May modalities.

$$\llbracket \rrbracket A \equiv A \quad (\text{B.3.22})$$

$$\llbracket \rrbracket \llbracket \rrbracket A \equiv \llbracket \rrbracket A \quad (\text{B.3.23})$$

$$\llbracket \rrbracket \llbracket \ell \rrbracket A \equiv \llbracket \ell \rrbracket A \quad (\text{B.3.24})$$

$$\llbracket \ell \rrbracket \llbracket \rrbracket A \equiv \llbracket \ell \rrbracket A \quad (\text{B.3.25})$$

$$\llbracket \rrbracket \llbracket \ell \rrbracket \llbracket \rrbracket A \equiv \llbracket \ell \rrbracket A \quad (\text{B.3.26})$$

We again have the analogue of the strong Must axioms. In the following laws we assume $\ell \neq \tau$. In the first law let $G = \llbracket \ell \rrbracket A \wedge \llbracket \rrbracket B$ and $G' = \llbracket \ell \rrbracket A' \wedge \llbracket \rrbracket B'$.

$$G \circ G' \supset \llbracket \ell \rrbracket ((A \circ G') \vee (G \circ A')) \quad (\text{B.3.27})$$

$$(\llbracket \ell \rrbracket A) \circ B \supset \llbracket \ell \rrbracket (A \circ B) \quad (B \supset \llbracket \ell \rrbracket F) \quad (\text{B.3.28})$$

$$\llbracket \ell \rrbracket A \circ \llbracket \rrbracket B \supset \llbracket \ell \rrbracket (A \circ B) \quad (\ell \text{ linear}) \quad (\text{B.3.29})$$

We also have:

$$\llbracket \ell \rrbracket A \circ \llbracket \rrbracket B \supset \llbracket \ell \rrbracket (A \circ B) \quad (\ell \text{ linear}) \quad (\text{B.3.30})$$

Observe that, by ℓ being linear and its being typable in the r.h.s., B cannot have ℓ as its capability.

For a server-typed:

$$\llbracket a(k), \Gamma \rrbracket A \circ \llbracket \bar{a}(k), \Delta \rrbracket B \equiv \llbracket a(k), \Gamma \rrbracket A \circ \llbracket \nu k. (\llbracket A \circ \rrbracket B) \rrbracket \quad (\text{B.3.31})$$

$$\llbracket a(k) \rrbracket A \circ \llbracket \ell \rrbracket B \equiv \llbracket a(k) \rrbracket A \circ \llbracket \ell \rrbracket (\nu a. (\llbracket a(k) \rrbracket A \circ B)) \quad (\text{B.3.32})$$

$$(\text{B.3.33})$$

The expansion law for Must modality is helped a great deal when we use linear actions. Assume the whole assertion is typed as $\Gamma \odot \delta, k : \perp$, where k is the subject channel of ℓ . Then we have:

$$\llbracket \ell, \Gamma \rrbracket A \circ \llbracket \bar{\ell}, \Delta \rrbracket B \supset (\llbracket \nu \text{bn}(\ell). (A \circ B) \rrbracket) \vee \text{noact}(\Gamma \odot \Delta) \quad (\ell \text{ linear}) \quad (\text{B.3.34})$$

If ℓ is non-linear, we need to replace $\llbracket \rrbracket$ with $\langle \langle \rrbracket$.

Finally we present a trial to do the Must expansion law in the weak setting for general actions, suggested by the existing action. Let:

$$G \stackrel{\text{def}}{=} \llbracket A \rrbracket \wedge \bigwedge_i \llbracket \ell_i \rrbracket B_i \wedge C \quad (\text{B.3.35})$$

$$G' \stackrel{\text{def}}{=} \llbracket A' \rrbracket \wedge \bigwedge_i \llbracket \bar{\ell}_i \rrbracket B'_i \wedge C' \quad (\text{B.3.36})$$

where C entails no other compensating actions are possible at G w.r.t. G' , similarly for C' , using the typing. Then we have:

$$G \circ G' \supset \llbracket ((G \circ A') \vee (G' \circ A) \vee (\bigvee_i B'_i \circ B_i)) \rrbracket \quad (\text{B.3.37})$$

For example let $G \stackrel{\text{def}}{=} \llbracket A \rrbracket \wedge \llbracket \ell \rrbracket B$ and $G' \stackrel{\text{def}}{=} \llbracket A' \rrbracket \wedge \llbracket \bar{\ell} \rrbracket B'$, and suppose $G \circ G'$ is typed under $k : \perp$ (and nothing else). Then we have:

$$G \circ G' \supset \llbracket ((G \circ A') \vee (G' \circ A) \vee (B \circ B')) \rrbracket \quad (\text{B.3.38})$$

B.3.3 Axioms for Mixed Modalities We shall use these axioms later in our examples (some of the important laws are also found in the axioms for server/client types discussed next). The importance of mixed axioms are because prefixes induce specifications with mixed (combined) modalities.

$$\langle \ell \rangle A \circ B \supset \langle \ell \rangle (A \circ B) \quad (\text{B.3.39})$$

$$\langle \ell \rangle A \circ \langle \bar{\ell} \rangle B \supset \langle \ell \rangle (A \wedge \langle \bar{\ell} \rangle B) \wedge \quad (\text{B.3.40})$$

$$\langle \ell \rangle \nu \text{bn}(l). (A \wedge B) \wedge \quad (\text{B.3.41})$$

$$\langle \bar{\ell} \rangle (\langle \ell \rangle A \circ B) \quad (\text{B.3.42})$$

$$G \circ G' \supset \langle \ell \rangle ((A \circ G') \vee (G \circ A')) \quad (\text{B.3.43})$$

where in the last line $G = \langle \ell \rangle A \wedge \langle \bar{\ell} \rangle B$ and $G' = \langle \ell \rangle A' \wedge \langle \bar{\ell} \rangle B'$. The following laws use types to obtain logical equivalence. In the first line we assume $\text{sbj}(\ell) \neq \text{sbj}(\ell')$.

$$\langle \ell \rangle A \circ \langle \ell' \rangle B \equiv \langle \ell \rangle (A \circ \langle \ell' \rangle B) \wedge \langle \ell' \rangle (\langle \ell \rangle A \circ B) \quad (\ell, \ell' \text{ linear/client}) \quad (\text{B.3.44})$$

$$\langle \ell \rangle A \circ \langle \ell \rangle B \equiv \langle \ell \rangle ((A \circ \langle \ell \rangle B) \vee (\langle \ell \rangle A \circ B)) \wedge \quad (\text{B.3.45})$$

$$\langle \ell \rangle (A \circ \langle \ell \rangle B) \wedge \quad (\text{B.3.46})$$

$$\langle \ell \rangle (\langle \ell \rangle A \circ B) \quad (\ell, \ell' \text{ linear/client}) \quad (\text{B.3.47})$$

$$\langle \ell \rangle A \circ \langle \bar{\ell} \rangle B \equiv \nu \text{bn}(l).(A \circ B) \quad (\ell \text{ linear}) \quad (\text{B.3.48})$$

The axiom (B.3.44) says if we can distinguish between two actions we can obtain the effect of parallel actions by interleaving and conjunction. The axiom (B.3.45) allows two actions to be the same by relaxing the conjunction to the disjunction while saying the two behaviours specified in the disjunction can surely take place. Since linear or client actions are stateless, this is enough to ensure the interleaving of actions. For actions at server-typed channels, we have, assuming a is server-typed below:

$$\langle a(k) \rangle A \circ \langle \bar{a}(k) \rangle B \equiv \langle a(k) \rangle A \circ \nu k.(A \circ B) \quad (\text{B.3.49})$$

$$\langle a(k) \rangle A \circ \langle \ell \rangle B \equiv \langle a(k) \rangle A \circ \langle \ell \rangle (\nu a.(\langle a(k) \rangle A \circ B)) \quad (\text{B.3.50})$$

We also have the following laws about the interplay with other connectives and quantifiers:

$$\langle \ell \rangle A \vee \langle \ell \rangle B \supset \langle \ell \rangle (A \vee B) \quad (\text{B.3.51})$$

$$\langle \ell \rangle (A \vee B) \equiv \langle \ell \rangle A \vee \langle \ell \rangle B \quad (A \wedge B \equiv \text{F}) \quad (\text{B.3.52})$$

$$\langle \ell \rangle (A \wedge B) \supset \langle \ell \rangle A \wedge \langle \ell \rangle B \quad (\text{B.3.53})$$

$$\langle \ell \rangle (A \triangleright B) \supset A \triangleright \langle \ell \rangle B \quad (\text{bn}(\ell) \cap \text{fn}(A) = \emptyset) \quad (\text{B.3.54})$$

The following law says that, even at a non-deterministic channel, a hidden one-to-one interaction has the same semantic effect as linear interaction.

$$\nu x.((\langle x(k), \emptyset \rangle A \wedge A') \circ (\langle \bar{x}(k), \emptyset \rangle B \wedge B')) \supset \nu x.((A \wedge A') \circ (B \wedge B')) \quad (\text{B.3.55})$$

Finally assume the following formula is typed with two linear types, $k : \perp, h : \perp, \Theta$ where $\Theta = \Gamma \odot \Delta$, and $\text{sbj}(l) = k, \text{sbj}(l') = h, h$ occurs in A in the same modality and k occurs in B in the same modality.

$$\langle l, \Gamma \rangle A \circ \langle l', \Delta \rangle B \equiv \text{noact}(k : \perp, h : \perp, \Theta) \quad (\text{B.3.56})$$

This law, which says that circular dependency in linear actions cannot be resolved, can be generalised to the n -party case.

B.3.4 Axioms for Server/Client Types We summarise the laws for server and client types. We assume a in each line is typed by a server type below.

$$\langle\langle a(k) \rangle\rangle A \circ \langle\langle \bar{a}(k) \rangle\rangle B \equiv \langle\langle a(k) \rangle\rangle A \circ \langle\langle \nu k.(A \circ B) \rangle\rangle \quad (\text{B.3.57})$$

$$\langle\langle a(k) \rangle\rangle A \circ \langle\langle \ell \rangle\rangle B \equiv \langle\langle a(k) \rangle\rangle A \circ \langle\langle \ell \rangle\rangle (\nu a.(\langle\langle a(k) \rangle\rangle A \circ B)) \quad (\text{B.3.58})$$

$$\llbracket a(k) \rrbracket A \circ \llbracket \bar{a}(k) \rrbracket B \equiv \llbracket a(k) \rrbracket A \circ \nu k.(A \circ B) \quad (\text{B.3.59})$$

$$\llbracket a(k) \rrbracket A \circ \llbracket \ell \rrbracket B \equiv \llbracket a(k) \rrbracket A \circ \llbracket \ell \rrbracket (\nu a.(\llbracket a(k) \rrbracket A \circ B)) \quad (\text{B.3.60})$$

$$\langle a(k) \rangle A \circ \langle \bar{a}(k) \rangle B \equiv \langle a(k) \rangle A \circ \nu k.(A \circ B) \quad (\text{B.3.61})$$

$$\langle a(k) \rangle A \circ \langle \ell \rangle B \equiv \langle a(k) \rangle A \circ \langle \ell \rangle (\nu a.(\langle a(k) \rangle A \circ B)) \quad (\text{B.3.62})$$

The following law, again assuming a is server-typed, shows significant impact determinism can have on weak transitions (compare this to the second to the last law above).

$$\langle a(k) \rangle A \circ \langle \bar{a}(k) \rangle B \equiv \langle a(k) \rangle A \circ \nu k.(A \circ B) \quad (\text{B.3.63})$$

Note we do not put any actions in front of the residual $\nu k.(A \circ B)$: this is possible because this reduction is semantically neutral, just as the case of linear weak interaction with mixed modality we saw before.

B.3.5 Miscellany Other laws include the co-variance laws for modal actions, \circ and ν (for example if $A \supset A'$ then $A \circ B \supset A' \circ B$ as far as types match), and the co-variance of the conclusion part and contra-variance of the assumption part in the \supset .

B.3.6 Axioms for Fixed Points The following laws are standard.

$$(\mu X(\tilde{x}).A)\langle \tilde{e} \rangle \equiv A[\mu X(\tilde{x}).A/X][\tilde{e}/\tilde{x}] \quad (\text{B.3.64})$$

$$\exists n.(\mu^n X(\tilde{x}).A)\langle \tilde{e} \rangle \supset (\mu X(\tilde{x}).A)\langle \tilde{e} \rangle \quad (\text{B.3.65})$$

Dually:

$$(\nu X(\tilde{x}).A)\langle \tilde{e} \rangle \equiv A[\nu X(\tilde{x}).A/X][\tilde{e}/\tilde{x}] \quad (\text{B.3.66})$$

$$(\nu X(\tilde{x}).A)\langle \tilde{e} \rangle \supset \forall n.(\nu^n X(\tilde{x}).A)\langle \tilde{e} \rangle \quad (\text{B.3.67})$$

Another axioms of interest are:

$$(\mu X(\tilde{x}_1 y \tilde{x}_2).A)\langle \tilde{e}_1 g \tilde{e}_2 \rangle \equiv (\mu X(\tilde{x}_1 \tilde{x}_2).A)\langle \tilde{e}_1 \tilde{e}_2 \rangle \quad (y \notin \text{fn}(A)) \quad (\text{B.3.68})$$

$$(\nu X(\tilde{x}_1 y \tilde{x}_2).A)\langle \tilde{e}_1 g \tilde{e}_2 \rangle \equiv (\nu X(\tilde{x}_1 \tilde{x}_2).A)\langle \tilde{e}_1 \tilde{e}_2 \rangle \quad (y \notin \text{fn}(A)) \quad (\text{B.3.69})$$

The following law is used to combine two or more states. First assume:

$$(A \circ B \supset C[X\langle\tilde{e}\rangle \circ Y\langle\tilde{g}\rangle]_i) \quad (\text{B.3.70})$$

is valid. Then we have:

$$(\forall X(\tilde{x}).A)\langle\tilde{e}\rangle \circ (\forall Y(\tilde{y}).B)\langle\tilde{g}\rangle \supset (\forall Z(\tilde{x}\tilde{y}).C[Z\langle\tilde{e}\tilde{g}\rangle]_i)\langle\tilde{e}\tilde{g}\rangle \quad (\text{B.3.71})$$

where $C[X\langle\tilde{e}\rangle_i \circ Y\langle\tilde{g}\rangle_i]_{i \in I}$ denotes a formula with multiple holes indexed by I , assuming all occurrences of X and Y are thus exhausted. This axiom combines the states of X and Y into one.

It is notable that many of the axioms for the weak modality can be validated through the usage of strong axioms with the help of linking axioms. We need further study on the status of these different kinds of axioms in terms of their inferential power and their semantic status.