Process Logic and Duality
Part I: Sequential Processes

Kohei Honda

Queen Mary, London

Abstract. We present typed process logics for the π-calculus with linear/affine type disciplines. Built on the preceding studies on logics for programs and processes, simple systems of assertions are developed, capturing the classes of behaviours ranging from purely functional interactions to those with destructive update, local state and genericity. A central feature of the logic is representation of the environments' behaviour as the dual of those of processes in assertions, which is crucial for obtaining compositional proof systems. This paper develops typed process logics starting from purely functional behaviours and treating increasingly complex ones, and illustrate their usage by deriving program logics for higher-order languages. The embedding of the proof rules in the derived logics into the process logics gives a simple proof of the soundness of the former. Some of the derived logics correspond to known program logics, including Hoare logic for imperative programs.

Table of Contents

List of Figures

1 Introduction

Program Logics offer abstraction of programs' behaviours centring on logical predicates on them, combined with proof systems for deriving valid judgements. They are useful both for the design of programs on a rigorous basis (i.e. specification) and analysis of existing programs (i.e. verification). In fact, these two aspects are most effectively integrated in real engineering practice, where code or design in production, annotated with predicates which express crucial safety properties, is subjected to logical reasoning and the result is reflected onto the process of design and coding. This reasoning part may even be half-mechanised, so that programmers can get feedbacks quickly and reliably. For all these purposes the proof system may as well be compositional (i.e. rules are built following the syntactic structure of the language) since, in this way, we can reason about a larger program based on properties derived for its constituting parts. Some logics are mathematically general for a given class of programs in that all meaningful observable properties are precisely specifiable (and, up to derivability of valid judgement in the associated domain, relatively provable) in the system. Since the program logics of this kind can express properties of programs in a general and rigorous fashion, and because the associated proof system offers a fundamental articulation of semantics of language constructs and their interplay, many engineering activities ranging from static analyses to program testing increasingly use program logics as their foundation.

One of the well-known instances of such logics, which in effect initiated the whole field of compositional program logics, is the specification logic by Tony Hoare [45], Hoare Logic, developed on the basis of earlier work by Floyd [36] and Nauer [92]. Hoare Logic, originally presented for a simple imperative programming language, is powerful from an engineering viewpoint: formulae and rules are simple and intuitive, capturing the essence of dynamics of imperative programs concisely and elegantly; and is satisfying from a theoretical viewpoint; not only the proof system is both sound and (relatively) complete with respect to a naturally defined model, but each of the rules has a decisive form which follows the principle of finding a weakest precondition for a desired conclusion. There have been a vast body of theoretical and practical studies starting from Hoare's work (see various surveys including e.g. [8, 24]). The maturity of the theoretical understanding and practical techniques is now reaching the stage where one can develop consistent proof rules for reasoning about non-trivial fragments of real-world programming languages [64, 94, 100].

Compositional program logics make it possible to reason about observable properties of software starting from their constituting parts. A piece of software, in principle and increasingly in practice, can consist of programs written in different languages. Even if a program is written in a single language, when it runs under an operating system, its properties should be considered in combination with those of the operating system, which may be written in a different language. Another instantiation of the same problem is when a high-level language calls native code (many libraries of a high-level language are implemented in this way). In fact, almost all kinds of software nowadays explicitly or implicitly rely on the functionalities of other software. A Java program relies on its APIs which are implemented using native code and OS libraries, an OS library uses kernel calls, and even a kernel relies on, for example, the proper working of the network for carrying out its essential functionalities. Thus reasoning about behaviour of software across language boundaries, but still on a rigorous logical basis, is fundamental for guaranteeing the safe behaviour of software. In spite of many studies on Hoare Logic and its derivatives, it seems this topic has not been investigated as extensively as the subject may deserve.

We present in this paper one possible approach to this problem domain. The idea is to develop compositional logics for a tiny formalism of interaction, and use it to derive, analyse and mediate, via encoding of language constructs in it, logics for more complex programming languages. This tiny formalism — which is a minimal core of a process calculus, the π-calculus, introduced by Milner, Parrow and Walker [84] — has a simple operation, *communication of names*, and a small set of algebraic operators, including *parallel composition*. The π-calculus is a calculus of concurrency, developed along the line of such process calculi as CCS [79], CSP [48] and ACP [12]. But, as the originators noted [84], it has great expressive power to represent diverse forms of computation, and as such may as well be regarded as a calculus for describing software behaviour in general, including sequential and functional computation [82]. In fact, if we constrain the composition of processes using a class of type annotations (which come from Linear Logic [39, 71], Game Semantics [6, 35, 58, 62, 93] and study of types for the π-calculus [51, 68, 81, 97, 111, 113]), we can embed various programming languages into name passing processes without losing essential semantic properties [13, 14, 55, 59, 60, 116, 117]. If, therefore, we can find a suitable notion of logic for typed name passing processes together with its sound compositional proof rules, we may as well be able to derive logics for different programming languages via encoding; and, in turn, the typed π-calculus would serve as a common stratum of these program logics, allowing the combined reasoning for multiple languages.

Let us further expand on the status of typed processes in our enterprise of developing a basis of language-independent reasoning on software behaviour. As we already noted, the typed π-calculus can map programs in different programming languages to a common mathematical domain, i.e. the universe of typed name passing processes, thus assigning a generic notion of software behaviour to them. This universe of behaviour is inherently typed, in the sense that the way of constructing behaviours from other behaviours is constrained by types. This constraint by types form an essential part of the notion of behaviour, since two behaviours may either be identical or distinct depending on what other behaviours can interact with them. Take two simple examples:

$$P_1 \equiv x := 1; x := 2 \qquad\qquad P_2 \equiv x := 2$$

Do P_1 and P_2 own identical behaviour, or are they distinct? The answer to this question depends on algebra of programs we consider. If we only consider **sequential composition** as a way of composing programs, then they would be safely considered to be the same "behaviour", since, in that case, intermediate results are never observed by other programs. But if we allow composition of programs on parallel, the change from 1 to 2 in x can be observed and can affect the interacting program (or environment), hence P_1 and P_2 should be regarded as semantically distinct — they own different behaviours. Thus what sorts of composition is being considered is important, and this is precisely what type structures offer us — the moulds by which we can shape behaviour. In fact, one of the distinguishing features of the π-calculus is that it offers very fine grains of software dynamics, so that almost any software behaviour can be constructed out of these grains, as far as we constrain their shape by suitable moulds (types).

Thus our process logics start from typed processes. Following Hennessy-Milner Logic [43], a basic modal logic for nondeterministic processes, assertions in the logic centre on communication, describing reciprocal interaction a process may have with its environment. Types offer fundamental organising principles for structuring assertions: among others, they lead to a form of logical description of behaviours where behaviour of an environment is represented as the dual of that of a process in assertions, which is crucial for obtaining compositional proof rules. The resulting logical description precisely captures semantics of typed processes, which is then reflected onto different programming languages via their embedding in the typed π-calculus. The present paper illustrates these key technical elements starting from the simplest setting, functional sequential processes and a basic call-by-value higher-order language, reaching classes of more complex sequential behaviours and different kinds of programming languages (including call-by-name, advanced type structures, data structure with destructive update, and global and local state). Among others we shall report how the original Hoare Logic is recoverable from a stateful version of the process logic, suggesting fruitful connections to the existing work on program logics.

The nature of the articulation of computational behaviours used for developing process/program logics in the present study starts from the class of behaviours that represent pure higher-order functions, and adds notions of state on its basis. A central reason for taking this approach is because the class of "function-like behaviour" in processes has intrinsic significance for logical articulation in that it is the only known class which exhibits strong polarisation in the sense of [60]: flow of information at each channel is either for input or for output, but never for both. This strict polarisation has a fundamental effect on its logical articulation, on whose basis stateful computation which exhibits mixed polarities, can be treated with a clean perspective. Further the need to provide clean logical articulation on the basis of strictly polarised behaviours necessitates distinction between two forms of stateful computation, called *open* and *local*, where the latter is stratified on the basis of the former, just as the former is stratified on the basis of purely functional behaviours. This incremental nature of our approach, which extends to basic classes of concurrent computation, is the key to tractable logical articulation of behaviours which can, among others, offer uniform compositional reasoning principles for complex data structures with higher-order, sharing and other features. No stratification of behaviours of the same level of generality has been known so far, which is the main reason why we take the present approach.

From a practical viewpoint, a basic reservation to our approach would be that its extensive use of types may result in difficulties in its practical adaptation. Examining the effects of close interplays between types and logics is one of the significant future issues. Further, because of dependency on types, the framework may in fact lose uniformity, in spite of our initial aim: the framework may not offer one unifying account, but fragmented ones, of program logics, which may become worse if concurrency and other features are treated. A possible advocacy of our approach would be that, at least in the sequential setting, each successively complex logic is in fact a conservative extension of the preceding simpler one — a more complex logic allows valid formulae for simpler behaviours precisely when the simpler logic do so. From a practical viewpoint, it remains to be examined how the interplays between types and logics can affect the use of the presented logics, in particular when we need to combine distinct typed behaviours in specification and in reasoning.

Concrete technical ideas of the typed process logic are outlined in Section 2.

1.1 Possible Technical Contributions

Apart from the general direction to use the typed logics for the π-calculus as a unifying basis of program logics, the following points may be counted among technical novelties of the present work.

1. The finding that duality-based types (originating in the foregoing studies on types and semantics of interaction [6, 39, 51, 62, 68, 81, 93, 97, 113]) leads to a highly structured form of Hennessy-Milner logic in the π-calculus, yielding, when combined with representation of the environment in assertions, simple compositional proof rules.
2. The use of fully abstract embedding of programming languages into typed π-calculi to derive, analyse and verify program logics for these languages. This extends various embedding results of programming languages in the π-calculus [13, 80, 112, 116] to a logical setting.
3. As an outcome of 1 and 2, a systematic technique to use names and operations on them for reasoning about programs and data structure on a uniform basis, including, for example, higher-order procedures, two forms of polymorphism, computation with local state, as well as data structures with destructive update.

Regarding the third point, it may be noted that uniform treatment of higher-order features of programming languages in compositional program logics, including higher-order procedures (without or with side effects) and methods in objects, may not have been known so far, in spite of extensive studies of Hoare-like logics for imperative programming languages, cf. [49]. The same would be true for a uniform treatment of diverse data structures including procedural data types, records, labelled unions, various user-defined data types, generic (polymorphic) data types, and shared data structures with destructive update. As far as we know, the presented theory offers the first framework to treat these features on a uniform basis. This is made possible by the underlying process logic where these complex structures are given an exact logical analysis via their decomposition into fine-grained dynamics and algebra of name passing processes.

1.2 Related work

As should be clear from the preceding discussions, the present work owes much to the three strands of research on semantics of computation, namely, compositional program logics, theories of processes, and types for functions and interaction. In the following we discuss those work which are closely related to the present study, categorised into these three strands. They are not exhaustive: any omission is due to the limited knowledge of the present author and (if one may add some excuse) the vastness and richness of the results from these fields.

(1) Program Logics. Among extensions of Hoare's axiomatic method, the proof rules for procedures studied by Hoare and Clint [23, 46] predict the logical treatment of replicated processes in the present work. Kowaltowsky [69] proposes a way to use names to express return values of procedures. More recently, Abadi and Leino [3] proposes a framework in which names of objects and operations on them are used as essential elements of a program logic for an object-oriented language. In this context, the present work extends the usage of names in these works to a general basis of specifications.

Jones [67] demonstrated that not only pre/post conditions but also an explicit assertion on the two-way constraints on a program and its environment is effective for reasoning about shared variable concurrency (a related idea is found in [37]). We owe to Jones' work the idea of representation of the environments' behaviour in assertions in the proposed framework, as recorded in the shape of the sequent in the present logic. It is notable that this framework is effectively used in the present theory ranging from purely functional behaviours to those with global and local state.

Reynolds, O'Hearn, Bornat and their colleagues [16, 95, 104, 105] present extensions of Hoare Logic with an aim to offer an effective reasoning method for low-level operations, including pointers, memory allocation/deallocation, and garbage collection. A central idea of their approach is to explicitly assume a dynamically allocated region of memory cells in the universe of discourse, and reason about them based on a conjunction which at the same time implies disjointness of such regions. The aim of the present logic differs in that it tries to capture general software behaviour as articulated by, among others, high-level programming constructs. As such, it offers a uniform way to reason about diverse data structures, including polymorphic data types and data structures with destructive update and sharing. The reasoning methods for low-level operations may as well be integrated with those for general programming constructs since, in real programming, the former arises combined with the latter. But the low-level constructs do have unique features, such as address arithmetic, which need specific treatment. From this and other viewpoints, it would be an interesting subject of study to integrate ideas cultivated through studies by Reynolds, O'Hearn and others for the reasoning on low-level constructs with the present framework (see Remark 11.7 in §11.6 for further discussions).

In a tradition different from Hoare Logic, equational logics for the λ-calculi have been studied since the classical work by Curry [26] and Church [21]. LCF [33] augments the standard equational theory of the λ-calculus with Scott's fixed point induction. Other prominent logics along this line include those for polymorphic λ-calculi [2, 99]. The program logics for higher-order functions derived from process logics differ in that their assertion describes behavioural properties of programs rather than equates them, allowing specifications with arbitrary degrees of precision as well as smoothly extending to non-functional behaviour. It should be however noted that, for calculating validity in proof rules in the present logics, it sometimes becomes necessary to make resort to semantic arguments on the target class of behaviours. This suggests possible fruitful interplay between the presented framework, on the one hand, and the reasoning principles as developed in, and extending, [2, 33, 99] (cf. [1, 14, 74, 75, 114]) on the other.

A work on program logics which is also based on Scott's theory of domains and which is closely related with intersection type disciplines [9, 30], is Abramsky's domain logic [4], where formulae describe properties of programs as those of their image in Scott domains, using the correspondence between domains and (logical presentation of) their topologies. Both Abramsky's approach and ours use a typed universe of behaviour for constructing generic program logics, even though methodologies are quite different. Abramsky's approach starts from domains and their algebra as a fundamental element in articulating computation behaviours, while ours starts from the dynamics and algebras of the π-calculus, enriched with type structures. While domains do offer clean and rich notions of algebras for a wide class of computational behaviours, the representation using domains may not extend easily to several significant classes of behaviours such as polymorphism and local state. Name passing processes are much more versatile in representing computation, even though capturing specific classes of behaviours and their algebra from this large universe, which is necessary for clean logical articulation, can be non-trivial. Relatedly, Abramsky's domain logic considers certain upper-closed subsets of domains (or equivalently open sets in the corresponding topologies) as specifiable properties, as well as limiting usable logical connectives. This restriction has the merit in that it automatically enables the reading of a domain as a logical theory. At the same time, the resulting logics can be practically too restrictive, in the sense that they may not allow us to directly describe some of the significant computational situations, such as partial correctness of sequential programs. In the approach taken in the present paper, essentially the whole (finitely describable) properties of programs are intended to be specifiable, combined with the full use of logical connectives. A technically interesting problem is how the restrictions of properties along the line of those given in the domain logic would become significant in the present framework. (Some discussions on this point are given in Sections 4 and 5.)

Dynamic Logic, introduced by Pratt [101], allows programs to appear as part of formulae, allowing detailed specifications on intensional properties. The present work gives formulae as information separate from a program itself, focussing on observational properties of programs (which follow the traditions of both Hoare Logic and Hennessy-Milner Logic). An interesting point is that the framework allows us to incorporate programs in formulae in a way different from Dynamic Logic: instead of having a program text itself in a formula, we have its interface points (names) and behaviours specified in a formula, leading to a rich specification framework (for example, it naturally offers explicit treatment of termination, which is not treated in the standard Hoare Logic). It remains to be seen how these two approaches exactly compare, in both practical and theoretical terms.

(2) Theories of Processes. In [76], Milner put forward the idea that semantics of interacting agents can be the basis of a theory of general computational behaviour. While he initially employed a domain-theoretic method, this idea later led to the first process calculus, CCS [78] and its behavioural theories, and, one decade later, to the π-calculus [84]. It is also notable that Hoare's early exposition of CSP [47] observed that communication primitives offer a unifying framework for giving the semantics of many programming constructs such as input/output and jumps that are difficult to treat otherwise. The theories and applications cultivated through the studies on process calculi and algebras form the basis of the present work, including the theories of behavioural equivalences, consistent syntactic treatment of various process operators, and logics for processes.

The embeddings of various computational calculi and programming languages in the π-calculus have been studied by many researchers, starting from Milner [80] (for higher-order functions) and Walker [112] (for object-oriented programs). The present work uses this embeddability for reflecting process logics onto program logics. A key technical aspect of this reflection is that it preserves and reflects the semantics of the logic: this guarantees, among others, the soundness of the derived program logic is automatically ensured by that of the original process logic (a related property, albeit in a purely semantic setting, is studied in the context of LCF-like logics under the terminology of *logical full abstraction* by Longley and Plotkin [72]). This semantic preservation crucially relies on the so-called definability property of the embedding, which is the idea originating in denotational semantics [77, 98].

As already noted, Hennessy-Milner Logic [43], a basic modal logic for nondeterministic processes, is one of the starting points of the present work. The logical articulation given by Hennessy-Milner logic is based on communication between processes. The presentation of properties of processes in Hennessy-Milner logic is based on decomposition of behaviours into all possible branches of interactions processes may be engaged in. The logical articulations of behaviours given in the presented process logics, hence in the derived program logics, are based on condensed representation of Hennessy-Milner logic, made possible by types, as concretely illustrated in Section 2. For capturing logical articulation of computational behaviours, the generality of logical representation in Hennessy-

Milner logics is a fundamental technical underpinning, offering a verbose but general representation of behaviours, which would become more prominent as our work moves beyond sequential computation. Hennessy-Milner logic enjoys a sound and complete equational characterisation for bisimilarity; the present logic enjoys the same equational characterisation with respect to the typed contextual congruence. Existing studies on logics for the π-calculus include [85] (for characterising different bisimilarities using Hennessy-Milner logics) and [17] (for capturing spatial structures of processes). Both of these studies are done in the untyped setting: the present work demonstrates that the use of types can lead to compositional process logics in which compositional program logics are precisely embeddable. It is interesting to note that compositional proof rules for the Hennessy-Milner logic of the untyped π-calculus becomes quite complex, as detailed by Dam [29]. This poses an interesting question: does the present framework — explicit representation of the environment — also help in the untyped (or weakly typed) setting for making a simple compositional logic? The same question may be asked for different classes of typed process behaviours than treated in the present note.

The embeddability of a program logic in a process logic is first demonstrated by Milner [79], which presents a sound embedding of formulae of a basic specification logic for shared variable concurrency in formulae in Hennessy-Milner Logic for CCS. His result suggested the present embedding results of program logics in process logics. In comparison, the present work has shown that, by the use of types, we can obtain a precise embeddability of compositional program logics in compositional process logics, not only in terms of validity of formulae, as done in [79], but also in terms of proof rules.

(3) Types for Functions and Interactions. The study of types in the modern sense started from mathematical logics, as found in the formalisation of mathematics by Russell and Whitehead [102]. In programming languages, the use of types comes from the need to identify, at the time of compilation, memory space to be allocated; later its benefit to structure and stratify programs and data types were gradually understood. A more formal study of types in programming languages [19, 96] started with the use of typed λ-calculi as the basis of theories of programming language, whose origins can be found in the early work by Landin [70] as well as Scott-Strachey's denotational semantics [108] (which, in effect, models programs as certain classes of typed higher-order functions). Typed λ-calculi are also closely related with proof theory by so-called Curry-Howard correspondence [27].

ML [83], developed by Milner and his colleagues (originally a meta-language for implementing LCF [33]), positions types as a central organising principle of programming in its language design, combining higher-order functions with imperative features, and has a prominent extension/implementation such as CAML [18]. Through the studies of ML, Haskell [42] and other functional programming languages, researchers found that strong typing is effective for high-level programming without compromising expressiveness or efficiency, while ensuring type safety. The obtained insight has been carried into widely used higher-level programming languages such as Java [65] and C$\sharp$[89]; and into the low-level code as studied by Morrisett and others [40, 87]. In the latter context, [87] has shown that even assemblers can be typed to ensure the basic type-safety; while, in [40, 88], a strong type discipline is given to an expressive low-level programming language, benefiting from the safety and clarity which types usually offer for high-level programming.

In the present study, types play the rôle of a fundamental stratum in logical specifications, where operators and judgements used in the logic (other than standard logical connectives) abstract algebra and dynamics of the target typed processes. Types in these processes in turn decide a certain class of behaviours, such as sequential computation. While types for processes in this setting differ from those for high-level programming languages in granularity (for example, even untyped λ-calculi become typed processes), there is a basic similarity, in that types are used as key organising principles of software behaviour.

The fine-grained types for processes on whose basis the present theory is developed come from, on the one hand, the notion of types for the π-calculus known as *sorting* [81] and its refinements [51, 68, 97, 113]; and, on the other, decomposition of types for higher-order functions into interaction originating in Linear Logic [39] (introduced in the context of studies of the correspondence between typed λ-calculi and proof theories noted before), which later led to type structures in game semantics [6, 62, 93]. Originally Linear Logic was conceived as both algebraic/semantic/operational decomposition of theories of proofs in constructive logics. It is interesting to note that typed name passing interaction was already present in the operational aspects of Linear Logic, albeit implicitly: as Hyland and Ong suggested, the intensional structure of game semantics can also be characterised as typed name passing. The type structures in game semantics can be considered as a "polarised" version of those used in Linear Logic, and, in that sense, is closely related with a polarised version of Linear Logic studied by Laurent [71]. In the present logical framework, a notion of polarities which indicate directions of flow of information, plays a fundamental role. As we already discussed, the type structures coming from these works allow us to have a precise embedding of programming languages into typed processes, which is a fundamental element of the present framework. As we shall see in the subsequent sections, there is a clean correspondence between type structures used and the shape of assertions for the class of typed behaviours, not only in purely functional behaviour but also in stateful ones.

1.3 Further Topics

The present work is but a first step towards language-independent theory of logics for software behaviour. Some of the remaining issues include: deeper studies of models for the presented logics, both in its theoretical and practical aspects, which would shed light on the method of calculation of validity in the present logics; extensions of the framework to the realm of behaviours beyond those treated so far, which include exceptions, input/output, concurrency and distribution (some of these may not be hard while some others can be challenging);

Experiment with individual mappings from a program logic to a process logic for non-trivial programming languages; Development of practical methods to integrate/interface distinct program logics using the underlying π-calculus logic; Search for natural specification methods for language-independent correctness of software, including experiments with practically feasible notations for them; incorporation of the present theory into a general specification framework such as VDM [66] and Z [118]; The notions of, and calculi for, refinement in the present framework; The use of the present framework for software testing; and a large scale experiment of engineering feasibility of the presented theory.

1.4 Structure of the Paper

This paper gives a comprehensive introduction to process logics and their applications. Part I focusses on sequential computation, while Part II will treat logics for a basic form of concurrency. A companion volume [53] offers a compilation of soundness proofs. In the following we present the outline of Part I.

Section 2 presents an informal preview of key technical ideas, outlining how the basic ideas in Hennessy-Milner Logic lead to, in the typed setting, compositional logics for assertions. Section 3 introduces the logic for linearly typed processes (which represent pure, terminating functional behaviour), as the simplest form of the typed process logic. We present the syntax and semantics of assertions, illustrate its proof rules, and conclude with a few examples. Section 4 augments the linear process logic with divergence. The single logic is now ramified into three: one about total correctness, one about partial correctness, and one about what we call generalised total correctness. Section 4 focusses on the logic for total correctness, while Section 5 discusses the logic for partial correctness and the generalised total logic, presenting the semantics and proof rules, as well as illustrating their difference via examples.

Sections 6. 7 and 8 are devoted to applications of process logics to program logics of purely functional computation. First, Section 6 derives a logic for total correctness for the call-by-value PCF (abbreviated as PCF_v) from the total process logic, establishing the former's soundness via embedding into the latter. We also discuss how various (purely functional) data types can be treated in the present logic. Section 7 develops two other variants of the program logic for PCF_v (one for partial correctness and the other for generalised total correctness), as well as the logics for the call-by-name PCF. The distinction between call-by-name and call-by-value leads to quite different program logics, both being justifiable through the underlying process logics. Section 8 discusses program logics for advanced data types. First the process logic is augmented with recursive types and second-order polymorphism, accompanied by simple proof rules. This is then reflected onto the program logics for corresponding data structures in PCF_v, for which we offer several non-trivial reasoning examples.

From Section 9 to the final section, we incorporate the notion of state into the classes of behaviour we consider, hence into the process/program logics. Section 9 treats logics for processes with open state, where "open" means state is not hidden but is exposed to the outside globally. Section 10 then demonstrates how the process logic for open state leads to logics for programming languages with (open) state, treating PCF_v with global state and the standard procedural language without dynamic structure. Section 11 then incorporates, on the basis of the process logic with open state, the behaviour in which state is hidden locally, enriching the realisable classes of behaviour. This in particular allows us to treat behaviour of dynamic data structure with destructive update. As a basic application of a logic with local state, Section 12 treats a simple program logic for an object-oriented language.

1.5 Notation

Throughout the paper we use the standard logical connectives ($\wedge$, $\vee$, $\neg$, $\supset$, $\forall$ and $\exists$) and constants (T and F), assuming their standard precedence/association. Among others:

- $A \supset B \supset C$ is parsed as $A \supset (B \supset C)$.
- $\forall x.A \circledast B$ (resp. $\exists x.A \circledast B$, resp. $\neg A \circledast B$) is parsed as $(\forall x.A) \circledast B$ (resp. $(\exists x.A) \circledast B$, resp. $(\neg A) \circledast B$) for each binary connective $\circledast$; and
- $A \circledast B \supset B$ is parsed as $(A \circledast B) \supset B$ for $\circledast \in \{\wedge, \vee\}$.

Thus, e.g. $A \wedge B \supset \forall x.C \vee D \supset E$ is parsed as $(A \wedge B) \supset (((\forall x.C) \vee D) \supset E)$. $\equiv$ on formulae is used to denote logical equivalence, i.e. $A \equiv B$ means $(A \supset B) \wedge (B \supset A)$. As far as no confusion arises, we use these connectives both syntactically (i.e. as connectives in formulae in process/program logics) and semantically (i.e. for discussing validity of various kinds). Our basic reference on the predicate calculus with equality (on which all presented logics for processes and programs are based), including terminology and inference rules, is Mendelson's book [73].

2 Preview of Main Ideas

2.1 Syntax of Processes

The syntax of processes we use is the asynchronous version of the π-calculus augmented with branching primitives. Branching and selection offer the basic forms of reception and emittance of information (or difference): the branching/selection as used in the following, are easily encodable into the syntax without them: however having them is useful in the typed setting, just as such constructs as injections and pairs are useful in the context of typed λ-calculi.

Let $x, y, \ldots$ range over an infinite set of names. $\vec{x}$ etc. denotes a vector of names. The grammar of processes follows.

$$P \ ::= \ x[\&_{i\in I}(\vec{y}_i).P_i] \ \mid \ \overline{x}\mathtt{in}_i(\vec{y})P \ \mid \ !x(\vec{y}).P \ \mid \ \overline{x}(\vec{y})P \ \mid \ P|Q \ \mid \ (\boldsymbol{\nu}\, x)P \ \mid \ \mathbf{0}\,.$$

$x[\&_{i\in I}(\vec{y}_i).P_i]$ is a (linear) *branching*, which offers I branches at a linear input channel x and, when the i-th one is selected and names are received, continues interaction as P_i after substitution. $\overline{x}\mathtt{in}_i(\vec{y})P$ is its dual, called (linear) *selection* which only send bound names [106]. We may consider this term as an abbreviation for $(\boldsymbol{\nu}\,\vec{y})(\overline{x}\mathtt{in}_i\langle\vec{y}\rangle|P)$ where $\overline{x}\mathtt{in}_i\langle\vec{y}\rangle$ is the standard free output with selection. Branching/selection use indexing sets: in the present paper we shall only consider $\{*\}$, $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$ and $\mathbb{N} = \{0, 1, \ldots\}$ as their indexing sets.[1] When the index is $*$ we use the standard unary notations (e.g. $x(\vec{y}).P$). We often omit the indexing set from branching. $!x(\vec{y}).P$ is a replicated input, which can be repeatedly invoked and, at each invocation, launches P after substitution. $\overline{x}(\vec{y})P$ is a dual output, which can again be considered as an abbreviation for $(\boldsymbol{\nu}\,\vec{y})(\overline{x}\langle\vec{y}\rangle|P)$ in the standard free output syntax. $P|Q$ is parallel composition of P and Q, $(\boldsymbol{\nu}\, x)P$ is the result of hiding x in P, and $\mathbf{0}$ is the inaction. We usually omit the empty vectors, writing e.g. $\overline{x}$, $x.P$, and $\overline{x}\mathtt{in}_i$.

The rules for the structural equality $\equiv$ are standard (see Appendix A.1). The reduction relation $\longrightarrow$ is generated from the following rules, closed under output prefix, restriction and parallel composition, taking processes modulo $\equiv$.

$$\begin{aligned} x[\&_i(\vec{y}_i).P_i] \mid \overline{x}\mathtt{in}_i(\vec{y}_i)Q &\longrightarrow (\boldsymbol{\nu}\,\vec{y}_i)(P_i \mid Q) \\ !x(\vec{y}).P \mid \overline{x}(\vec{y})Q &\longrightarrow\, !x(\vec{y}).P \mid (\boldsymbol{\nu}\,\vec{y})(P \mid Q) \end{aligned}$$

We present simple examples of processes.

Example 1. 1. $!x(w).\overline{w}$ is a *unit process*, which, when invoked at x with a parameter name, immediately outputs at that name.
2. Another example is $!x(w).\overline{w}\mathtt{in}_2$ is a process which outputs 2 after invocation.

---
[1] The use of infinite branching enable a clean representation of language constructs associated with infinite atomic data (the π-calculus is here used as a descriptive tool rather than for implementation). At the same time, it easily allows us to specify uncomputable interactions unless we put certain constraints. Such constraints are discussed in Appendix A.2.

3. As an example of branching, $!x(w).\overline{y}(u)u[\&_{n\in\mathbb{N}}.\overline{w}\mathtt{in}_{n*2}]$ would, when invoked at x, ask for a natural number at y, receives n (in the shape of branching), and gives the double of it as the answer to its original question.
4. $[x \to y] \stackrel{\text{def}}{=} !x(c).\overline{y}(c')c'.\overline{c}$ is a *copy-cat*, linking two locations x and y.
5. Let $\Omega_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)([u \to y]|[y \to u])$. Then we can observe $\Omega_u|\overline{u}(y)P \longrightarrow \cdots$, that is this process immediately diverges after the initial invocation at u.

2.2 Types

As untyped processes, the above syntax can represent a variety of sequential and concurrent behaviours, starting from pure higher-order functions [80] to shared variable concurrency to synchronously and asynchronously communicating processes. We use the following set of channel types to constrain their behaviour. Below $\mathbf{!}, \downarrow, \mathbf{?}, \uparrow$ are *action modes* or *modes* which indicate the ways a process would interact at a given channel (in brief: $\mathbf{!}$ and $\mathbf{?}$ respectively indicate replicated input and its dual output, while $\downarrow$ and $\uparrow$ are linear input and output).

$$\tau \quad ::= \quad (\vec{\tau}^{\mathbf{?}} \rho^{\uparrow})^{\mathbf{!}} \;\mid\; [\&_{i\in I}\vec{\tau}_i^{\mathbf{?}}]^{\downarrow} \;\mid\; (\vec{\tau}^{\mathbf{!}} \rho^{\downarrow})^{\mathbf{?}} \;\mid\; [\oplus_{i\in I}\vec{\tau}_i^{\mathbf{!}}]^{\uparrow} \;\mid\; \updownarrow$$

In the above grammar, $\rho^{\mathbf{!}}$ indicates the outermost mode of ρ is $\mathbf{!}$ (which we call the *mode of* τ). Similarly for $\vec{\rho}^{\mathbf{!}}$. Types of mode $\mathbf{!}, \uparrow$ are *positive*, while those of mode $\mathbf{?}, \downarrow$ are *negative*. $\updownarrow$ is *neutral*. $\overline{\tau}$ (defined only when $\tau \neq \updownarrow$) is the *dual* of τ exchanging $\mathbf{!}$ and $\mathbf{?}$, $\downarrow$ and $\uparrow$, as well as $\&$ and $\oplus$. An *action type* $(\Gamma, \Delta, \Theta, \ldots)$ is a finite map from names to channel types. $\Gamma; \Delta$ stands for an action type whose negative (resp. positive/neutral) component is Γ (resp. Δ). This notation becomes important when we discuss sequents of the logic. An action type is *sequential* if it contains at most one $\uparrow$-type. *In the present paper we shall only consider sequential action types.*

A typed term is given by the sequent of the following form:

$$\vdash P \triangleright \Gamma.$$

Two basic kinds of type disciplines, one called *linear* and another called *affine*, will be used in the present paper, where *linear* means processes are strongly normalising; and *affine* means they may contain divergence. We do not need the details of the typing rules in this section. The purely functional linear typing will be presented in Section 3, the corresponding affine typing in Section 4, and its stateful versions in Sections 9 and 11. In all these systems treated in the present paper, typable processes are sequential, in the sense that, at one time, there is at most one active process. A few examples of typable processes follow, using processes in Example 1.

Example 2. 1. $\vdash !x(w).\overline{w} \triangleright x : (()^{\uparrow})^{\mathbf{!}}$, where $()^{\uparrow}$ stands for $[\oplus_{i\in\{*\}}\varepsilon]^{\uparrow}$ such that ε is the null vector, which is often omitted). The type indicates:

The process can be repeatedly ($\mathbf{!}$) invoked at x, carrying a single name; to which it would signal an output once ($\uparrow$).

The interacting party, or the environment, would be given its dual typing, $x : (()^{\downarrow})^{?}$, which indicates the precisely dual behaviour: the process now would ask at a replicated channel x carrying a single name, via which it would once receive a natural number.

2. $\vdash !x(w).\overline{w}\mathtt{in}_3 \triangleright x : ([\oplus_{i\in\mathbb{N}}]^{\uparrow})^{!}$ where $\mathbb{N}$ is the set of natural numbers. This type means, at x, the process can be repeatedly invoked by a message carrying a single name: and via that single name, it would once output one of the natural numbers without carrying no further channel.
3. Let us write $\mathbb{N}^{\circ}$ for $([\oplus_{i\in\mathbb{N}}]^{\uparrow})^{!}$ for brevity. Then $\vdash !x(w).\overline{y}(u)u[\&_{n\in\mathbb{N}}.\overline{w}\mathtt{in}_{2*n}] \triangleright x : \mathbb{N}^{\circ}, y : \overline{\mathbb{N}^{\circ}}$ is linearly typed. Note $\overline{\mathbb{N}^{\circ}} \stackrel{\text{def}}{=} ([\&_{i\in\mathbb{N}}]^{\downarrow})^{?}$, indicating an action of asking at a channel to receive a branching action.
4. $\vdash [x \to y] \triangleright x : (()^{\uparrow})^{!}, y : (()^{\downarrow})^{?}$. In Section 3, we shall define $[x \to y]^{\tau}$ for an arbitrary input type τ, for which we have $\vdash [x \to y]^{\tau} \triangleright x : \tau, y : \overline{\tau}$.
5. Ω_x is untypable in the linear typing, but has type $\vdash \Omega_x \triangleright x : (()^{\uparrow})^{!}$ in the affine typing.

Thus types abstract behaviour of processes in a simple way: the logics we are going to study refine this abstraction so that they can capture observable behaviours with arbitrary precision.

2.3 Transition and Duality

A starting point of our study is a logic for processes pioneered by Hennessy and Milner [43]: and how, in the typed setting, it allows a surprisingly simple abstract representation. Let us take a small example, a process which asks the environment, via y, to obtain a number, and computes the double of it and returns it via an output channel z.

$$P \stackrel{\text{def}}{=} \overline{y}(c)c[\&_{n\in\mathbb{N}}\overline{z}\mathtt{in}_{n*2}]. \tag{2.1}$$

The behaviour of P can be understood in terms of the following transition sequence: $\overline{y}(c)\, c\mathtt{in}_n\, \overline{z}\mathtt{in}_{n*2}$. Since the π-calculus decomposes complex behaviours including those of higher-order procedures and objects into name passing, such transition sequences can represent many things — thus if we use Hennessy-Milner Logic for the π-calculus, we may as well be able to represent many kinds of behaviours, all using atomic name passing sequences.

Next consider the following process.

$$P' \stackrel{\text{def}}{=} \overline{y}(c)c[\&_{n\in\mathbb{N}}\overline{y}(c')c'[\&_{m\in\mathbb{N}}\overline{z}\mathtt{in}_{n+m}]. \tag{2.2}$$

This process asks for a natural number at y, receiving n: then it asks at y again, and, upon receiving m, returns the sum of n and m to z. Note that, even though their communication sequences differ considerably, P and P' would return precisely the same answer via z, the double of what y carries, *provided the agent waiting at y returns the same number each time it is asked.* This argument can be justified by noting, in the (purely functional) linear/affine π-calculus, P

and P' are equated in the naturally defined contextual equivalence, in spite of their apparently different interactive behaviours. The question is: how can we say P and P' have the same communication behaviour, even though they do have different communication sequences?

One possible answer, which has become the basis of the typed process logics, is to represent how the environment behaves. Assume for example the environment is the process representing the natural number 3 (which appeared in Example 2 (2)). Then we can write the behaviour of P *in the presence of that agent* as follows:

$$P \models \mathbf{rely}\ \overline{y}(c)c\mathtt{in}_3\ \mathbf{guar}\ \overline{z}\mathtt{in}_6 \tag{2.3}$$

This sequent says that, if P asks at y and the agent in the environment returns 3, then the process would in turn outputs 6 via z. **rely** means it relies on the behaviour of the environment as specified: while **guar** means it guarantees to perform the specified behaviour provided the **rely** condition is satisfied (the idea to use rely-guarantee in logics has a long history: among others the work by Cliff Jones uses this idea to extend Hoare Logic to concurrency). From a logical viewpoint, the behaviour written in **rely** part is *negative behaviour*, which receives information, while **guar** part specifies *positive behaviour*, which sends information. Note P' has precisely the same behaviour, assuming $\overline{y}(c)c\mathtt{in}_3$ in the rely part means the environment is prepared to return 3 how many times it is asked (this distinction between positivity and negativity roughly corresponds to polarised formulae in Linear Logic [71]). Since we can argue the same holds for arbitrary n, we may conclude P and P' have indeed the same behaviour.

2.4 Replicated behaviour

Milner [80] introduced the replication $!x(\vec{y}).P$ when he embeds the behaviour of higher-order functions in name passing processes. How does it affect our specification of communication behaviour? We take the following two agents, made from P and P' above:

$$Q \stackrel{\text{def}}{=} !x(yz).P, \qquad Q' \stackrel{\text{def}}{=} !x(yz).P' \tag{2.4}$$

We again expect they represent the same behaviour, and indeed they are again contextually equivalent. How can we express this fact? Since y and z are now bound, one possible idea is this:

$$Q \models \mathbf{rely}\ \emptyset\ \mathbf{guar}\ x(yz)(\overline{y}(c)c\mathtt{in}_3,\ \overline{z}\mathtt{in}_6) \tag{2.5}$$

It now relies on nothing (at least explicitly), but guarantees, when it is asked at x with two new names y and z, and if its question at y is answered with 3, returns 6. Note two specifications $\overline{y}(c)c\mathtt{in}_3$ and $\overline{z}\mathtt{in}_6$ are not temporally related but simply juxtaposed, reflecting the previous specification (2.3). This allows Q and Q' to again satisfy the same rely-guarantee condition. Noting these processes are replicated, we may as well write the behaviour of Q (and Q') by the following conjunctive formulae:

$$Q \models \mathbf{rely}\ \emptyset\ \mathbf{guar}\ \wedge_{n\in\mathbb{N}}\, x(yz)(\overline{y}(c)c\mathtt{in}_n,\ \overline{z}\mathtt{in}_{n*2}) \tag{2.6}$$

This is an infinite conjunction, saying:

> *If I am asked at x with two names, y and z, and the answer to my question at y is n, then I would return the double of n via z.*

2.5 Behaviour constants

In the specifications (2.3) and (2.5), we can see all names except the top ones are bound: in fact, it is easy to see we can abstract away these names one by one from the bottom, leaving only the initial free names, as the following mapping shows.

$$\begin{array}{rcl} \overline{y}(c)c\mathtt{in}_3 & \mapsto & y : (\mathtt{in}_3()^{\downarrow})^{?} \\ \overline{z}\mathtt{in}_6 & \mapsto & z : \mathtt{in}_6()^{\uparrow} \\ \wedge_{n\in\mathbb{N}} x(yz)(\overline{y}(c)c\mathtt{in}_n,\ \overline{z}\mathtt{in}_{n*2}) & \mapsto & x : \wedge_{n\in\mathbb{N}}((\mathtt{in}_n()^{\downarrow})^{?}(\mathtt{in}_{n*2}()^{\uparrow})^{!} \end{array}$$

Here we put symbols **!** etc. to indicate the directions and modes of actions. We call these type-like entities *behavioural constants*. The structure of behavioural constants precisely follow the corresponding type, but is more specific about the representing behaviour.

Let us now see how we can write the specifications for P and Q (hence P' and Q') using behavioural constants. Then we obtain, instead of (2.3) and (2.5):

$$P \models \mathbf{rely}\ \ y : (\mathtt{in}_3()^{\downarrow})^{?}\ \ \mathbf{guar}\ \ z : \mathtt{in}_6()^{\uparrow}\ , \tag{2.7}$$

$$Q \models \mathbf{rely}\ \ \emptyset\ \ \mathbf{guar}\ \ x : \wedge_{n\in\mathbb{N}}((\mathtt{in}_n()^{\downarrow})^{?}\mathtt{in}_{n*2}()^{\uparrow})^{!}. \tag{2.8}$$

Note we have assignments of behavioural constants to names in both rely and guarantee parts of the judgements.

2.6 Assertions for typed processes

Since behavioural constants are in general infinitary entities, they are not suitable for logical reasoning and specifications (while a proof system for deriving judgements as (2.7) above exists, it needs infinite conjunction). Furthermore, since they directly represent behaviour rather than describe them, they cannot allow us to be as vague as we like, for example to say P above returns an even number whenever invoked. This means we cannot specify properties we require, but always have to specify the whole process behaviour in every detail, which is theoretically limited and practically prohibitive: we need a way of asserting arbitrary properties of processes with arbitrary precision. This is why the use of logical formulae is required.

One of the main conceptual challenges to reach a suitable notion of assertions in the present context is how we can treat the counterpart of an infinitary entity such as $\wedge_{n\in\mathbb{N}}((\mathtt{in}_n()^{\downarrow})^{?}(\mathtt{in}_{n*2}()^{\uparrow})^{!})$ in a finitary, syntactic formula. For concreteness, let's consider this very formula, with respect to the behaviour of Q. How can we specify that, at x, Q has this behaviour? To answer this question, we go back to the origin of the behavioural constant. A behavioural constant represents communication, and the best way to unfold it is to interact — or to test its content by interaction [43]. For example, we may consider a testing of the form $x(yz).\overline{z}\mathtt{in}_6$, where we send two names to x, and we receive 6 as a return. We can represent this as:

$$x \bullet y = \mathtt{in}_6()^{\uparrow}. \tag{2.9}$$

In the above formula, $x \bullet y$ and $\mathtt{in}_6()^{\uparrow}$ are terms (as well as their components), which are equated as in the standard first-order logic with equality. The term $x \bullet y$, which consists of a name x, operator $\bullet$, and the singleton vector of names x, is best understood as abstraction of an output of form $\overline{x}(yc)$ (where we omit the compulsory linear name c); while $\mathtt{in}_6()^{\uparrow}$ can be understood as a linear output $\overline{c}\mathtt{in}_6$ (for some c). Thus $x \bullet y = \mathtt{in}_6()^{\uparrow}$ says that the invocation $\overline{x}(yc)$ is answered by a linear output $\overline{c}\mathtt{in}_6$. The bound c is not mentioned since its concrete naming is insignificant. Thus the above formula reads: "if we send y *and* a fresh name (say z) to x, the resulting reaction would be an output $\mathtt{in}_6()^{\uparrow}$ via z".

In the same way, the behaviour of the natural number agent $[\![3]\!]_y$ can be characterised by the formula "$y \bullet \varepsilon = \mathtt{in}_3()^{\uparrow}$" (where ε is the empty string). Using this, we can refine (2.9) by adding a constraint on y, obtaining:

$$\forall y.\ (y \bullet \varepsilon = \mathtt{in}_3()^{\uparrow}\ \supset\ x \bullet y = \mathtt{in}_6()^{\uparrow}). \tag{2.10}$$

We can further refine this into:

$$\forall y, i.\ (y \bullet \varepsilon = \mathtt{in}_i()^{\uparrow}\ \supset\ x \bullet y = \mathtt{in}_{i*2}()^{\uparrow}), \tag{2.11}$$

reaching the full specification of Q in finite syntax, as given in the following judgement:

$$Q \models \mathbf{rely}\ \top\ \mathbf{guar}\ \forall y, i.\ (y \bullet \varepsilon = \mathtt{in}_i()^{\uparrow}\ \supset\ x \bullet y = \mathtt{in}_{i*2}()^{\uparrow}). \tag{2.12}$$

which says that, under any (typable) environment, Q has the behaviour such that, if it is invoked with (a name at which it behaves as) a numeral i as an argument, then it returns the double of i as its answer. Note Q' has precisely the same specification.

As we noted, one of the benefits of using logical formulae is to be able to be vague. Thus, assuming $Even(n)$ means n is an even number, we can write:

$$Q \models \mathbf{rely}\ \top\ \mathbf{guar}\ \forall y, n.\ (x \bullet y = \mathtt{in}_n()^{\uparrow}\ \supset\ Even(n))\ . \tag{2.13}$$

Another instance of a vague specification, for P and P' above:

$$P \models \mathbf{rely}\ y \bullet \varepsilon = \mathtt{in}_n()^{\uparrow}\ \mathbf{guar}\ \exists m.\ z = \mathtt{in}_m()^{\uparrow} \tag{2.14}$$

As a convention, the variable n is implicitly universally quantified. Thus the judgement says that, whatever a typed agent outside would give to P, P returns an output of a certain number — the judgement does not however say which it is. In Section 3, we shall formally introduce the proof rules using which we can derive such assertions.[2]

[2] For convenience, we generally call Hoare-triple like logical judgements "assertions" and standard logical formulae "formulae," even though we also sometimes call a formula in a logical judgement "assertions" as far as no confusion arises.

2.7 Partial and total correctness

So far we have blurred one aspect of our specifications for sequential processes, which is the distinction between *partial correctness* and *total correctness*. In the former, specifications do not care about divergence: when a program diverges, the logic assigns an arbitrary specification. In the latter, convergence is implied when we say, for example, a program computes a double of a certain number. As a concrete illustration of this distinction in the present context, in the logic for partial correctness, a divergent replicated process Ω_u (which diverges whenever it is invoked, that is: $\Omega_u|\overline{u}(c)P \Uparrow$) satisfies all specifications (2.11), (2.12) and (2.14); while it satisfies none of them in the logic for total correctness.

While the logic for linear processes does not need this distinction (because all typed processes are always terminating), the logic for affine processes does need this distinction. In fact it is natural to consider the third variant, which we call the logic for generalised total correctness, where we can discuss both convergence and divergence of processes. Not only does this logic enjoy generality in that the assertions now capture both convergent and divergent behaviour, but also other two logics arise as special instances of this logic, both in semantics and proof rules. Since these three variations have distinct uses in reasoning about processes and programs, we discuss each of them one by one. In the present section we restrict our attention to total correctness.

2.8 A Program logics for PCF_v

Equational logics for higher-order functions have been studied since Church's λ-calculus. The logic for sequential processes as we outlined above directly gives rise to a program logic for sequential functions via the encoding of functions into processes, where assertions describe behaviours of programs rather than directly equate them. To see how this is possible, let us consider the call-by-value PCF, henceforth PCF_v, as our target calculus. We first show how the typing judgement in PCF_v translates to that of the π-calculus.

$$y_1:\beta_1,..,y_n:\beta_n \vdash M : \alpha \quad \mapsto \quad \vdash \langle\!\langle M \rangle\!\rangle_u \triangleright y_1:\overline{\beta_1^\circ},..,y_n:\overline{\beta_n^\circ}, u : (\alpha^\circ)^\uparrow$$

where α° is given: $\mathbb{N}^\circ \stackrel{\text{def}}{=} ([\oplus_{i\in\mathbb{N}}\varepsilon]^\uparrow)^!$ and $(\alpha \Rightarrow \beta)^\circ \stackrel{\text{def}}{=} (\overline{\alpha^\circ}(\beta^\circ)^\uparrow)^!$, where $\overline{\alpha}$ is the dual of α (i.e. the result of simply turning each symbol in α to its dual one). Note we need an additional name u attached to the term M. The key idea to move from the process logic to the corresponding to PCF_v is to retain this name u when asserting about the given term. This name u is called *anchor*. Thus we obtain, for example:

$$Even(x) \models x + 1 :_u Odd(u) \qquad (2.15)$$

which says that: *if what is denoted by the variable x is even, then the term $x+1$ (which is named with the anchor u) should be odd.* We contrast (2.15) with its "original" π-calculus specification:

$$\begin{aligned} !u(c).\overline{x}(c')c'[\&_n\overline{c}\mathtt{in}_{n+1}] \models\ & \mathbf{rely}\ \ \exists n.(x \bullet \varepsilon = \mathtt{in}_n()^\uparrow \wedge Even(n)) \\ & \mathbf{guar}\ \ \exists m.(u \bullet \varepsilon = \mathtt{in}_m()^\uparrow \wedge Odd(m)) \end{aligned}$$

here u and x are indeed interaction points, which are "folded" into simple variables in the assertion (2.15) for PCF_v.

One of the basic properties of such an embedding, which will be illustrated in later sections in detail, is that it is "logically fully abstract" in the sense of Longley and Plotkin [72]: that is, a program satisfies a certain formula if and only if its encoding satisfies the encoding of that formula. This property is fundamental when we prove the soundness of the proof rules for the derived logic via that of the original process logic, as we shall see in Section 6.

Let us present two more simple specifications for PCF_v. The first one uses a higher-order free variable, while the second one uses recursion.

$$\forall n.\ \mathit{Even}(y \bullet n) \models \lambda x.yx + 1 :_u \mathit{Odd}(u \bullet 3) \tag{2.16}$$

It says that if y is a function which always returns an even number then $\lambda x.yx+1$ will return an odd number for an argument 3, which is indeed reasonable.

A final specification is for a familiar recursive program for a factorial function.

$$\mathsf{T} \models \mu f.\lambda x.\texttt{if}\ x=0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1) :_u \forall n.\ u \bullet n = n!, \tag{2.17}$$

The specification should need no illustration. In Section 6 we shall formally introduce the logic for PCF_v and demonstrate how we can derive these assertions using compositional proof rules. Similarly Section 7 will treat call-by-name PCF.

2.9 Extensions

The method of specifications outlined above — which is based on the idea that behaviour is interactional, so that it is best specified by unfolding it one by one via interaction — has further significance when the complexity of behaviour increases, either in type/data structures or in the nature of computation such as statefulness. Later sections of the present note report how the proposed approach scales in both dimensions in the context of a large class of sequential behaviours, both for processes and for programs. The incorporation of state in target process behaviours leads to refinement in the shape of operators and assertions, from which we can derive program logic for higher-order imperative languages. Logical treatment of stateful behaviour is divided into two kinds, one for behaviour with global state (where significant state change is never hidden to observers) and one for behaviour with local state (where state change can generally be hidden). The latter is developed based on the former, both in the shape of assertions and in proof rules. The logic for processes with local state allows us to capture, among others, data structures with destructive update and sharing, extending the range of specifications to real-world programming examples. Another, and orthogonal, refinement is in terms of type structure: among others we treat second-order polymorphisms (both universal and existential abstractions) and recursive types. We note logics for sums and products are already embeddable in the first-order process logic.

These extensions are developed in the later part of the present paper, after we explore logics for purely functional behaviour of first-order types.

$$\frac{-}{\vdash \mathbf{0} \triangleright \_}\ (\mathsf{Zero}) \qquad \frac{\vdash P_1 \triangleright \Gamma_1\,;\, \Delta_1, \Theta \quad \vdash P_2 \triangleright \Gamma_2, \overline{\Theta}\,;\, \Delta_2}{\vdash P_1|P_2 \triangleright \Gamma_1 \cup \Gamma_2\,;\, \Delta_1,\, \Delta_2,\, \Theta'}\ (\mathsf{Par})\quad (\Theta \odot \overline{\Theta} = \Theta')$$

$$\frac{\vdash P \triangleright \Gamma,\, x\!:\!\tau \quad \mathsf{md}(\tau) \in \{\updownarrow, \mathbf{!}\}}{\vdash (\boldsymbol{\nu}\, x)P \triangleright \Gamma}\ (\mathsf{Res}) \qquad \frac{\vdash P \triangleright \Gamma^{\text{-}x} \quad \mathsf{md}(\tau) \in \{\updownarrow, \mathbf{?}\}}{\vdash P \triangleright \Gamma\,,\, x\!:\!\tau}\ (\mathsf{Weak})$$

$$\frac{\forall i.\ \vdash P_i \triangleright \mathbf{?}\Gamma^{\text{-}x},\, \vec{y}_i\!:\!\vec{\tau}_i,\, v\!:\!\rho^{\uparrow}}{\vdash x[\&_i(\vec{y}_i).P_i] \triangleright \Gamma,\, x\!:\![\&_i\vec{\tau}_i]^{\downarrow},\, v\!:\!\rho}\ (\mathsf{Bra}^{\downarrow}) \qquad \frac{\vdash P \triangleright \mathbf{?}\Gamma,\, \vec{y}\!:\!\vec{\tau}_i}{\vdash \overline{x}\mathtt{in}_i(\vec{y})P \triangleright \Gamma,\, x\!:\![\oplus_{i\in I}\vec{\tau}_i]^{\uparrow}}\ (\mathsf{Sel}^{\uparrow})$$

$$\frac{\vdash P \triangleright \mathbf{?}\Gamma^{\text{-}x},\, \vec{y}\!:\!\vec{\tau},\, z\!:\!\rho}{\vdash !\, x(\vec{y}z).P \triangleright \Gamma,\, x\!:\!(\vec{\tau}\rho)^{!}}\ (\mathsf{In}^{!}) \qquad \frac{\vdash P \triangleright \mathbf{?}\Gamma,\, \vec{y}\!:\!\vec{\tau},\, z\!:\!\rho',\, v\!:\!\rho^{\uparrow}}{\vdash \overline{x}(\vec{y}z)(P) \triangleright \Gamma,\, v\!:\!\rho}\ (\mathsf{Out}^{?})\quad (\Gamma(x) = (\vec{\tau}\rho')^{?})$$

Fig. 1. Typing Rules for Sequential Linear Processes

3 A Process Logic for Stateless, Terminating Behaviour

3.1 The Linear π-Calculus

We begin our study of typed process logics with a most restricted form of sequential behaviour, the linearly typed processes [116]. Linearly typed processes are characterised by two features, statelessness (in the sense that interacting with a replicated input twice leads to identical interactions) and guaranteed termination (in the sense that typable processes always terminate). The lack of state change makes logical articulation of behaviour much simpler than otherwise (this is mainly because of the strict polarisation of channels, cf. Section 2.2), while the guarantee of termination means processes never have divergence, so that distinction between total and partial correctness need not be considered. These two aspects lead to a simplest, yet non-trivial, typed logic for processes, offering an ideal starting point for the present inquiries.

We use the same syntax of processes and types as introduced in Section 2. The typing rules for linearly typed processes are given in Figure 1, using the sequent $\vdash P \triangleright \Gamma$ (which we often write P^{Γ}). In each rule, we assume all action types are well-formed, including those in conclusions. "," in "Γ, Δ" denotes the union of Γ and Δ, indicating at the same time disjointness of the their domains. $\mathbf{?}\Gamma$ etc. indicates the mode of the types in Γ, while $\Gamma^{\text{-}x}$ says x does not occur in Γ. In (Par), the notation $\Gamma; \Delta$ means Γ uses negative channel types while Δ uses positive/neutral channel types. The operation $\Theta \odot \overline{\Theta}$ (where the notation $\overline{\Theta}$ assumes Θ does not contain $\updownarrow$) is given by the following induction:

$$(1)\ \ \emptyset \odot \emptyset = \emptyset \qquad (2)\ \ (\Theta \cdot x : \tau) \odot (\overline{\Theta} \cdot x : \overline{\tau}) \ = \ \Theta \odot \overline{\Theta} \cdot x : (\tau \odot \overline{\tau})$$

where we set $\tau^{!} \odot \overline{\tau} = \overline{\tau} \odot \tau^{!} = \tau^{!}$ and $\tau^{\downarrow} \odot \overline{\tau} = \overline{\tau} \odot \tau^{\downarrow} = \updownarrow$ (which says a replicated channel can be composed with as many dual outputs as possible; while a linear input channel and its dual, once composed, can never be used again). In (Res) and (Weak), $\mathsf{md}(\tau)$ denotes the mode of τ, with $\mathsf{md}(\updownarrow) = \updownarrow$.

The typing system in Figure 1 is a variant of sequential linear typing in [116]. The typing does not use IO-modes (unlike [13, 116]), but can still ensure sequentiality thanks to the use of sequential action types. Further it dispenses with causality edges by having an equivalent non-circularity condition in (Par) rule. This typing is used because it offers a clearer articulation of proof rules (a drawback of not having IO-modes/causality edges is that the typability is not closed under $\equiv$, or, more concretely, under associativity of $|$, even though this never causes a problem in the present investigation). Processes typable with these rules cover essentially all sequential linear processes in [116] modulo $\equiv$. The typability is closed under reduction in the way: $\vdash P \triangleright \Gamma$ and $P \longrightarrow P'$ implies $P' \equiv P''$ such that $\vdash P'' \triangleright A$ for some P''.

The processes typable in this system are called *linearly typable*. The typability is closed under reduction in the sense that $\vdash P \triangleright \Gamma$ and $P \longrightarrow P'$ implies $P' \equiv P''$ for some P'' such that $\vdash P'' \triangleright A$. By the result in [116], linearly typable processes are strongly normalising. Their another property is sequentiality, that is if P is typable then $P \longrightarrow P_{1,2}$ implies $P_1 \equiv P_2$.

3.2 Syntax of Assertions

In Section 2, we have already seen some examples of assertions. An assertion uses a pair of standard first-order logical formulae (one for rely and another for guarantee) to describe behavioural properties of a typed process. Formulae are made up from equations on terms and their combinations by usual logical connectives. For example, in the following assertion:

$$\overline{y}(c)c[\&_i \overline{z}\mathtt{in}_i] \models \mathbf{rely}\ \ y \bullet \varepsilon = \mathtt{in}_n()^{\uparrow} \ \ \mathbf{guar}\ \exists m.\ z = \mathtt{in}_m()^{\uparrow} \tag{3.1}$$

"$y \bullet \varepsilon$", "$\mathtt{in}_n()^{\uparrow}$", "$z$" and "$\mathtt{in}_m()^{\uparrow}$" are terms, while "$y \bullet \varepsilon = n$" and "$\exists m.\ z = \mathtt{in}_m()^{\uparrow}$" are formulae. Below we formally introduce these constructs one by one.

Terms. The set of *terms* in the logic is given by the following grammar.

$$\begin{array}{lcl} e & ::= & * \ \mid\ \mathbf{t} \ \mid\ \mathbf{f} \ \mid\ n \ \mid\ x^{\mathsf{nat}} \ \mid\ x^{\mathsf{bool}} \ \mid\ e + e' \ \mid\ e \wedge e' \ \mid\ \ldots \\ a & ::= & x^{\tau} \ \mid\ a \bullet \vec{b} \ \mid\ \mathtt{in}_e^{\rho^{\uparrow}}(\vec{a})^{\uparrow}\ . \end{array}$$

The first set of terms $(e, e', \ldots)$ are called *data expressions*, while the second set $(a, a', \ldots)$ *behaviour expressions*. We call "$a \bullet \vec{b}$" and "$\mathtt{in}_e^{\rho}(\vec{a})^{\uparrow}$", *invocation* and *linear output*, respectively. Data expressions denote elements from a suitable universe of sets of atomic data with operations on them which at least includes a distinguished singleton with a unique element $*$, the set of booleans and the set of natural numbers. Constants in data expressions (ranged over by $\mathsf{c}, \mathsf{c}', \ldots$) are often identified with the corresponding elements.

The well-typedness of data expressions are standard, assigning a unique type T to a well-typed term e. For behaviour expressions, the well-typedness is considered with respect to channel types, as inductively given below.

1. x^τ, where τ should always be positive, i.e. of mode either $\mathbf{!}$ or $\uparrow$, has type τ.
2. $a \bullet \vec{b}$ has type $\rho^\uparrow$ iff a has type $(\overline{\vec{\rho}}\tau)^{\mathbf{!}}$ and $\vec{b}$ has type $\vec{\rho}$ (where we say $a_1..a_n$ *has type* $\tau_1..\tau_n$ iff a_i has type τ_i for each i).
3. $\mathtt{in}^\rho_e(\vec{a})^\uparrow$ has type $\rho = [\oplus_{i\in T}\vec{\tau_i}]^\uparrow$ iff e has type T and that, for each valuation, if $e = i \in T$ under the valuation, then $\vec{a}$ has type $\vec{\tau_i}$ under the same valuation.

In Clause 3, a *valuation* is a map from typed variables to elements of the underlying sets. Each well-typed behaviour expression has a unique type thanks to the type annotation ρ in $\mathtt{in}^\rho_e(\vec{a})^\uparrow$: in spite of this, we often omit this type annotation. $\mathtt{in}_e(\vec{a})^\uparrow$ is written ${}^\rho(\vec{a})^\uparrow$ when the underlying set is a singleton. Also note if $\mathsf{md}(\tau) = \mathbf{!}$ then a^τ is always a name by well-typedness.

Formulae. The set of formulae of the specification logic is given by the following grammar. Below $\star$ ranges over $\{\wedge, \vee, \supset\}$ and $\mathcal{Q}$ over $\{\forall, \exists\}$.

$$A \ ::= \ e_1 = e_2 \ | \ a_1 = a_2 \ | \ \neg A \ | \ A_1 \star A_2 \ | \ \mathcal{Q}i.A \ | \ \mathcal{Q}x.A$$

We also use the truth T (definable as, for example, $1 = 1$) and the falsity F (which is $\neg\mathsf{T}$). The quantifications induce binding, for which we assume the standard bound name convention. $\mathsf{fn}(A)$ denotes the set of free names in A. A is *well-typed* if whenever a variable/name occurs twice they own the same type and each pair of equated terms have the same type. *We only consider well-typed formulae from now on.*

Let us partition the set of free names occurring in A into two, say X_1 and X_2 (we later discuss how such a partition is decided). We then write $\Gamma;;\Theta \vdash A$ iff:

1. Γ and Θ are disjoint in their domains such that $X_1 \subset \mathsf{dom}(\Gamma)$ and $X_2 \subset \mathsf{dom}(\Theta)$; and
2. Each name, say x, in A has type τ iff either $\Gamma(x) = \tau$ or $\Theta(x) = \tau$.

Note a formula never contains a name typed with $\updownarrow$, so a neutral type in Γ of A^Γ is never used for typing.

We can now formally introduce the syntax of assertions. Assume given P such that $\vdash P \triangleright \Gamma; \Delta$, that is P has a typing with Γ for its negative channels, Δ for its positive/neutral channels. Then an *assertion for* $P^{\Gamma;\Delta}$, or often simply an *assertion*, has the shape:

$$P^{\Gamma;\Delta} \models \mathbf{rely}\ A\ \mathbf{guar}\ B \tag{3.2}$$

where we assume $\overline{\Gamma};;\Theta \vdash A$ and $\Delta;;\Theta \vdash B$ for some Θ. Note formulae in assertions are completely typed in their use of names. An assertion is often written $P^{\Gamma} \models \mathbf{rely}\, A\, \mathbf{guar}\, B$ without mentioning the polarisation, or, with a typing implicit, even $P \models \mathbf{rely}\, A\, \mathbf{guar}\, B$. We can read (3.2) as:

> *Given a process P typed as $\Gamma;\Delta$, if we can rely on A from the behaviour of an environment at $\overline{\Gamma}$, then the process combined with this environment can guarantee B at Δ.*

We dualise Γ above, since A is specified from the viewpoint of the environment rather than that of the process (which makes the notation less verbose). Note this assertion mentions neither state change nor termination: this is because linear processes are stateless and are always terminating. We also note we may consistently allow negative names to occur in the guarantee formula for convenience in proofs (cf. Remark 1 later), even though in the present section we shall stick to the present more strict name usage.

The following distinction in name usage, coming from Hoare Logic, is significant in the whole family of typed process logics.

Definition 1. (primary and auxiliary names) *Let $\vdash P \triangleright \Gamma;\Delta$ and fix an assertion $P^{\Gamma;\Delta} \models \mathbf{rely}\, A_1\, \mathbf{guar}\, A_2$. Then: (1) The set of* primary names in A_i, *written* $\mathsf{prim}(A_i)$*, are those names which occur in A_i as well as in $\Gamma;\Delta$. (2) The set of* auxiliary names in A_i, *written* $\mathsf{aux}(A_i)$*, are those names occurring in A_i which are not primary.*

This distinction is illustrated in the next subsection using concrete examples.

3.3 Examples of Assertions

Let $\mathsf{double}_{xy} \stackrel{\text{def}}{=} !x(w).\overline{y}(u)u[\&_{n\in\mathbb{N}}.\overline{w}\mathtt{in}_{2*n}]$. As we have seen in Example 2 (3) in Section 2, we can type this process as $\vdash \mathsf{double}_{xy} \triangleright \Gamma$ with $\Gamma \stackrel{\text{def}}{=} y:\overline{\mathbb{N}^\circ}, x:\mathbb{N}^\circ$. We first consider a simple, concrete assertion for this process.

$$\mathsf{double}_{xy}^{\Gamma} \vdash \ \mathbf{rely}\ \ y \bullet \varepsilon = \mathtt{in}_3()^{\uparrow}\ \ \mathbf{guar}\ \ x \bullet \varepsilon = \mathtt{in}_6()^{\uparrow}\ \ . \tag{3.3}$$

The assertion says that, if the environment is such that, whenever it is invoked at y with the null argument, it returns (a linear output whose index is) 3, then the process would return 6 when it is invoked similarly. In this assertion, the set of primary names are $\{x, y\}$ while there are no auxiliary names. To see (3.3) is a sound judgement, we can compose P with $!y(c).\overline{c}\mathtt{in}_3$, then the composed process will surely output 6 when invoked at x.

The next assertion is more detailed than (3.3).

$$\mathsf{double}_{xy}^{\Gamma} \vdash \ \mathbf{rely}\ y \bullet \varepsilon = \mathtt{in}_{i^{\mathbb{N}}}()^{\uparrow}\ \mathbf{guar}\ x \bullet \varepsilon = \mathtt{in}_{2*i^{\mathbb{N}}}()^{\uparrow} \tag{3.4}$$

Since i does not occur in Γ, i is auxiliary. As in Hoare Logic, auxiliary names are considered to be universally quantified in the whole sequent. Thus (3.4) says that, for each $i \in \mathbb{N}$, if the environment is such that it returns i for its invocation at y, then P returns $2 * i$, which indeed sounds quite reasonable.

The following assertion is more vague than (3.4).

$$\mathsf{double}^{\Gamma}_{xy} \vdash \ \mathbf{rely}\ \mathsf{T}\ \mathbf{guar} \quad \exists i^{\mathbb{N}}.\,(x \bullet \varepsilon = \mathtt{in}_{2*i}()^{\uparrow})\ . \tag{3.5}$$

Note T is well-typed under $y : \overline{\mathbb{N}^{\circ}}$. Intuitively (3.5) says that however an environment typed under $y : \overline{\mathbb{N}^{\circ}}$ would behave, P would, upon its invocation, return an even number.

Let us now consider a process which slightly alters double_{xy}, defined as:

$$\mathsf{double}'_{x} \stackrel{\text{def}}{=} !x(yw).\overline{y}(u)u[\&_{n\in\mathbb{N}}.\overline{w}\mathtt{in}_{2*n}].$$

In $\mathsf{double'}_x$, the free y in P becomes an argument of the initial invocation, hence is now bound. The process is now typed as $\mathsf{double'}_x \vdash \mathbf{rely}\ \Delta\ \mathbf{guar}$ where we set $\Delta \stackrel{\text{def}}{=} x : (\overline{\mathbb{N}^{\circ}}[\oplus_{i\in\mathbb{N}}]^{\uparrow})^{!}$. Then the assertion corresponding to (3.4) becomes:

$$\mathsf{double'}^{\Delta}_{x} \vdash \ \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ y \bullet \varepsilon = \mathtt{in}_i()^{\uparrow} \supset x \bullet y = \mathtt{in}_{2*i}()^{\uparrow}. \tag{3.6}$$

Note y is now auxiliary, together with i. Since both auxiliary names are now in a single formula, we can universally quantify them inside the formula, so that (3.6) is equivalent to:

$$\mathsf{double'}^{\Delta}_{x} \vdash \ \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \forall y, i.\,(y \bullet \varepsilon = \mathtt{in}_i()^{\uparrow} \supset x \bullet y = \mathtt{in}_{2*i}()^{\uparrow}). \tag{3.7}$$

(3.6) and (3.7) both say that, for each i and y, if $\mathsf{double'}_x$ is invoked at x with a single argument y that returns i when invoked, then the process will return $2*i$ to the invoker, giving the full detail of the behaviour of $\mathsf{double'}_x$.

After getting familiar with the basic ideas of assertions, our next task is to make precise what these assertions mean.

3.4 Meaning of Assertions

In the standard Hoare Logic, an assertion speaks about a state transformation caused by a program. A state is a finite map from variables to natural numbers. If we have a Hoare triple $\{C\}P\{C'\}$, then the assertion says that each state which satisfies C should be transformed, under the assumption of termination of P, into some state which satisfies C'. The world of states and the world of programs are sharply separated.

In the present setting, we cannot maintain this separation. Given an assertion $P \models \mathbf{rely}\ A\ \mathbf{guar}\ B$, the formulae A and B also speak about process behaviours, that is, about the very world where P resides. This apparent circularity however comes from the core spirit of the present theory: the universe of typed processes is *the* place where diverse computational situations are asserted and interpreted, including such notions as memory regions and synchronisations, without any need of referring to the outside. Thus a model needs be stipulated which are about the very universe of behaviours of typed processes we are interested in.

In the following we shall present a simplest way to define such a model (an alternative, but equivalent, model construction is discussed in Appendix B, which use behavioural constants discussed in Section 2). We use the standard contextual equality on typed processes. Below we write $C[\ \cdot\ ]^{\Delta}_{\Gamma}$ for a typed context whose hole has type Γ and whose resulting type is Δ.

$$P^{\Gamma} \cong Q^{\Gamma} \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \forall C[\ \cdot\ ]^{x:[\oplus_{i\in\mathbb{B}}]^{\uparrow}}_{\Gamma}.\ (C[P] \Downarrow^{\mathtt{t}}_{x} \quad \equiv \quad C[Q] \Downarrow^{\mathtt{t}}_{x})\ .$$

Model. Let τ be positive. Then an *abstract process* p of type τ, often written p^{τ}, is a map from the names to the $\cong$-congruence classes such that: (1) $P \in p(x)$ implies $\vdash P \triangleright x : \tau$, and (2) p is closed under injective renaming: $p(x) = p(y)\binom{xy}{yx}$ for each x, y, where $\binom{xy}{yx}$ is a name permutation acting pointwise on processes. For example, an abstract process corresponding to a simplest linear output $\overline{x}$ (of type $()^{\uparrow}$) is $\{P^{y:()^{\uparrow}} \mid P \cong_{\pi} \overline{y}, y \in \mathcal{N}\}$ where $\mathcal{N}$ is the set of all names.

A *model* ξ is a well-typed finite map from names to abstract processes. ξ has type Γ iff $\xi = x_1 : p_1^{\tau_1}, .., x_n : p_n^{\tau_n}$ and $\Gamma = x_1 : \tau_1, .., x_n : \tau_n$. ξ^{Γ} indicates ξ which has type Γ. We write $P^{\Gamma} \models_{\mathtt{b}} \xi^{\Gamma}$ with $\xi = x_1 : p_1, .., x_n : p_n$ for $P \cong \Pi_i P_i$ such that $P_i \in p_i(x_i)$ for each i.

Satisfaction. Fix auxiliary names in A and their interpretation $\mathcal{I}$, the latter mapping the former to constants and abstract processes. The definition of $\mathcal{I}[\![e]\!]$ is standard. For behaviour expressions, we set $[\![a]\!]_{\mathcal{I}\cdot\xi}$ by:

- $[\![x]\!]_{\mathcal{I}\cdot\xi} = (I \cdot \xi)(x)$,
- $[\![x \bullet \vec{y}]\!]_{\mathcal{I}\cdot\xi} = [\![x]\!]_{\mathcal{I}\cdot\xi} \bullet [\![y_1]\!]_{\mathcal{I}\cdot\xi} .. [\![y_n]\!]_{\mathcal{I}\cdot\xi}$ and
- $[\![\mathtt{in}_e(x_1..x_n)^{\uparrow}]\!]_{\mathcal{I}\cdot\xi} = \mathtt{in}_{\mathcal{I}[\![e]\!]}([\![x_1]\!]_{\mathcal{I}\cdot\xi} .. [\![x_n]\!]_{\mathcal{I}\cdot\xi})^{\uparrow}$.

where $p \bullet q_1..q_n$ and $\mathtt{in}_i(p_1..p_n)^{\uparrow}$ are given as follows.

- $p \bullet \vec{q}$ is the abstract process whose image at, say, z is given by $(\boldsymbol{\nu}\, x\vec{y})(p(x) \mid \overline{x}(\vec{y}z')(\Pi q_i(y_i) \mid [z' \to z]^{\overline{\rho}}))$.
- $\mathtt{in}_j(p_1..p_n)^{\uparrow}$ is the abstract process whose image at, say, x is given by $\overline{x}\mathtt{in}_j(\vec{y})\Pi_i p_i(y_i)$.

Above $p(x)$ etc. should in fact be a concrete process in $p(x)$: confusing a $\cong$-congruence class and its element in these clauses, or their use of concrete names, in these clauses does not lead to ambiguity because of congruency and renaming closure, due to which an abstract process is uniquely determined by picking up one concrete process in its image at an arbitrary name.

We can now introduce the satisfaction of A by a model ξ under an interpretation $\mathcal{I}$, written $\xi \models^{\mathcal{I}} A$, by induction on the structure of A. We start from:

$$\begin{array}{lcl} \xi \models^{\mathcal{I}} e_1 = e_2 & \equiv & \mathcal{I}[\![e_1]\!] = \mathcal{I}[\![e_2]\!]. \\ \xi \models^{\mathcal{I}} a_1 = a_2 & \equiv & [\![a_1]\!]_{\mathcal{I}\cdot\xi} = [\![a_2]\!]_{\mathcal{I}\cdot\xi} \end{array}$$

The rest is the standard interpretation of logical connectives. Below $[\![\tau]\!]$ (for τ positive) denotes the set of all definable behaviour constants of type τ.

$$\begin{array}{rcl}
\xi \models^{\mathcal{I}} A_1 \wedge A_2 & \equiv & (\xi \models^{\mathcal{I}} A_1) \quad \wedge \quad (\xi \models^{\mathcal{I}} A_2) \\
\xi \models^{\mathcal{I}} A_1 \vee A_2 & \equiv & (\xi \models^{\mathcal{I}} A_1) \quad \vee \quad (\xi \models^{\mathcal{I}} A_2) \\
\xi \models^{\mathcal{I}} A \supset B & \equiv & (\xi \models^{\mathcal{I}} A) \quad \supset \quad (\xi \models^{\mathcal{I}} B) \\
\xi \models^{\mathcal{I}} \neg A & \equiv & \neg\, (\xi \models^{\mathcal{I}} A) \\
\xi \models^{\mathcal{I}} \forall x^{T}.A & \equiv & \forall \mathsf{c} \in T.\ \xi \models^{I\cdot x:\mathsf{c}} A, \\
\xi \models^{\mathcal{I}} \forall x^{\tau}.A & \equiv & \forall p \in [\![\tau]\!].\ \xi \models^{I\cdot x:p} A, \\
\xi \models^{\mathcal{I}} \exists x^{T}.A & \equiv & \exists \mathsf{c} \in T.\ \xi \models^{I\cdot x:\mathsf{c}} A. \\
\xi \models^{\mathcal{I}} \exists x^{\tau}.A & \equiv & \exists p \in [\![\tau]\!].\ \xi \models^{I\cdot x:p} A.
\end{array}$$

The interpretation of quantifiers treats names just like standard variables, which reflects the stateless nature of linear processes.

We can now define the semantics of assertion.

Definition 2. (semantics of assertions) *Let $P \models^{\mathcal{I}} A$ stand for $\xi.(P \models_{\mathsf{b}} \xi$ and $\xi \models^{\mathcal{I}} A$ for some ξ. Then an assertion $P^{\overline{\Gamma};\Delta} \models \textbf{rely}\, A\, \textbf{guar}\, B$ is valid when, for each appropriate $\mathcal{I}$ of type Θ, and for each R^{Γ}, the following implication holds.*

$$R^{\Gamma} \models^{\mathcal{I}} A \quad \supset \quad (\boldsymbol{\nu}\, \mathsf{fn}(\Gamma))(P|R) \models^{\mathcal{I}} B\ .$$

Henceforth we simply write $P^{\overline{\Gamma};\Delta} \models \textbf{rely}\, A\, \textbf{guar}\, B$ to indicate the assertion is valid, often omitting type annotation as far as no confusion arises.

One of the basic properties of assertions on linear processes is that they can capture any observable differences of processes, in the sense that the set of valid formulae for a process precisely characterise its semantics up to the observational congruence, as we shall prove at the end of the section.

3.5 Proof Rules

One of the significant aspects of the typed process logic is that there are simple compositional proof rules by which we can syntactically infer valid assertions. We write $P^{\Gamma;\Delta} \vdash \textbf{rely}\, A\, \textbf{guar}\, B$ for a syntactically derivable assertion, under the same typability condition as before. In the following we stipulate:

Convention 1. *In each proof rule, all occurring sequents are well-typed including in the conclusions. If two or more sequents appear in the antecedent, a primary name of one sequent is never used as an auxiliary name of another.*

First, if a behaviour satisfies A, its composition with $\mathbf{0}$ satisfies A.

$$(\mathsf{Zero})\ \frac{-}{\mathbf{0}^{\emptyset} \vdash \textbf{rely}\, A\, \textbf{guar}\, A} \tag{3.8}$$

Next, if some behaviour satisfies B assuming A, and B does not mention x, then the same behaviour with x hidden, also does so.

$$(\mathsf{Res})\ \frac{P^{\Gamma,x:\tau} \vdash \textbf{rely}\, A\, \textbf{guar}\, B^{\text{-}x}}{(\boldsymbol{\nu}\, x)P^{\Gamma} \vdash \textbf{rely}\, A\, \textbf{guar}\, B} \tag{3.9}$$

(Zero)
$$\frac{-}{\mathbf{0}^{\emptyset} \vdash \mathbf{rely}\ A\ \mathbf{guar}\ A}$$

(Par)
$$\frac{P_1 \vdash \mathbf{rely}\ A_1\ \mathbf{guar}\ B_1 \wedge E \qquad P_2 \vdash \mathbf{rely}\ A_2 \wedge E\ \mathbf{guar}\ B_2}{P_1 | P_2 \vdash \mathbf{rely}\ A_1 \wedge A_2\ \mathbf{guar}\ B_1 \wedge B_2}$$

(Res)
$$\frac{P^{\Gamma, x:\tau} \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B^{\text{-}x}}{(\boldsymbol{\nu}\, x)P^{\Gamma} \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}$$

(Weak)
$$\frac{P^{\Gamma} \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}{P^{\Gamma, x:\tau} \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}$$

(Bra$^{\downarrow}$)
$$\frac{\forall i.\ P_i \vdash \mathbf{rely}\ A[\mathtt{in}_i(\vec{y}_i)^{\uparrow}/x]\ \mathbf{guar}\ B}{x[\&_i(\vec{y}_i).P_i] \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}$$

(Sel$^{\uparrow}$)
$$\frac{P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B[\mathtt{in}_i(\vec{y})^{\uparrow}/x]}{\overline{x}\mathtt{in}_i(\vec{y})P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}$$

(In$^{!}$) $(\forall \vec{y}.\, (B_1 \supset B_2[x \bullet \vec{y}/z]) \supset B)$
$$\frac{P \vdash \mathbf{rely}\ A^{\text{-}\vec{y}} \wedge B_1^{\vec{y}}\ \mathbf{guar}\ B_2}{!x(\vec{y}z).P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}$$

(Out$^{?}$) $(A \supset A_1 \supset A_2[x \bullet \vec{y}/z])$
$$\frac{P \vdash \mathbf{rely}\ A^{\text{-}z} \wedge A_2^{z}\ \mathbf{guar}\ A_1^{\vec{y}} \wedge B^{\text{-}\vec{y}}}{\overline{x}(\vec{y}z)P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}$$

Fig. 2. Proof Rules for Linear Processes (main rules)

Here $B^{\text{-}x}$ says x does not occur in B. Note x does not occur in A either because of the typing.

Next if some behaviour which does not own x satisfies B under A then we may as well assume that the behaviour has x under the same specification since it gives no more and no less constraint.

$$\text{(Weak)}\ \frac{P^{\Gamma} \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}{P^{\Gamma, x:\tau} \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B} \tag{3.10}$$

Note x may as well occur in A as an auxiliary rule, which is indeed the only non-trivial usage of this rule.

The rule for parallel composition is reminiscent of the cut rule in the sequent calculus. Assume P_1 guarantees B_1 and E relying on A_1, while P_2 guarantees B_2 relying on A_1 and E. Then if P_1 and P_2 are composed and can rely on A_1 and A_2, surely P_1 guarantees B_1 and E, on which P_2 can rely in addition to A_2, hence P_2 can guarantee B_2, so P_1 and P_2 together guarantee both B_1 and B_2.

$$\text{(Par)}\ \frac{P_1 \vdash \mathbf{rely}\ A_1\ \mathbf{guar}\ B_1 \wedge E \quad P_2 \vdash \mathbf{rely}\ A_2 \wedge E\ \mathbf{guar}\ B_2}{P_1 | P_2 \vdash \mathbf{rely}\ A_1 \wedge A_2\ \mathbf{guar}\ B_1 \wedge B_2}$$

For linear selection we can reason thus:

$$\text{(Sel}^{\uparrow}\text{)}\ \frac{P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B[\mathtt{in}_i(\vec{y})^{\uparrow}/x]}{\overline{x}\mathtt{in}_i(\vec{y})P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B},$$

which says: for an output process $\overline{x}\mathtt{in}_i(\vec{y})P$ to guarantee B relying on A, P itself should guarantee B relying on A, where B's constraint on x is unfolded into those on its carried values, $\vec{y}$. Dually we reason for linear input:

$$(\mathsf{Bra}^{\downarrow})\ \frac{\forall i.\ P_i \vdash \textbf{rely}\ A[\mathtt{in}_i(\vec{y}_i)^{\uparrow}/x]\,\textbf{guar}\ B}{x[\&_i(\vec{y}_i).P_i] \vdash \textbf{rely}\ A\,\textbf{guar}\ B}$$

which says that, if $x[\&_i(\vec{y}_i).P_i]$ is to guarantee B relying on A, then each of its branch P_i should guarantee B relying on A which is now specialised, by substitution, into the i-th branch of its specification on x.

For replication and its dual, the rules are slightly more complex.

$$(\mathsf{In}^{!})\ \frac{P \vdash \textbf{rely}\ A^{\text{-}\vec{y}} \wedge B_1^{\vec{y}}\ \textbf{guar}\ B_2\ ,\ \ \forall \vec{y}.\,(B_1 \supset B_2[x\bullet\vec{y}/z]) \supset B}{!x(\vec{y}z).P \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

$A^{\text{-}\vec{y}}$ indicates no name in $\vec{y}$ occurs in A, while $B_1^{\vec{y}}$ indicates all primary names in B_1 are in $\vec{y}$. Note also $\vec{y}$ never occur in B_2 by typing. The rule is easier read downward: suppose P guarantees B_2 at z, relying on A and B_1, with only the latter being about $\vec{y}$. Then $!x(\vec{y}z).P$ can guarantee that, for all $\vec{y}$ satisfying B_1, the result of invoking x with $\vec{y}$ will satisfy B_2 which is now about x and $\vec{y}$ by substitution. ($\mathsf{In}^{!}$) has two equivalent variants, which are listed in Section 3.6.

The replicated output is reasoned dually:

$$(\mathsf{Out}^{?})\ \frac{P \vdash \textbf{rely}\ A^{\text{-}z} \wedge A_2^{z}\ \textbf{guar}\ A_1^{\vec{y}} \wedge B^{\text{-}\vec{y}} \qquad A \supset A_1 \supset A_2[x\bullet\vec{y}/z]}{\overline{x}(\vec{y}z)P \vdash \textbf{rely}\,A\ \textbf{guar}\ B}$$

The rule is equivalent to the following decomposed version up to $\cong$.

$$(\mathsf{Out}^{?}_{d})\ \frac{S \vdash \textbf{rely}\,A\,\textbf{guar}\,A_1,\ \ P \vdash \textbf{rely}\,A^{\text{-}z} \wedge A_2\,\textbf{guar}\,B,\ \ A \supset A_1 \supset A_2[x\bullet\vec{y}/z]}{\overline{x}(\vec{y}z)(S|P) \vdash \textbf{rely}\,A\ \textbf{guar}\ B}$$

Since this decomposed form gives a more analytical presentation without substantially losing generality, we illustrate the rule using this decomposed version in the following. The rule is again better read downward. Suppose we have:

(1) S, which is to become the argument of a **?**-output at x in the conclusion, guarantees A_1 at $\vec{y}$ relying on A (A may mention x);
(2) If A and A_1 hold, then A_2 (with appropriate substitution) also holds;
(3) P, which receives a linear action at z, can guarantee B relying on A and A_2.

Now assume the environment satisfies A. Then by (1), S can guarantee A_1. When $\overline{x}(\vec{y}z)(S|P)$ invokes x with S as arguments (which, as we already know, satisfies A_1), (2) says that what P would receive via z would surely satisfy A_2. But if so, P can rely on A and A_2, hence can guarantee B by (3). Note the rule assumes A, on which S relies upon in the first step, can still be relied by P, indicating stateless nature of the logic.

(Consequence) $(A \supset A',\ B' \supset B)$	(Aux-$\forall$)	(Aux-$\exists$)
$\dfrac{P \vdash \mathbf{rely}\, A'\ \mathbf{guar}\, B'}{P \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B}$	$\dfrac{P \vdash \mathbf{rely}\, A^{\text{-}i}\, \mathbf{guar}\, B}{P \vdash \mathbf{rely}\, A\, \mathbf{guar}\, \forall i.B}$	$\dfrac{P \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B^{\text{-}i}}{P \vdash \mathbf{rely}\, \exists i.A\, \mathbf{guar}\, B}$

Fig. 3. Proof Rules for Linear Processes (structure rules)

As in Hoare Logic, we use the rule of consequence, which says that if P guarantees B' relying on A', and if P can rely on a stronger condition A, then surely it guarantees a condition B which is weaker than B'.

$$\text{(Consequence)}\ \frac{P \vdash \mathbf{rely}\, A'\ \mathbf{guar}\, B' \quad A \supset A' \quad B' \supset B}{P \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B} \tag{3.11}$$

In the corresponding rule in Hoare Logic, "$\supset$" is about number-theoretic validity: here it is also about equality in the domain of sequential behaviour, for which additional inference rules may as well be stipulated. Two basic rules are:

(extensionality) $\quad \forall \vec{x}.\ u \bullet \vec{x} = v \bullet \vec{x} \quad \supset \quad u = v.$
(data) $\quad \mathtt{in}_i(\vec{x}) = \mathtt{in}_j(\vec{y}) \quad \supset \quad i = j \wedge \vec{x} = \vec{y}.$

We may extend these rules to reach a more comprehensive axiomatisation of the underlying refined applicative structure.

The consequence rule is assisted by the following two rules for auxiliary variables, which are in fact simple instances of Hoare's well-known adaptation rule [46]. These two specific rules are however found to be expressive in the present proof system when combined with the remaining rules. Other structural rules are also useful but many of them may be derivable in the present system. Below $i, \ldots$ exclusively range over on auxiliary variables.

$$\text{(Aux-}\forall\text{)}\ \frac{P \vdash \mathbf{rely}\, A^{\text{-}i}\, \mathbf{guar}\, B}{P \vdash \mathbf{rely}\, A\, \mathbf{guar}\, \forall i.B} \qquad\qquad \text{(Aux-}\exists\text{)}\ \frac{P \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B^{\text{-}i}}{P \vdash \mathbf{rely}\, \exists i.A\, \mathbf{guar}\, B}$$

Recalling auxiliary names are universally quantified in the whole sequent, these rules are immediately valid. Other structure rules are treated in the next subsection.

The following result shows the proof system only infers valid assertions (we believe the converse also holds). The proof of the theorem is straightforward and is found in [53, Section 2].

Theorem 1. (soundness of linear process logic) $\quad P \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B$ *implies* $P \models \mathbf{rely}\, A\, \mathbf{guar}\, B$.

At this point we leave open whether our proof system is complete or not (note completeness implies closure of the rules under $\cong$, which in particular implies subject expansion).

Remark 1. (use of negative names in guarantee formulae) As we mentioned briefly at the end of §3.2, we may allow negative names to occur in the guarantee condition, which is similar to the use of hooked variables in VDM [66]. In this case, the semantics of a judgements is given by calculating the validity of the guarantee formula by interpreting its negative names following their interpretation in the rely formula. Concretely we set $P^{\overline{\Gamma};\Delta} \models \mathbf{rely}\, A\, \mathbf{guar}\, B$ as:

$$R^{\Gamma} \models^{\mathcal{I}} A \quad \supset \quad (P|R)^{\Gamma \cdot \Delta} \models^{\mathcal{I}} B$$

where B is now about Γ and Δ. All proof rules remain as they are. This format of judgement tends to make proofs much shorter than the original one, even though it does not add expressiveness since we can easily translate a judgement in the new format to the original format using fresh auxiliary names.

3.6 Variants of Proof Rules

Below we briefly discuss some of the basic variants of the proof rules.

A More Convenient Version of $(\mathsf{In}^!)$**.** In practice, the following strengthened version of $(\mathsf{In}^!)$ rule, immediately inferable by combining the original rule with $(\mathsf{Aux}\text{-}\forall)$, is almost always more convenient.

$$(\mathsf{In}^!\text{-}\forall)\ \frac{P \vdash \mathbf{rely}\ A^{\text{-}\vec{y}\vec{i}} \wedge B_1^{\vec{y}}\ \mathbf{guar}\ B_2\ ,\quad \forall \vec{y}\vec{i}.(B_1 \supset B_2[x\bullet\vec{y}/z]) \supset B}{!x(\vec{y}z).P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}$$

A More Elementary Version of $(\mathsf{In}^!)$**.** Another variant of $(\mathsf{In}^!)$ may be awkward to use in practice, but is most concise. Below let $\vec{i}$ be fresh.

$$(\mathsf{In}^!\text{-subs})\ \frac{P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}{!x(\vec{y}z).P \vdash \mathbf{rely}\ A[\vec{i}/\vec{y}]\ \mathbf{guar}\ B[x\bullet\vec{i}/z]}$$

This rule is equi-potent to the original rule when combined with the following structural rule:

$$(\mathsf{Entailment})\ \frac{P \vdash \mathbf{rely}\, A \wedge B\, \mathbf{guar}\, C, \qquad \mathsf{fn}(B) \cap \mathsf{fn}(\Gamma) = \emptyset}{P^{\Gamma} \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B \supset C}.$$

Note $\mathsf{fn}(B) \cap \mathsf{fn}(\Gamma) = \emptyset$ indicates only auxiliary names occur in B. It is easy to derive $(\mathsf{In}^!)$ from $(\mathsf{In}^!\text{-subs})$ combined with $(\mathsf{Entailment})$. Conversely, from $P \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B$ we derive $P \vdash \mathbf{rely}\, A[\vec{i}/\vec{y}] \wedge \vec{i} = \vec{y}\, \mathbf{guar}\, B$ by (Conseq). Then by $(\mathsf{In}^!)$ we obtain $!x(\vec{y}z).P \vdash \mathbf{rely}\ A[\vec{i}/\vec{y}]\ \mathbf{guar}\ \forall\vec{y}.(\vec{i} = \vec{y} \supset B)$, from which we get $!x(\vec{y}z).P \vdash \mathbf{rely}\, A[\vec{i}/\vec{y}]\, \mathbf{guar}\, B[\vec{i}/\vec{y}]$ by (Conseq), reaching $(\mathsf{In}^!\text{-subs})$.

Proof Rules for Free Output. The standard presentation of the asynchronous version of the π-calculus [56] (cf. [10]) uses free outputs. Below we list the free output counterpart of ($\mathrm{Sel}^{\uparrow}$) and ($\mathrm{Out}^{?}$). We first present their typing rules.

$$(\mathsf{FSel}^{\uparrow})\quad (\Gamma(y_i) = \overline{\tau_i})\qquad \frac{-}{\vdash \overline{x}\mathtt{in}_i\langle\vec{y}\rangle \triangleright \Gamma,\, x : [\oplus_{i\in I}\tau_i]^{\uparrow}} \qquad\qquad (\mathsf{Out}^{?})\quad (\Gamma(y_i) = \overline{\tau_i},\ \Gamma(x) = (\vec{\tau}\rho)^{?})\qquad \frac{-}{\vdash \overline{x}\langle\vec{y}z\rangle \triangleright \Gamma,\, z : \overline{\rho}}$$

The proof rule for the free selection follows.

$$(\mathsf{FSel}^{\uparrow})\ \frac{-}{\overline{x}\mathtt{in}_i\langle\vec{y}\rangle \vdash \mathbf{rely}\ A[\mathtt{in}_i(\vec{y})^{\uparrow}/x]\ \mathbf{guar}\ A}$$

which says: assuming the behaviour of $\vec{y}$ in A, that assumption is precisely what is guaranteed by the free output of $\vec{y}$ together with the selection. The rule for a free co-replication is given similarly:

$$(\mathsf{FOut}^{?})\ \frac{-}{\overline{x}(\vec{y}z)P \vdash \mathbf{rely}\ A[x \bullet \vec{y}/z]\ \mathbf{guar}\ A}$$

The rule says that the assumptions on negative names (x and $\vec{y}$) are precisely what is ensured by the (unique) positive primary name z in A.

It is notable that these rules for free output, on the one hand, and their counterpart for bound output, on the other, are mutually derivable. For deriving a bound output rule from its free output counterpart, we decompose bound output into free output, parallel composition and hiding (this needs a non-trivial use of (Consequence) rule). The other direction uses the standard encoding of free outputs by copycats [13], and is discussed in detail in the next subsection.

Other Structural Rules. Proof rules are called *structural* if it does not involve change of the syntactic structure of a process (a better name would be *logical rules* since they are purely about logical consequences based on semantics of judgements: it seems however "structural" is more commonly used). Besides those presented in the previous subsection, we list several rules of interest. We believe all these rules are admissible in the present system, that is, their introduction does not add derivability, at least in the proof system for linear processes.

The first pair of rules reflect the polarities of rely/guarantee formulae.

$$(\mathsf{Conj})\ \frac{P^{\Gamma} \vdash \mathbf{rely}\,A\,\mathbf{guar}\,B_i\quad (i = 1, 2)}{P^{\Gamma} \vdash \mathbf{rely}\,A\,\mathbf{guar}\,B_1 \wedge B_2} \qquad (\mathsf{Disj})\ \frac{P^{\Gamma} \vdash \mathbf{rely}\,A_i\,\mathbf{guar}\,B\quad (i = 1, 2)}{P^{\Gamma} \vdash \mathbf{rely}\,A_1 \vee A_2\,\mathbf{guar}\,B}$$

The next pair of rules are close to (Aux-$\forall$) and (Aux-$\exists$).

$$(\mathsf{Aux}\text{-}\wedge)\ \frac{P^{\Gamma} \vdash \mathbf{rely}\,A\,\mathbf{guar}\,B\quad \mathsf{fn}(C) \cap \mathsf{dom}(\Gamma) = \emptyset}{P^{\Gamma} \vdash \mathbf{rely}\,A \wedge C\,\mathbf{guar}\,B \wedge C}$$

$$(\mathsf{Aux}\text{-}\vee)\ \frac{P^{\Gamma} \vdash \mathbf{rely}\,A\,\mathbf{guar}\,B\quad \mathsf{fn}(C) \cap \mathsf{dom}(\Gamma) = \emptyset}{P^{\Gamma} \vdash \mathbf{rely}\,A \vee C\,\mathbf{guar}\,B \vee C}$$

The soundness of these rules is again immediate from the semantics of judgement.

3.7 Reasoning Examples

Simple Agents. We show a couple of concrete inferences using the proof rules. The first example is a derivation of the assertion in (3.6). Let $R \stackrel{\text{def}}{=} u[\&_n \overline{w}\mathtt{in}_{2*n}]$ and $A \stackrel{\text{def}}{=} (y \bullet \varepsilon = \mathtt{in}_n()^{\uparrow} \supset x \bullet y = \mathtt{in}_{2*n}()^{\uparrow})$ below.

1.	$\mathbf{0} \vdash \mathbf{rely}\ \mathtt{in}_n()^{\uparrow} = \mathtt{in}_n()^{\uparrow}\ \mathbf{guar}\ \mathtt{in}_{2*n}()^{\uparrow} = \mathtt{in}_{2*n}()^{\uparrow}$	(Zero), (Conseq)
2.	$\overline{w}\mathtt{in}_{2*n} \vdash \mathbf{rely}\ \mathtt{in}_n()^{\uparrow} = \mathtt{in}_n()^{\uparrow}\ \mathbf{guar}\ w = \mathtt{in}_{2*n}()^{\uparrow}$	(Sel)
3.	$u[\&_n \overline{w}\mathtt{in}_{2*n}()^{\uparrow}] \vdash \mathbf{rely}\ u = \mathtt{in}_n()^{\uparrow}\ \mathbf{guar}\ w = \mathtt{in}_{2*n}()^{\uparrow}$	(Bra)
4.	$R \vdash \mathbf{rely}\, y \bullet \varepsilon = \mathtt{in}_n()^{\uparrow} \wedge u = \mathtt{in}_n()^{\uparrow}\, \mathbf{guar}\ w = \mathtt{in}_{2*n}()^{\uparrow}$	(Conseq)
5.	$\overline{y}(u)R \vdash \mathbf{rely}\ y \bullet \varepsilon = \mathtt{in}_n()^{\uparrow}\ \mathbf{guar}\ w = \mathtt{in}_{2*n}()^{\uparrow}$	(Out)
6.	$!x(yw).\overline{y}(u)R \vdash \mathbf{rely}\ \top\ \mathbf{guar}\ \forall y.(y \bullet \varepsilon = \mathtt{in}_n()^{\uparrow} \supset x \bullet y = \mathtt{in}_{2*n}()^{\uparrow})$	(In)
7.	$!x(yw).\overline{y}(u)R \vdash \mathbf{rely}\ \top\ \mathbf{guar}\ \forall y, n.\, A$	(Aux-$\forall$)

Since "$\mathtt{in}_n()^{\uparrow} = \mathtt{in}_n()^{\uparrow}$" and "$\mathtt{in}_{2*n}()^{\uparrow} = \mathtt{in}_{2*n}()^{\uparrow}$" are trivially valid, Line 1 is immediate. Lines 2 an 3 are direct from the rules. In Line 4, we add $y \bullet \varepsilon = \mathtt{in}_n()^{\uparrow}$ in the rely formula by (Conseq). In Line 5, we apply (Out$^?$), using the following tautology:

$$y \bullet \varepsilon = \mathtt{in}_n()^{\uparrow}\ \wedge\ \top\ \supset\ (u = \mathtt{in}_n()^{\uparrow})[y \bullet \varepsilon / u]$$

In Line 6, we apply (In$^!$). Note y is auxiliary in the conclusion. In Line 7, we apply (Aux-$\forall$) to the auxiliary names y and n. Note lines 6 and 7 can be done in one step using (In$^!$-$\forall$) discussed in Section 3.6. In the following we shall be always using (In$^!$-$\forall$) for reasoning about replication.

Parallel Composition. The next example shows a simple usage of (Par) rule. Below let $S_{xy} \stackrel{\text{def}}{=} !x(w).\overline{y}(u)u[\&_{n\in\mathbb{N}}.\overline{w}\mathtt{in}_{2*n}]$. We first observe:

$$S_{xy} \vdash \mathbf{rely}\ z \bullet \varepsilon = \mathtt{in}_6()^{\uparrow}\ \mathbf{guar}\ \forall y, n.\, (y \bullet \varepsilon = \mathtt{in}_n()^{\uparrow} \supset x \bullet y = \mathtt{in}_{2*n}()^{\uparrow})$$

by only abstracting w in Line 6 of the inference above. We can now infer:

1.	$S_{xz} \vdash \mathbf{rely}\ z \bullet \varepsilon = \mathtt{in}_6()^{\uparrow}\ \mathbf{guar}\ x \bullet \varepsilon = \mathtt{in}_{12}()^{\uparrow}$	
2.	$S_{yz} \vdash \mathbf{rely}\ y \bullet \varepsilon = \mathtt{in}_3()^{\uparrow}\ \mathbf{guar}\ z \bullet \varepsilon = \mathtt{in}_6()^{\uparrow}$	
3.	$S_{xz} \vert S_{zy} \vdash \mathbf{rely}\ y \bullet \varepsilon = \mathtt{in}_3()^{\uparrow}\ \mathbf{guar}\ x \bullet \varepsilon = \mathtt{in}_{12}()^{\uparrow}$	(1, 2, Par)

where (1, 2, Par) indicates Lines 1 and 2 are used as antecedents for the inference of (Par).

Copycat and Free Output. As the third example, we establish a basic property of copy-cats. The copy-cat for the type $(()^{\uparrow})^!$ already appeared in Example

2-(4), page 18: the following extends the construction to arbitrary input types, using representation of free name outputs by copy-cats [13, 106].

$$
\begin{aligned}
[x \to x']^{(\vec{\tau}\rho)^!} &\stackrel{\text{def}}{=} !x(\vec{y}z)\overline{x'}\langle\vec{y}z\rangle^{\vec{\tau}\rho}\\
[x \to x']^{[\&_i\vec{\tau}_i]^\downarrow} &\stackrel{\text{def}}{=} x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i\langle\vec{y}_i\rangle^{\vec{\tau}_i}]\\
\overline{x'}\langle\vec{y}z\rangle^{\vec{\tau}\rho} &\stackrel{\text{def}}{=} \overline{x'}(\vec{y'}z')(\Pi_i[y'_i \to y_i]^{\overline{\tau_i}} \mid [z' \to z]^{\overline{\rho}})\\
\overline{x'}\mathtt{in}_i\langle\vec{y}\rangle^{\vec{\tau}} &\stackrel{\text{def}}{=} \overline{x'}\mathtt{in}_i(\vec{y'})\Pi_i[y'_i \to y_i]^{\overline{\tau_i}}.
\end{aligned}
$$

Copycats forward a behaviour from one place to another, thus linking two locations. Their origins can be traced back to at least three traditions: *forwarder* in Hewitt's actors [7, 44], *link agent* in CCS/π-calculus [57, 78, 106], and *chit-for-chat* in game-based models of Intuitionistic Logic. This is a basic behaviour in many computing scenes, ranging from routers in the network to variables in the λ-calculus and to dynamic linkage in operating systems. From its behaviour, we may expect that, in the specification of a copy-cat, any property of the environment is copied to that of the process, similarly for free outputs. Thus we arrive at the following assertions (assuming well-typedness as always).

$$[x \to y]^{\tau^!} \vdash \textbf{rely } A[y/x] \textbf{ guar } A \tag{3.12}$$

$$[x \to y]^{\tau^\downarrow} \vdash \textbf{rely } A[x/y] \textbf{ guar } A \tag{3.13}$$

$$\overline{x}\langle\vec{y}z\rangle^{\vec{\rho}\tau} \vdash \textbf{rely } A[x \bullet \vec{y}/z] \textbf{ guar } A \tag{3.14}$$

$$\overline{x}\mathtt{in}_i\langle\vec{y}\rangle^{\vec{\rho}} \vdash \textbf{rely } A[\mathtt{in}_i(\vec{y})/x] \textbf{ guar } A \tag{3.15}$$

Note the last two lines are precisely the axioms for free outputs given in §3.6, Page 35. We also use a strengthened version of (3.12), writing $[\vec{x} \to \vec{y}]$ for $\Pi_i[x_i \to y_i]$.

$$[\vec{x} \to \vec{y}] \vdash \textbf{rely } A[\vec{y}/\vec{x}] \textbf{ guar } A \tag{3.16}$$

In the following we infer these properties using the proof rules. We first infer (3.14) from (3.16) and (3.13).

1.	$[\vec{y'} \to \vec{y}] \vdash \textbf{rely } A[x \bullet \vec{y}/z] \textbf{ guar } A[x \bullet \vec{y'}/z]$	(from 3.16)
2.	$[z' \to z] \vdash \textbf{rely } A[x \bullet \vec{y}/z] \wedge A[z'/z] \textbf{ guar } A$	(from 3.13)
3.	$\overline{x}(\vec{y'}z')([\vec{y'} \to \vec{y}] \mid [z' \to z]) \vdash \textbf{rely } A[x \bullet \vec{y}/z] \textbf{ guar } A$	(1,2,Out)

Next we derive (3.15) from (3.16).

1.	$[\vec{y'} \to \vec{y}] \vdash \textbf{rely } A[\mathtt{in}_i(\vec{y})/x] \textbf{ guar } A[\mathtt{in}_i(\vec{y'})/x]$	(from 3.16)
2.	$\overline{x}\mathtt{in}_i(\vec{y'})[\vec{y'} \to \vec{y}] \vdash \textbf{rely } A[\mathtt{in}_i(\vec{y})/x] \textbf{ guar } A$	(Sel)

The following is the inverse of the above inference, deriving (3.13) from (3.14).

1.	$\forall i.\overline{x'}\mathtt{in}_i\langle\vec{y}_i\rangle \vdash \textbf{rely } A[x'/x][\mathtt{in}_i(\vec{y}_i)/x'] \textbf{ guar } A$	(from 3.16)
2.	$x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i\langle\vec{y}_i\rangle] \vdash \textbf{rely } A[x'/x] \textbf{ guar } A$	(Bra)

A most complicated case is to infer (3.12) from (3.14) and (3.15), let $A' \stackrel{\text{def}}{=} A[i/x] \wedge z = i \bullet \vec{j}$ with $i\vec{j}$ fresh.

1.	$\overline{x'}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ x'=i \wedge A'[x' \bullet \vec{y}/z]\ \mathbf{guar}\ A'$	(from 3.16)
2.	$\overline{x'}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ x'=i \wedge A[i/x] \wedge \vec{y}=\vec{j}\ \mathbf{guar}\ A'$	(Conseq)
3.	$!x(\vec{y}z).\overline{x'}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ x'=i \wedge A[i/x]\ \mathbf{guar}\ \forall\vec{y}\vec{j}.(\vec{y}=\vec{j} \supset A'[x \bullet \vec{y}/z])$	(In-$\forall$)
4.	$!x(\vec{y}z).\overline{x'}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ x'=i \wedge A[i/x]\ \mathbf{guar}\ A[i/x] \wedge \forall\vec{j}.x \bullet \vec{j}=i \bullet \vec{j}$	(Conseq)
5.	$!x(\vec{y}z).\overline{x'}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ x'=i \wedge A[i/x]\ \mathbf{guar}\ \exists i.(A[i/x] \wedge x=i)$	(Conseq)
6.	$!x(\vec{y}z).\overline{x'}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ \exists i.(x'=i \wedge A[i/x])\ \mathbf{guar}\ \exists i.(A[i/x] \wedge x=i)$	(Aux-$\exists$)
7.	$!x(\vec{y}z).\overline{x'}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ A[x'/x]\ \mathbf{guar}\ A$	(Conseq)

Note we used extensionality in Line 5. In Line 3, $\vec{y}=\vec{j}$ stands for the conjunction of the equation for each name.

Finally we derive (3.16) by induction on the length of the string, using (3.12) as the base case.

1.	$[\vec{x} \to \vec{x'}] \vdash \mathbf{rely}\ A[\vec{x'}\vec{i}/\vec{x}\vec{y}]\ \mathbf{guar}\ A[\vec{i}/\vec{y}]$	(Ind. Hyp.)
2.	$[\vec{y} \to \vec{y'}] \vdash \mathbf{rely}\ \vec{y'} = \vec{i}\ \mathbf{guar}\ \vec{y} = \vec{i}$	(Ind. Hyp.)
3.	$[\vec{x}\vec{y} \to \vec{x'}\vec{y'}] \vdash \mathbf{rely}\ A[\vec{x'}\vec{i}/\vec{x}\vec{y}] \wedge \vec{y'}=\vec{i}\ \mathbf{guar}\ A[\vec{i}/\vec{y}] \wedge \vec{y}=\vec{i}$	(1,2,Par)
4.	$[\vec{x}\vec{y} \to \vec{x'}\vec{y'}] \vdash \mathbf{rely}\ A[\vec{x'}\vec{i}/\vec{x}\vec{y}] \wedge \vec{y'}=\vec{i}\ \mathbf{guar}\ \exists\vec{i}.(A[\vec{i}/\vec{y}] \wedge \vec{y}=\vec{i})$	(Conseq)
5.	$[\vec{x}\vec{y} \to \vec{x'}\vec{y'}] \vdash \mathbf{rely}\ \exists\vec{i}.(A[\vec{x'}\vec{i}/\vec{x}\vec{y}] \wedge \vec{y'}=\vec{i})\ \mathbf{guar}\ \exists\vec{i}.(A[\vec{i}/\vec{y}] \wedge \vec{y}=\vec{i})$	(Aux-$\exists$)
6.	$[\vec{x}\vec{y} \to \vec{x'}\vec{y'}] \vdash \mathbf{rely}\ A[\vec{x'}\vec{y'}/\vec{x}\vec{y}]\ \mathbf{guar}\ A$	(Conseq)

This concludes the proof of the assertions (3.12)–(3.15).

3.8 Linear Process Logic and Contextual Congruence

In this subsection we shall show the set of valid assertions for a process precisely characterise its behaviour up to the standard contextual congruence for linear processes. In other words, the assertions in the logic differentiate no more and no less than the behavioural equality of processes. From a philosophical viewpoint, this shows two fundamental ways to articulate behaviour of processes — one, the process equivalence, by identifying what it means for two behaviours to be identical; and the other, the process logic, by describing individual behavioural properties of processes — are in harmony with each other, clarifying the semantic status of the process logic we set up in the present inquiries. The same property holds in each process logic we shall investigate in the present work. Below we define $P^\Gamma =_\mathcal{L} Q^\Gamma \ \equiv\ \mathcal{L}(P^\Gamma) = \mathcal{L}(Q^\Gamma)$ with $\mathcal{L}(P^\Gamma) \stackrel{\text{def}}{=} \{\langle A, B\rangle \mid P^\Gamma \models \mathbf{rely}\ A\ \mathbf{guar}\ B\}$.

Theorem 2. *$P^\Gamma \cong Q^\Gamma$ if and only if $P^\Gamma =_\mathcal{L} Q^\Gamma$.*

The proof of Theorem 2 uses the following result. Below we let $\mathcal{M}(P^{\Gamma}) \stackrel{\text{def}}{=} \{(\xi,\eta) \mid \forall R^{\Gamma} \models_{\mathsf{b}} \xi.(\boldsymbol{\nu}\,\mathsf{fn}(\Gamma))(P|R) \models_{\mathsf{b}} \eta\}$.

Lemma 1. *$P^{\Gamma} =_{\mathcal{L}} Q^{\Gamma}$ iff $\mathcal{M}(P^{\Gamma}) = \mathcal{M}(Q^{\Gamma})$ if $P \cong Q$.*

Proof. The first half is immediate from the definition of $\models^{\mathcal{I}}$, while the second half is direct from Lemma 4 (1). $\square$

Lemma 1 already gives the "only if" direction of Theorem 2. For the "if" direction, write $\mathcal{F}^{\overline{\Gamma};\Delta}$ for a function from the models of type Γ to those of type Δ. Note $\mathcal{M}(P^{\Gamma})$ for each P defines such a function. We now define the actions on such functions induced by well-typed contexts.

$$
\begin{array}{rcl}
x[\&_i(\vec{y}_i).\mathcal{F}_i^{\overline{\Gamma\cdot\vec{y}_i:\vec{\rho}_i};v:\sigma}] : \xi^{\Gamma}\cdot x:\mathtt{in}_i(\vec{p}) \mapsto v:\gamma & \equiv & \mathcal{F}_i : \vec{y}_i:\vec{p} \mapsto v:\gamma \\
\overline{x}\mathtt{in}_i(\vec{y})\mathcal{F}^{\overline{\Gamma};\vec{y}:\vec{\rho}} : \xi^{\Gamma} \mapsto x:\mathtt{in}_i(\vec{p})^{\uparrow} & \equiv & \mathcal{F} : \xi \mapsto \vec{y}:\vec{p} \\
!x(\vec{y}z).(\mathcal{F}^{\overline{\Gamma,\vec{y}:\vec{\tau}};z:\rho}) : \xi^{\Gamma} \mapsto x:p & \equiv & \mathcal{F} : \xi\cdot\vec{y}:\vec{q}_i \mapsto z:p\bullet\vec{q}_i \\
\overline{x}(\vec{y}z)(\mathcal{F}^{\overline{\Gamma};\vec{y}:\vec{\tau}},\ \mathcal{G}^{\overline{\Gamma,z:\rho};v:\sigma}) : \xi^{\Gamma} \mapsto v:\gamma & \equiv & \mathcal{F} : \xi \mapsto \vec{y} : \vec{p}_i\ \wedge \\
 & & \mathcal{G} : \xi\cdot z:\xi(x)(\vec{p}_i) \mapsto v:\gamma \\
\mathcal{F}_1^{\overline{\Gamma_1};\Delta_1\cdot\Theta}|\mathcal{F}_2^{\overline{\Gamma_2\cdot\Theta};\Delta_2} : \xi_1^{\Gamma_1}\cdot\xi_2^{\Gamma_2} \mapsto \eta_1^{\Delta_1}\cdot\eta_2^{\Delta_2}\cdot\varsigma^{\Theta\odot\overline{\Theta}} & \equiv & \mathcal{F}_1 : \xi_1 \mapsto \eta_1\cdot\varsigma'\ \wedge \\
 & & \mathcal{F}_2 : \xi_2\cdot\varsigma' \mapsto \eta_2\ \wedge\ \varsigma'\odot\overline{\varsigma'} = \varsigma \\
(\boldsymbol{\nu}\,x)(\mathcal{F}^{\overline{\Gamma};\Delta\cdot x:\tau^{!}}) : \xi^{\Gamma} \mapsto \eta^{\Delta} & \equiv & \mathcal{F} : \xi^{\Gamma} \mapsto \eta^{\Delta}\cdot x:\gamma
\end{array}
$$

($\varsigma'\odot\overline{\varsigma'} = \varsigma$ is defined just as $\Theta\odot\overline{\Theta}$). We can easily check each action results in a function of an appropriate type. Given a well-typed context $C[\ ]$, we write $C[\mathcal{F}]$ for the corresponding compound action. We note:

Lemma 2. (1) $\mathcal{M}(C[P^{\Gamma}]) = C[\mathcal{M}(P^{\Gamma})]$. (2) $\mathcal{M}(P^{x:[\oplus_{i\in\mathbb{B}}]^{\uparrow}}) = \mathcal{M}(Q^{x:[\oplus_{i\in\mathbb{B}}]^{\uparrow}})$ *implies* $P\Downarrow_x^{\mathbf{t}} \equiv Q\Downarrow_x^{\mathbf{t}}$.

Proof. Both are direct from the definition of $\models_{\mathsf{b}}$. $\square$

We can now infer:

$$
\begin{array}{rcll}
P^{\Gamma} \cong_{\mathcal{L}} Q^{\Gamma} & \supset & \mathcal{M}(P^{\Gamma}) = \mathcal{M}(Q^{\Gamma}) & \text{(Lemma 1)} \\
 & \supset & \forall C[\,\cdot\,]_{\Gamma}^{x:[\oplus_{i\in\mathbb{B}}]^{\uparrow}}.\ (C[\mathcal{M}(P)] = C[\mathcal{M}(Q)]) & \\
 & \supset & \forall C[\,\cdot\,]_{\Gamma}^{x:[\oplus_{i\in\mathbb{B}}]^{\uparrow}}.\ (\mathcal{M}(C[P]) = \mathcal{M}(C[Q])) & \text{(Lemma 2-(1))} \\
 & \supset & \forall C[\,\cdot\,]_{\Gamma}^{x:[\oplus_{i\in\mathbb{B}}]^{\uparrow}}.\ (C[P]\Downarrow_x^{\mathbf{t}} \equiv C[Q]\Downarrow_x^{\mathbf{t}}) & \text{(Lemma 2-(2))} \\
 & \supset & P^{\Gamma} \cong Q^{\Gamma}. &
\end{array}
$$

We have now established Theorem 2. Other open issues include the existence of characteristic formulae for processes modulo $\cong$.

$$\frac{(\mathsf{Par})\quad (\Theta_1, \Theta_2 \odot \overline{\Theta_1}, \overline{\Theta_2} = \Theta') \qquad \vdash P_1 \triangleright \Gamma_1, \overline{\Theta_2}\,;\, \Delta_1, \Theta_1 \quad \vdash P_2 \triangleright \Gamma_2, \overline{\Theta_1}\,;\, \Delta_2, \Theta_2}{\vdash P_1|P_2 \triangleright \Gamma_1 \cup \Gamma_2\,;\, \Delta_1,\, \Delta_2,\, \Theta'}$$

Fig. 4. Affine (Par) Rule

4 Logics for Stateless Processes with Divergence (1)

4.1 Circular Composition and General Recursion

On the basis of the logic for linearly typed processes, we next explore how the incorporation of unrestricted recursion, which is a source of general computing power in programming languages (in the shape of both loop and recursion), affects the process logic. At the level of processes, the introduction of recursion can be done in two distinct, but equivalent, ways: one, by relaxing the typing rule for the parallel composition; and, two, by adding a specific rule for recursion. These two forms yield an identical class of behaviour. Since these two forms shed different light on the logical analysis of recursive behaviour, we treat both forms in the present section. The resulting class of behaviour is called *affine processes*, for which we present three logical articulations: one, the logic for total correctness (in the standard sense of Hoare logic); two, the logic for partial correctness (again in the standard sense); and a generalised version of total correctness. The semantics and proof rules for these three variants of the affine process logics are closely related and together offer the basis of applications.

Affine processes use the same set of syntax for processes and types. In Figure 4 above, we present the relaxed (Par) rule for affine processes. In comparison with (Par) rule in Figure 1, it now allows circular composition of processes. All other typing rules for affine processes are from Figure 1. Since the new (Par) rule subsumes the original rule, the class of affine processes include the class of linearly typed processes as its subset: as can be easily observed, this inclusion is strict (a basic witnessing process is a circular wire, presented in Example 3 (1) below). The extension of the class of behaviour enables, for example, fully abstract embeddings of call-by-name and call-by value PCF in processes. To observe the connection between recursion and circular composition in the affine (Par) rule, the following notation is useful.

Definition 3. (recursion through circular composition [41]) Let P be such that $\vdash P \triangleright \mathbf{?}\Gamma,\ \vec{y}:\overline{\vec{\tau}},\ \vec{x}:\vec{\tau}^{!}$ with $\vec{x} = x_0..x_{n-1}$ and $\vec{y} = y_0..y_{n-1}$ $(n \geq 0)$. Then we define: $\mu\vec{y}{=}\vec{x}.P \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, y)(P|\Pi_{0\leq i\leq n-1}[x_i \to y_i]^{\tau_i})$.

Note $\vdash \mu\vec{y}{=}\vec{x}.P \triangleright \Gamma, \vec{x} : \vec{\tau}$. Using the notation, below we illustrate affine typing by a couple of examples.

$$\frac{\begin{array}{l}(\mathsf{Rec})\\ \vdash P \triangleright \Gamma, \vec{y}{:}\overline{\vec{\tau}}\,;\, \vec{x}{:}\vec{\tau}^{!}\end{array}}{\vdash \mu\vec{y}{=}\vec{x}.P \triangleright \Gamma, \vec{x}{:}\vec{\tau}}$$

Fig. 5. Typing Rule for Recursion

Example 3. 1. *Lazy divergence.* $\Omega_u^\tau \stackrel{\text{def}}{=} \mu x = u.[u \to x]^\tau$, which is, by definition, the same thing as $(\boldsymbol{\nu}\, x)([u \to x]^\tau|[x \to u]^\tau)$ (this agent already appeared in Example 1). Then $\vdash \Omega_u^\tau \triangleright u : \tau$. We often omit τ, writing Ω_u. Note we have $\Omega_u|\overline{u}(\vec{z})P \longrightarrow \Omega_u|\overline{u}(\vec{z})P$. We call Ω_u, a *lazy divergent process.*

2. *Eager divergence.* Let $\omega_u^{\tau^\uparrow} \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, w)(\Omega_w^{(()^\uparrow)^!}|\overline{w}(c)c(u).R)$ where $\vdash R \triangleright x : \tau$. Then $\vdash \omega_u \triangleright u : \tau$. Again we often omit τ. Note $\omega_u \longrightarrow \omega_u$. The process ω_u is called an *eager divergent process*, or simply a *divergent process.*
3. *Factorial.* Let i range over natural numbers and $[\![i]\!]_u \stackrel{\text{def}}{=} !u(c).\overline{c}\mathtt{in}_i$. We set:

$$\mathsf{Fact}_f \stackrel{\text{def}}{=} \mu g{=}f.\ (\,!f(xz).\overline{x}(c)c[\&_i P_i]\,) \tag{4.1}$$

where $P_0 \stackrel{\text{def}}{=} \overline{z}\mathtt{in}_1$ and $P_{i+1} \stackrel{\text{def}}{=} \overline{g}(x'z')([\![i]\!]_{x'}|z'[\&_j\overline{z'}\mathtt{in}_{i\times j}])$. This is a familiar recursive factorial program in the process form.

4.2 Alternative Affine Typing: Incremental Approach

In the previous subsection we defined recursion $\mu x{=}y.P$ using circular composition made possible by the relaxed (Par) rule. But it is easy to see there is a reverse transformation. Below $\approx$ is the standard untyped weak bisimilarity.

Proposition 1. *Let* $\vdash P_1 \triangleright \Gamma_1, \overline{\Theta_2}\,;\, \Delta_1, \Theta_1$ *and* $\vdash P_2 \triangleright \Gamma_2, \overline{\Theta_1}\,;\, \Delta_2, \Theta_2$ *such that* $\mathsf{fn}(\Theta_2) = \{\vec{x}\}$. *Then* $P_1|P_2 \ \approx\ \mu\vec{y}{=}\vec{x}.(\boldsymbol{\nu}\,\vec{y})(P_1|P_2[\vec{y}/\vec{x}])$ *with* $\vec{y}$ *fresh.*

Proof. Because $(\boldsymbol{\nu}\, y)(P|[y \to z]^\tau) \approx P[z/y]$ with fresh z, which in turn is proved by induction on τ together with $(\boldsymbol{\nu}\, y)([x \to y]|!y(\vec{z}).P) \approx !x(\vec{z}).P$ for x fresh. □

Observing the composition $P_1|P_2[\vec{y}/\vec{x}]$ above follows the original (Par) rule (since two processes now only intersect at $\mathsf{fn}(\Theta_1)$), Proposition 1 means we can generate the affine behaviour if the recursive prefix $\mu\vec{y} = \vec{x}.P$ is treated as a primitive construct and is added to the linear typing. This construct obeys the natural structural law and the following reduction rule.

$$\mu\vec{y}{=}\vec{x}.(!x_i(\vec{z}).P\,|\,\overline{y_i}(\vec{z})Q\,|\,R) \longrightarrow \mu\vec{y}{=}\vec{x}.(!x_i(\vec{z}).P\,|\,(\boldsymbol{\nu}\,\vec{z})(P|Q)\,|\,R)$$

We also close the reduction under the recursive prefix. Formal syntactic details of this construct, including labelled transition, are listed in Appendix A.3. The typing rule for this alternative presentation of recursion is listed in Figure 5. The same linear typing rules of Figure 5 are used for all other constructs, including parallel composition. This alternative articulation gives a more analytical view of affine typing, which we shall find useful for the present inquiry.

4.3 Three Process Logics for Affine Behaviour

Before embarking on logical articulation of affine behaviour, let us illustrate briefly how the introduction of potential divergence leads to different ways to extend the logic in Section 3. In fact there are three distinct extensions.

1. *The logic for total correctness*, in which a valid non-trivial assertion indicates the process in question terminates. For brevity this logic will sometimes be called *total affine logic*.
2. *The logic for generalised total correctness*, in which an assertion can specify the whole of affine process behaviour, including divergent computation. Validity/provability does not entail termination, but when so specified, it ensures the process in question terminates. This logic is sometimes called *general affine logic*.
3. *The logic for partial correctness*, in which assertions do not take termination into account (if a process does not terminate, it can be given an arbitrary specification). We shall call this logic *partial affine logic* for brevity.

Historically, Hoare presented his logic in the form of partial correctness, whose extensions to total correctness are later studied in various ways. A practical wisdom from a long academic and engineering experience would be that a validity proof of an assertion for a program is carried out for partial correctness first, to which we add a termination proof.

A program logic for total correctness however has significance in that, from an engineering viewpoint, total correctness offers a basic criteria for correctness of sequential programs (note it is hard to imagine the use of divergent behaviour in the sequential, non-interactive computation). Further, in the present setting, the semantics and proof rules of the logic for total correctness arise as simple increment over those of the linear process logic, which makes this logic an ideal starting point for introducing affine logics.

Finally what we call generalised total correctness can specify and ensure the whole of affine process behaviours, which includes both terminating and possibly divergent ones. Since partiality — or possibility of divergence — is essential part of the affine behaviour, the universe of logic may as well include the phenomenon in its most general form. Further having partiality in the logic does not let us lose any precision since the logic does ensure termination when so specified in an assertion, while we can also specify we tolerate (or even guarantee) divergence. Thus the general affine logic subsumes both the partial logic and the total logic, though at the cost of verbosity in assertions. There are also significant practical applications of the generality of this logic, including logical justification of compositional program analyses (which usually does not exclude potentially non-terminating programs from the targets of analyses).

In the rest of the present section, we shall introduce the total affine process logic, presenting its semantics and proof rules followed by simple inference examples. The next section first presents the logic for partial correctness, then move to the logic which integrates two, the general affine logic.

4.4 Total Affine Logic (1): Syntax and Semantics

The logical language for the total logic remains the same as the linear process logic except we add one constant denoting the divergence at each $\uparrow$-type. Let $\mathsf{md}(\rho) = \uparrow$ in the second line.

$$
\begin{array}{rcl}
e & ::= & * \mid \mathbf{t} \mid \mathbf{f} \mid n \mid x^{\mathsf{nat}} \mid x^{\mathsf{bool}} \mid e + e' \mid e \wedge e' \mid \ldots \\
a & ::= & x^{\tau} \mid a \bullet \vec{b} \mid \mathtt{in}^{\rho}_{e}(\vec{a})^{\uparrow} \mid \omega^{\rho} \\
A & ::= & e_1 = e_2 \mid a_1 = a_2 \mid \neg A \mid A_1 \star A_2 \mid \mathcal{Q}i.A \mid \mathcal{Q}x.A
\end{array}
$$

The well-typedness is defined just as before, based on which we only consider well-typed expressions from now on. We usually omit ρ in ω^{ρ}. ω denotes divergence: While the total logic is mainly about terminating processes so that totality is often implicit in specifications, the logic does incorporate partiality "under replicated prefix" (in the sense to be made clear later), so that it does need to mention convergence/divergence explicitly in some situations. For discussing convergence, it is convenient to use the following notation.

$$a^{\rho} \Downarrow \quad \equiv \quad \exists \vec{y}i.\,(a = \mathtt{in}_i(\vec{y})^{\uparrow}).$$

For example, given x and $\vec{y}$, $x \bullet \vec{y} \Downarrow$ means that the result of invoking x with $\vec{y}$ does return some answer. Since $a \Downarrow$ can also be written $\neg(a = \omega)$, the constant ω is in fact redundant, even though having it is harmless and is sometimes convenient.

The notion of *abstract processes* is defined precisely as in Section 3 (see §3.4, Page 30), starting from the contextual congruence $\cong$ on affine processes defined by the same clauses for linear processes (we can equivalently use a simpler observable type, $()^{\uparrow}$, for defining $\cong$). We write $p \Uparrow$ if some (hence every) process in p has an infinite reduction sequence. $p \Downarrow$ if $\neg(p \Uparrow)$. If $p \Downarrow$ we say p is *total.* Note all abstract processes of a replicated type τ are total. Note also there is one and only one abstract process p satisfying $p \Uparrow$ in each $\uparrow$-type τ: we shall be writing such a process ω^{τ} or ω. A *model* is a finite map from names to abstract processes. In the total logic, we only use total models in the following sense.

Definition 4. *ξ is* total *iff it only uses total abstract processes, i.e. iff $\omega \notin \mathsf{ran}(\xi)$ ($\mathsf{ran}(\xi)$ denotes the range of ξ).*

The total logic ensures totality in the sense that it preclude, from its domain of discourse, immediately diverging processes. However a lazy divergence is safely included among total processes since, intuitively, it *is* harmless as far as no invocation takes place.

The definition of the satisfaction relation is precisely as in Section 3 (cf. §3.4, Page 30) except that ξ in $\xi \models^{\mathcal{I}} A$ should be total. We however *do* allow the range of $\mathcal{I}$ to contain immediately diverging behaviours: while we require all primary names to be total, some of them can induce, upon invocation, divergence, hence terms do contain divergent behaviours. $P \models^{\mathcal{I}} A$ stand for $\xi.\,(P \models_{\mathsf{b}} \xi$ and $\xi \models^{\mathcal{I}} A$ for some (total) ξ. The validity of an assertion $P^{\Gamma} \models \mathbf{rely}\,A\,\mathbf{guar}\,B$ is then defined just as before, which we again reproduce:

$$P^{\overline{\Gamma};\Delta} \models \mathbf{rely}\,A\,\mathbf{guar}\,B \;\equiv\; \forall R^{\Gamma}.\,(R^{\Gamma} \models^{\mathcal{I}} A \;\supset\; (\boldsymbol{\nu}\,\mathsf{fn}(\Gamma))(P|R) \models^{\mathcal{I}} B)\;. \quad (4.2)$$

Thus an assertion $P^{\Gamma} \models \textbf{rely}\, A\, \textbf{guar}\, B$ demands (unless it is vacuous by A being equivalent to F) not only the process but also the environment is terminating. The verification of the following statement shows this is indeed prerequisite for the consistency of this logic to be total. Below we write $P \Downarrow$ for P not having an infinite series of reductions:

Proposition 2. *Let $\Gamma;;\Theta \vdash A$ be satisfiable, i.e. $\xi \models^{\mathcal{I}} A$ for some ξ and I. Then $P^{\overline{\Gamma};\Delta} \models^{I} \textbf{rely}\, A\, \textbf{guar}\, B$ implies $P \Downarrow$.*

Proof. Fix an interpretation I. If R^{Γ} satisfies A, then $(\boldsymbol{\nu}\, \mathsf{fn}(\Gamma))(P|R)$ should satisfy B, that is it should define a terminating behaviour. But if $(\boldsymbol{\nu}\, \mathsf{fn}(\Delta))(P|R)$ terminates then P should also terminate. $\square$

In spite of Proposition 2, the expressible properties in the present logic include those for partial correctness. For example, the following formula (assuming x is a primary name) gives, in the standard partial ordering of behaviours, a set of points which are closed downward:

$$x \bullet \varepsilon = \omega \;\vee\; x \bullet \varepsilon = 1,$$

which is satisfied by, for example, $!x(c).\overline{c}\mathtt{in}_1$ as well as $!x(c).\omega_c$. For reference we give a typical example of a property which is closed upward.

$$\forall y.\, (y \bullet \varepsilon = \mathtt{in}_1 \supset x \bullet y = 2) \;\wedge\; (y \bullet \varepsilon = \mathtt{in}_3 \supset x \bullet y = 6),$$

which is satisfied, in the present logic, by all replicated processes which include the above interactions (i.e. which transform 1 and 3 to 2 and 6, respectively) as part of their behaviours.

A significant property of the logic (which is related to the expressiveness of assertions noted above) is that the valid assertions in the logic gives the same equational characterisation as we have seen in Section 3:

Theorem 3. *$P^{\Gamma} \cong Q^{\Gamma}$ if and only if $P^{\Gamma} =_{\mathcal{L}} Q^{\Gamma}$ in the total affine logic.*

Proof. We observe, in the proof of Theorem 2 (§3.8), we can consider each $\mathcal{F}$ to be a typed process context without loss of generality, since $\mathcal{M}(P^{\overline{\Gamma};\Delta})$ is in fact characterised by the context $(\boldsymbol{\nu}\, \mathsf{fn}(\Gamma))(P|[\,\cdot\,])$ (up to $\cong$). Note the typed actions on a typed context yields another typed context (e.g. $!x(\vec{y}z).C[\,\cdot\,]$). Thus we can safely restrict the functions considered to typed contexts, adding:

$$\mu\vec{y}=\vec{x}.(\mathcal{F}^{\overline{\Gamma},\vec{y}:\vec{\tau};\vec{x}\vec{\tau}^{!}}) \quad \stackrel{\text{def}}{=} \quad \mu\vec{y}=\vec{x}.C[\,\cdot\,]^{\vec{x}:\vec{\tau}^{!}}_{\overline{\Gamma},\vec{y}:\vec{\tau}} \qquad (\mathcal{F} \text{ characterised by } C[\,\cdot\,])$$

Let $\vdash P \triangleright \overline{\Gamma}, \vec{y}:\vec{\tau}; \vec{x}:\vec{\tau}^{!}$. By definition we have, for each R^{Γ}:

$$\mu\vec{y}=\vec{x}.(\boldsymbol{\nu}\, \mathsf{fn}(\Gamma))(P|R) \quad \equiv \quad (\boldsymbol{\nu}\, \mathsf{fn}(\Gamma))(\mu\vec{y}=\vec{x}.P|R)$$

hence we immediately know $\mathcal{M}(\mu\vec{y}=\vec{x}.P) \;\equiv\; \mu\vec{y}=\vec{x}.\mathcal{M}(P)$. The rest precisely follows the development in §3.8, noting a divergent process P is characterised by a pair of rely/guarantee formulae (F, F). $\square$

The other notable property of the affine process logic is that it is a conservative extension of the linear process logic in that essentially the same set of assertions are valid in two logics for linearly typed processes.

$$
(\mathsf{Out}^?)\quad \dfrac{(A \supset A_1 \supset A_2[x\bullet\vec{y}/z] \wedge x \bullet \vec{y} \Downarrow) \qquad P \vdash \mathbf{rely}\ \ A^{\text{-}z} \wedge A_2^{z}\ \ \mathbf{guar}\ \ A_1^{\vec{y}} \wedge B^{\text{-}\vec{y}}}{\overline{x}(\vec{y}z)P \vdash \mathbf{rely}\ A\ \ \mathbf{guar}\ \ B}
\qquad
(\mathsf{Rec})\quad \dfrac{P^{\Gamma\cdot\vec{y}:\overline{\vec{\tau}};\vec{x}:\vec{\tau}^{!}} \vdash \mathbf{rely}\ \ A^{\text{-}\vec{y}}\ \ \mathbf{guar}\ \ B(0) \qquad P\ \vdash \mathbf{rely}\ \ A \wedge B(i)[\vec{y}/\vec{x}]\ \ \mathbf{guar}\ \ B(i+1)}{\mu\vec{y}{=}\vec{x}.P\ \vdash \mathbf{rely}\ \ A\ \ \mathbf{guar}\ \ B}
$$

Fig. 6. Altered/Added Proof Rules for Total Affine Logic

4.5 Total Affine Logic (2): Proof Rules

Main Rules. The proof system for the total affine logic, whose sequent we write, as in the linear process logic, $P^{\Gamma;\Delta} \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B$, with the same typing constraint, augments the proof system for the linear process logic (cf. Figure 2, §3.5, 32) in two points. First it refines (Out) rule so that interaction with the environment ensures totality. Second it treats circular composition, either by adding a rule for recursion (following Figure 5) or by adding a rule for circular parallel composition (following Figure 4). In the following we present the refined $(\mathsf{Out}^?)$ rule and the rule for recursion. The rule for circular composition as well as other variants are discussed in the latter part of the present section.

Figure 6 lists the refined $(\mathsf{Out}^?)$ rule and the rule for recursion (as before, we assume Convention 1, §3.5, for each rule). Some illustration:

- $(\mathsf{Out}^?)$ strengthens the original side condition by demanding the invocation at x surely returns an answer. We recall the notation $a^{\tau^{\uparrow}}\Downarrow$ means $\exists \vec{y}i.\ a = \mathtt{in}_i(\vec{y})^{\uparrow}$, so that $x \bullet \vec{y} \Downarrow$ indicates x returns an answer to invocation with $\vec{y}$ as arguments. In practice, this side condition is often implied by $A_2[x\bullet\vec{y}/z]$ (in fact it only suffices z occurs in A_2 since z is primary).
- (Rec) combines mathematical induction and recursive behaviour, ensuring total correctness of a typed process by a well-founded induction. In the rule, $A(e)$ denotes the result of substituting e for a hole in a formula (e.g. if $A(\,\cdot\,)$ is $u \bullet (\,\cdot\,) = (\,\cdot\,)!$, then $A(i+1)$ becomes $u \bullet (i+1) = (i+1)!$).

Let us informally trace how the recursion rule works. There are two judgements in the antecedent:

(base) P can guarantee $B(0)$ at $\vec{x}$ relying on A.
(induction) P can guarantee $B(i+1)$ at $\vec{x}$ relying on $B(i)[\vec{y}/\vec{x}]$.

From **(base)**, $(\boldsymbol{\nu}\,\vec{y})(P|R)$ can guarantee $B(0)$ relying on A, for any well-typed R. With $R \stackrel{\text{def}}{=} \Omega_{\vec{y}}(\stackrel{\text{def}}{=} \Pi_i \Omega_{y_i})$, let us set $P_0 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,\vec{y})(P|R)$. Then by **(induction)** we have $(\boldsymbol{\nu}\,\vec{y})(P|P_0[\vec{y}/\vec{x}])$, which we call P_1, can guarantee $B(1)$ relying on A.

Now we repeat the procedure: let $P_2 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,\vec{y})(P|P_1[\vec{y}/\vec{x}])$, then P_2 can guarantee $B(2)$ in addition to $B(0)$ and $B(1)$, relying on A. Similarly we construct $P_3, P_4, P_5, \ldots$, for which P_{n+1} satisfies $B(0), .., B(n+1)$. Because P_n precisely simulates $\mu\vec{y} = \vec{x}.P$ up to $n+1$-step transitions of P, we can now appeal to the classical mathematical induction to conclude P guarantees $B(i)$ for each i, reaching the conclusion of the proof rule (Rec). Since the remaining rules are sound for the same reason as in the linear process logic, we have the following result (for the proof see [53]).

Theorem 4. (soundness of affine process logic) $P \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B$ *implies* $P \models \mathbf{rely}\, A\, \mathbf{guar}\, B$ *in the total affine process logic.*

Remark 2. (completeness, Clarke's result, and strength of recursion rule)

1. We suspect the present proof system for the affine logic (regardless of the use of the variants of the rules discussed below) is not complete with respect to the given model. This is because formulae in the present logic express a larger set of properties than total correctness — for example they include a partial correctness discussed in §4.4. Thus we ask: can we obtain completeness of the proof system for an appropriately restricted class of formulae (cf. §5.3)? If we can find one, how does it relate to the properties treated in domain logics [4], which arguably correspond to total correctness properties?
2. We note a possible completeness property along the line of 1 may not contradict Clarke's result [22] because, among others, the models we consider are much narrower than what Clarke treated in [22].
3. We also note when we extend the affine logic for stateful computation, the resulting rule can justify, for a standard imperative language, the rule for the while construct which is known to be complete for strong models, as demonstrated in §10.6, Page 10.6. This suggests potential strength of the present (Rec) rule.

Variants of Proof Rules. We record the axiom for the free replicated output in the affine logic.

$$(\mathsf{FOut}^?)\ \frac{-}{\overline{x}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ A[x \bullet \vec{y}/z] \wedge x \bullet \vec{y} \Downarrow\ \mathbf{guar}\ A}$$

which says that what one can rely on the environment for invocation, including the eventual output (termination) of an answer, is what the process can guarantee for the positive primary z (again $x \bullet \vec{y} \Downarrow$ may often be implied by $A[x \bullet \vec{y}/z]$). Note z is sent to the environment and is used there, so the assumption does make sense from the viewpoint of total correctness. Note also the proof rule simply adds the termination condition to its counterpart in the linear process logic in Page 36, §3.6, which is parallel to the change in (Out$^?$).

We also record that all auxiliary structural rules in the linear process logic, given in §3.6, Page 36, are valid in the total affine logic (so are they in the other variants of the affine logics treated in the next section).

An alternative proof rule for circular behaviour is listed in Figure 7 (which is to replace (Rec) rule in Figure 6: we can also combine this rule with (Par) rule). The first antecedent tells us that, under A_1, the process P_1 guarantees $B_1(0)$. By the third antecedent, this means P_2 can guarantee $B_2(0)$ in the presence of

$$
\begin{array}{l}
(\textsf{Par-Rec}) \quad (B \equiv B_1 \wedge B_2) \\
P_1^{\overline{\Gamma_1 \cdot \vec{x}_2 : \vec{\tau}_2^{!}}; \vec{x}_1 : \vec{\tau}_1^{!}} \ \vdash \mathbf{rely}\ \ A_1^{\text{-}\vec{x}_2}\ \ \mathbf{guar}\ \ B_1(0) \\
P_1^{\overline{\Gamma_1 \cdot \vec{x}_2 : \vec{\tau}_2^{!}}; \vec{x}_1 : \vec{\tau}_1^{!}} \ \vdash \mathbf{rely}\ \ A_1^{\text{-}\vec{x}_2} \wedge B_2(i)\ \ \mathbf{guar}\ \ B_1(i+1) \\
P_2^{\overline{\Gamma_2 \cdot \vec{x}_1 : \vec{\tau}_1^{!}}; \vec{x}_2 : \vec{\tau}_2^{!}} \ \vdash \mathbf{rely}\ \ A_2^{\text{-}\vec{x}_1} \wedge B_1(i)\ \ \mathbf{guar}\ \ B_2(i) \\
\hline
P_1 | P_2 \ \vdash \mathbf{rely}\ \ A_1 \wedge A_2\ \ \mathbf{guar}\ \ B
\end{array}
$$

Fig. 7. Proof Rule for Circular Parallel Composition (total correctness)

P_1, relying on A_2, This is fed back to the initial sequent, giving us $B_1(1)$. By repeating this, we know the composition of P_1 and P_2 satisfies $B(i)$ for each i.

The rules (Par-Rec) and (Rec) are equi-potent in the following way. Below we use the closure of proof rules under injective renaming, which is an elementary, but perhaps most fundamental, property of both validity and provability of the present logic, locating the present logic as part of the general process theory [52]. We do not discuss this point further except for mentioning that it is because of this closure under injective renaming that the logic allows a smooth treatment of binding and auxiliary names.

- (Rec) is derivable from (Par-Rec) by setting P_2, B_2 and A_2 in the latter as $[\vec{x}_2 \to \vec{x}_1]$, $B_1[\vec{x}_2/\vec{x}_1]$ and T (all under the condition $\vec{\tau_1} = \vec{\tau_2}$), respectively. Via the definition of recursion via copy-cat, we obtain(Rec).

- Conversely, (Par-Rec) is derivable from (Rec) combined with (Par), as follows. Firstly, the first and third sequents in the antecedent of (Par-Rec) lead to, via (Par) and modulo injective renaming:

$$P_1[\vec{y}_2/\vec{x}_2]|P_2 \vdash \mathbf{rely}\, A_1 \wedge A_2\, \mathbf{guar}\, B(0). \tag{4.3}$$

  where we choose $\vec{y}_2$ to be fresh. Similarly the second and third sequents in the antecedent of (Par-Rec) give us, by (Conseq) and with $\vec{y}_1$ fresh:

$$P_1[\vec{y}_2/\vec{x}_2]|P_2 \vdash \mathbf{rely}\, A_1 \wedge A_2 \wedge B(i)[\vec{y_1}\vec{y_2}/\vec{x_1}\vec{x_2}]\, \mathbf{guar}\, B(i+1). \tag{4.4}$$

  By applying (Rec) to (4.3) and (4.4) we obtain $\mu\vec{y_1}\vec{y_2} = \vec{x_1}\vec{x_2}.P_1[\vec{y}_2/\vec{x}_2]|P_2$ (here $\vec{y_1}$ is vacuously used), which coincides with $P_1|P_2$ up to an appropriate equality (for example strong bisimilarity), hence reaching (Par-Rec).

In the following technical development we shall be using (Rec) since it is simpler while serving our present purposes. The rule is especially convenient when we wish to analyse the proof rules for various recursion/loop-related language constructs via encoding.

4.6 Total Affine Logic (3): Reasoning Examples

Below we give two simple inferences of reasoning in the affine process logic. More examples are found in the form of affine program logics in Sections 6, 7 and 8. The first assertion we treat is:

$$\Omega_u \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{T} \tag{4.5}$$

Since if we do anything at u, Ω_u starts diverging, this is essentially all we can say about this agent. The inference is quite simple, as follows. Below we remember $\Omega_u \stackrel{\text{def}}{=} \mu x = u.[u \to x]$.

1.	$[u \to x] \vdash \mathbf{rely}\ \ \mathsf{T}\ \ \mathbf{guar}\ \mathsf{T}$	((3.12) in §3)
2.	$[u \to x] \vdash \mathbf{rely}\ \ \mathsf{T} \wedge \mathsf{T}\ \mathbf{guar}\ \mathsf{T}$	(Conseq)
3.	$\mu x = u.[u \to x] \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{T}$	(1, 2, (Rec)).

The next example uses the factorial which appeared in (4.1) (§4.1, Example 3 (3), Page 42). We recall:

$$\mathsf{Fact}_f \quad \stackrel{\text{def}}{=} \quad \mu g = f.!f(xz).\overline{x}(c)c[\&_i P_i]$$

where:

- $P_0 \stackrel{\text{def}}{=} \overline{z}\mathtt{in}_1$ and
- $P_{i+1} \stackrel{\text{def}}{=} \overline{g}(x'z')([\![i]\!]_{x'}|z'[\&_j \overline{z}\mathtt{in}_{(i+1)\times j}])\ \ (i \geq 0)$, where, as before, we set $[\![i]\!]_u \stackrel{\text{def}}{=} !u(c).\overline{c}\mathtt{in}_i$.

Now let

$$B(f)(i) \stackrel{\text{def}}{=} \forall x.(x \bullet \varepsilon = \mathtt{in}_i()^{\uparrow} \supset f \bullet x = \mathtt{in}_{i!}()^{\uparrow}). \tag{4.6}$$

Our purpose is to derive:

$$\mathsf{Fact}_f \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \forall i.B(i).$$

The inference follows. For the base case:

1.	$\mathbf{0} \vdash \mathbf{rely}\ \ \mathtt{in}()^{\uparrow}_{0!} = \mathtt{in}()^{\uparrow}_{0!}\ \ \mathbf{guar}\ \ \mathtt{in}()^{\uparrow}_{0!} = \mathtt{in}()^{\uparrow}_{0!}$	(Zero)
2.	$\overline{z}\mathtt{in}_1 \vdash \mathbf{rely}\ \ \mathsf{T}\ \ \mathbf{guar}\ \ z = \mathtt{in}()^{\uparrow}_{0!}$	(Sel, Conseq)
3.	$P_{i+1} \vdash \mathbf{rely}\ \ \mathsf{F}\ \ \mathbf{guar}\ \ z = \mathtt{in}()^{\uparrow}_{0!}$	(Falsity)
4.	$c[\&_i P_i] \vdash \mathbf{rely}\ \ c = \mathtt{in}_0()^{\uparrow}\ \ \mathbf{guar}\ \ z = \mathtt{in}()^{\uparrow}_{0!}$	(1,2, Bra)
5.	$\overline{x}(c)c[\&_i P_i] \vdash \mathbf{rely}\ \ x \bullet \varepsilon = 0\ \ \mathbf{guar}\ \ z = \mathtt{in}()^{\uparrow}_{0!}$	(Out)
6.	$!f(xz).\overline{x}(c)c[\&_i P_i] \vdash \mathbf{rely}\ \ \mathsf{T}\ \ \mathbf{guar}\ \ B(f)(0)$	(In)

Above (Falsity) is the lemma:

$$(\mathsf{Falsity})\ \frac{-}{P \vdash \mathbf{rely}\ \mathsf{F}\ \mathbf{guar}\ A}$$

which is easily derivable by induction (assuming the well-typedness of the sequent, as always). For induction step:

$$
\begin{array}{lll}
1. & [\![i]\!]_{x'} \vdash \mathbf{rely}\ \top\ \mathbf{guar}\ x' \bullet \varepsilon = \mathtt{in}_i()^{\uparrow} & \textsf{(Zero, Sel, Rep)} \\
\hline
2. & \overline{z}\mathtt{in}_0 \vdash \mathbf{rely}\ \top\ \mathbf{guar}\ z = \mathtt{in}()^{\uparrow}_{0!} & \textsf{(Zero, Sel, Conseq)} \\
\hline
3. & \overline{z}\mathtt{in}_{(i+1)\times i!} \vdash \mathbf{rely}\ \mathtt{in}_j()^{\uparrow} = \mathtt{in}_{i!}()^{\uparrow}\ \mathbf{guar}\ z = \mathtt{in}()^{\uparrow}_{(i+1)!} & \textsf{(Zero, Sel)} \\
\hline
4. & z'[\&_j \overline{z}\mathtt{in}_{i\times j}] \vdash \mathbf{rely}\ z' = \mathtt{in}_{i!}()^{\uparrow}\ \mathbf{guar}\ z = \mathtt{in}_{(i+1)!}()^{\uparrow} & \textsf{(Bra)} \\
\hline
5. & [\![i]\!]_{x'} \mid z'[\&_j \overline{z}\mathtt{in}_{i\times j}] \ \vdash\ \mathbf{rely}\ z' = \mathtt{in}_{i!}()^{\uparrow} & \\
 & \qquad \mathbf{guar}\ x' \bullet \varepsilon = \mathtt{in}_i()^{\uparrow}\ \wedge z = \mathtt{in}_{(i+1)!}()^{\uparrow} & \textsf{(1, 4, Par)} \\
\hline
6. & [\![i]\!]_{x'} \mid z'[\&_j \overline{z}\mathtt{in}_{i\times j}] \ \vdash\ \mathbf{rely}\ B(g)(i)\ \wedge\ z' = \mathtt{in}_{i!}()^{\uparrow} & \\
 & \qquad \mathbf{guar}\ x' \bullet \varepsilon = \mathtt{in}_i()^{\uparrow}\ \wedge z = \mathtt{in}_{(i+1)!}()^{\uparrow} & \textsf{(Conseq)} \\
\hline
7. & \overline{g}(x'z')([\![i]\!]_{x'} | z'[\&_j \overline{z}\mathtt{in}_{i\times j}]) \vdash \mathbf{rely}\ B(g)(i)\ \mathbf{guar}\ z = \mathtt{in}()^{\uparrow}_{i+1!} & \textsf{(Out)} \\
\hline
8. & c[\&_i P_i] \vdash \mathbf{rely}\ B(g)(i) \wedge c = \mathtt{in}_i()^{\uparrow}\ \mathbf{guar}\ z = \mathtt{in}()^{\uparrow}_{i!} & \textsf{(Bra)} \\
\hline
9. & \overline{x}(c)c[\&_i P_i] \vdash \mathbf{rely}\ B(g)(i) \wedge x \bullet \varepsilon = i\ \mathbf{guar}\ z = \mathtt{in}()^{\uparrow}_{(i+1)!} & \textsf{(Out)} \\
\hline
10. & !f(xz).\overline{x}(c)c[\&_i P_i] \vdash \mathbf{rely}\ B(g)(i)\ \mathbf{guar}\ B(f)(i+1) & \textsf{(Out)}
\end{array}
$$

We can now combine the conclusions of the two inferences:

$$
\begin{array}{lll}
1. & !f(xz).\overline{x}(c)c[\&_i P_i] \vdash \mathbf{rely}\ \top\ \mathbf{guar}\ B(f)(0) & \textsf{(In)} \\
\hline
2. & !f(xz).\overline{x}(c)c[\&_i P_i] \vdash \mathbf{rely}\ B(g)(i)\ \mathbf{guar}\ B(f)(i+1) & \textsf{(Out)} \\
\hline
3. & \mu g = f.\,!f(xz).\overline{x}(c)c[\&_i P_i] \vdash \mathbf{rely}\ \top\ \mathbf{guar}\ \forall i.B(f)(i) & \textsf{(Rec)},
\end{array}
$$

as required.

5 Logics for Stateless Processes with Divergence (2)

5.1 Partial Affine Logic (1): Divergence and Partial Correctness

This section treats two further affine process logics (following our discussion in §4.3). The first logic is the logic for partial correctness, where assertions do not care about divergence: When a process diverges, an arbitrary (well-typed) statement becomes valid. The second logic, the general affine logic, extends the total logic so that it can specify and ensure the whole behavioural properties of affine behaviour, including possibly divergence ones, leading to the logic which subsumes both the total and partial logics.

In the partial logic, we require the process to guarantee the stated specification only when it terminates. More concretely, an assertion:

$$P^{\Gamma;\Delta} \models \mathbf{rely}\ A\ \mathbf{guar}\ B \tag{5.1}$$

now indicates that, for any R satisfying A, if $(\boldsymbol{\nu}\,\mathsf{fn}(\Gamma))(P|R)$ converges, then the result should satisfy B. The consideration of partial correctness necessitates significant change in the status of the invocation operator $x \bullet \vec{y}$, both in semantics and in proof rules. Below we illustrate the reason why we cannot maintain the original semantics of invocation, hence its proof rule, using simple examples.

Recall ω_u, which immediately diverges. From our informal understanding of partial correctness, we expect the following assertion to be valid:

$$\omega_z \models \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{F} \tag{5.2}$$

just as $\{\mathsf{T}\}$while *tt* do skip$\{\mathsf{F}\}$ is valid in Hoare Logic for partial correctness. Arguably the assertion (5.2) would be of no practical use — neither is it harmful, however, and it precisely conforms to the idea of partial correctness.

When we apply to (5.2) the proof rule for replicated input ($\mathrm{In}^{!}$) of the total logic, cf. Figure 1, the situation starts to change from useless to seriously wrong:

$$!u(z).\omega_z \models \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{F}. \tag{5.3}$$

Since $u \notin \mathsf{fn}(\mathsf{F})$, we may apply (Res) of Figure 1 — which should be completely harmless, either in total or partial logic — to derive:

$$(\boldsymbol{\nu}\,u)!u(z).\omega_z \models \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{F}. \tag{5.4}$$

But this assertion is semantically absurd, since it says that the agent which has neither any interaction point nor internal activity satisfies the strongest possible specification. In fact, for any reasonable behavioural equality $\sim$ (for example take the untyped strong bisimilarity), the following equation holds.

$$(\boldsymbol{\nu}\,u)!u(z).\omega_z \sim \mathbf{0}.$$

Thus $(\boldsymbol{\nu}\,u)!u(z).\omega_z \models \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{F}$ is semantically the same as $\mathbf{0} \models \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{F}$, but this means (by $P|\mathbf{0} \equiv P$ and $P \models \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{T}$ for any P) $P \models \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{F}$ for any P by parallel composition, making the logic semantically trivial.

The culprit for this triviality (which makes the logic useless) lies in the assertion (5.3): the given lazy divergent process is indeed completely harmless *as far as we do not invoke it*: if we ever invoke it, it does diverge, so any property — in particular F — may as well be asserted as its specification. But if we don't, it stays silent and harmless, so it is wrong to say the process satisfies an arbitrary property. The question is: what can we say about the process in (5.3), in addition to the trivial property T? Let us enunciate the computational situation:

($\star$) *The process can be invoked at* u*: if this ever happens, then the result would satisfy arbitrary properties, including* F*.*

In the next subsection we shall introduce a logical statement which can precisely state this property.

5.2 Partial Affine Logic (2): Logical Language and Semantics

The following grammar of terms takes off the invocation and the divergence constant from the total process logic. For the former this is because of the change of the meaning of invocation. For the latter having the divergence is harmless but has no significance (because of its special logical status in the satisfaction relation, as formalised later).

$$\begin{array}{lcl} e & ::= & * \mid \mathbf{t} \mid \mathbf{f} \mid n \mid x^{\mathsf{nat}} \mid x^{\mathsf{bool}} \mid e + e' \mid e \wedge e' \mid \ldots \\ a & ::= & x^{\tau} \mid \mathtt{in}_e^{\rho^{\uparrow}}(\vec{a})^{\uparrow} \ . \end{array}$$

The changed meaning of invocation is bourne by the following added syntax.

$$A \quad ::= \quad \ldots \quad \mid \quad \langle\!\langle x \bullet \vec{y}/z \rangle\!\rangle A$$

The predicate $\langle\!\langle x \bullet \vec{y}/z \rangle\!\rangle A$, which we shall call *partial substitution*, means that the result of invocation of x with $\vec{y}$ either diverges or, if it ever converges, the returned value, which we set to be z, is required to satisfy A. We can now assert for the lazy divergent behaviour in (5.3) in the following way.

$$!u(z).\omega_z \models \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \langle\!\langle u \bullet \varepsilon/z \rangle\!\rangle \mathsf{F} \tag{5.5}$$

which precisely states what we described in ($\star$). Note a partial substitution is no longer a standard first-order predicate (cf. [73, §8]) since it takes a formula as an argument: it is however a natural predicate as a typed reformulation of modal formulae in Hennessy-Milner logic. For example, a modal formula corresponding to $\langle\!\langle u \bullet \varepsilon/z \rangle\!\rangle \mathsf{F}$ may be written down as:

$$\langle u(c) \rangle \Omega \ \vee \ \langle u(c) \rangle \mathsf{F}$$

where Ω is the predicate for an infinite sequence of τ-actions: in the modal μ-calculus we can write $\Omega \stackrel{\text{def}}{=} \mu X.\langle \tau \rangle X$.

We use the same notion of abstract processes as before, which are affine processes modulo the contextual congruence up to the injective renaming (cf. §4.4, Page 44). A *model* $(\xi, \ldots)$ is, again as before, a finite map from names to abstract processes. The relation $P \models_{\mathsf{b}} \xi$ is defined just as before (cf. §3.4, Page 30).

In the satisfaction relation, we allow models to be partial: a model may as well contain the divergent process ω in its range. The satisfaction relation is defined using precisely the same clauses as for the total affine, hence linear, process logic (cf. §3.4, Page 30), adding the following clauses in the name of partiality:

- $\xi \cdot x{:}\omega \models^{\mathcal{I}} A$ for any A.
- $\xi \models^{\mathcal{I}} \langle\!\langle x \bullet \vec{y}/z \rangle\!\rangle A$ if $\xi \cdot z{:}\xi(x) \bullet \xi(\vec{y}) \models^{\mathcal{I}} A$.

The semantics of judgement is then defined by the now familiar entailment for the rely/guarantee (cf. §3.4, Definition 2, Page 31), which we reproduce below.

$$P^{\overline{\Gamma};\Delta} \models \mathbf{rely}\, A\, \mathbf{guar}\, B \quad \equiv \quad \forall R.\,(\, R^{\Gamma} \models^{\mathcal{I}} A \quad \supset \quad (\boldsymbol{\nu}\, \mathsf{fn}(\Gamma))(P|R) \models^{\mathcal{I}} B\,).$$

Remark 3. (satisfaction) By the first added defining clause for the satisfaction relation. we have $\xi \cdot x : \omega \models^{\mathcal{I}} \mathsf{F}$, which allows non-empty inhabitants in F (the best way to understand this is to consider only total inhabitants). An alternative way is to consider only total models and let the semantics of judgement mimic Hoare Logic for partial correctness, resulting in essentially the same logic.

Remark 4. (partial equality) As we noted above, the introduction of partial substitutions means we no longer need $x \bullet \vec{y}$ as a term — indeed having it is problematic since such an equation $x \bullet \vec{y} = \mathtt{in}_1()^{\uparrow}$ denotes something which is beyond (or contrary to) partial correctness. However there *is* a way to save invocation as a term by using a specific form of equality, called *partial equality*, which is derived from the partial substitution. As an example, we may set:

$$x \bullet \vec{y} \mathrel{\overset{\circ}{=}} a \quad \overset{\text{def}}{=} \quad \langle\!\langle x \bullet \vec{y}/z \rangle\!\rangle (z = a). \tag{5.6}$$

The definition tells us that the equality $x \bullet \vec{y} \mathrel{\overset{\circ}{=}} a$ means either $x \bullet \vec{y}$ diverges, or, if it ever converges, it returns an answer a (of a linear output mode), substantiating the partiality in specification in the form of equality. In general, $a \mathrel{\overset{\circ}{=}} b$ means "if both a and b converge then they should be equal". This partial equality however needs be used with care, since, for example, it does not satisfy such properties as associativity and substitutivity when invocation occurs in either side of a given equality. Another related point is treatment of inequality. The following example shows how a natural notion of inequality is defined in the present setting.

$$x \bullet \vec{y} \mathrel{\overset{?}{\neq}} \mathtt{in}_i(\vec{u}) \quad \overset{\text{def}}{=} \quad \langle\!\langle x \bullet \vec{y}/z \rangle\!\rangle (z \neq \mathtt{in}_i(\vec{u})^{\uparrow}).$$

which reads: "if $x \bullet \vec{y}$ converges then it is not the same as $\mathtt{in}_i(\vec{u})^{\uparrow}$". Note this is quite different from writing

$$\neg(x \bullet \vec{y} \mathrel{\overset{\circ}{=}} \mathtt{in}_i(\vec{u})), \tag{5.7}$$

which is never satisfied if $x \bullet \vec{y}$ diverges. In fact it demands that the process converges, stating so-called liveness property, which we shall be excluding from the partial logic. In the following technical development, we shall not include partial equality as part of formulae (though it will later re-appear in a program logic for a higher-order language).

5.3 Partial Affine Logic (3): Admissible Formulae

As Hoare has shown [45] based on the preceding work by Floyd [36], there is a tractable proof rule for recursive behaviour (or loop) when our aim is to establish partial correctness. This is because, in the first-order setting (which Hoare Logic mainly treats), the satisfaction of a formula by a behaviour P is the same thing as the one by a chain of behaviours approximating P, starting from the least defined behaviour. This property of a formula with respect to models may be called *admissibility* [32], which we now define.

Definition 5. (approximation) $P^\Gamma \sqsubseteq Q^\Gamma$ *denote* $C[P]_\Gamma^{x:()^\uparrow} \Downarrow$ *implies* $C[Q]_\Gamma^{x:()^\uparrow} \Downarrow$ *for each* $C[\ \cdot\ ]_\Gamma^{x:()^\uparrow}$. $\{P_i^\Gamma\}_{i\in\mathbb{N}}$ *such that* $P_i \sqsubseteq P_{i+1}$ approximate P_ω^Γ *iff* P_ω^Γ *is the least upper bound of* $\{P_i^\Gamma\}$ *with respect to the partial order from* $\sqsubseteq$.

Let Γ be positive ad $P \vdash \Gamma$. Then P is *least defined* if either (1) Γ contains a $\uparrow$-mode type and P is immediately diverging; or (2) Γ only contains **!**-mode types and P is of the form $\Pi_i \Omega_{x_i}$ up to $\cong$. Further let us say A is *$\mathcal{I}$-satisfiable under Γ* if we have $\xi \models^{\mathcal{I}} A$ for some $\Gamma \vdash \xi$. A is *satisfiable under Γ* if it is *$\mathcal{I}$-satisfiable* for some $\mathcal{I}$. We now set:

Definition 6. (admissibility) *Let* $\Gamma;;\Theta \vdash A$ *and* $\Theta \vdash \mathcal{I}$*. Then* A *is* admissible under $\mathcal{I}$ *iff: (1) whenever* A *is* $\mathcal{I}$*-satisfiable under* Γ *and* P^Γ *is least defined, we have* $P \models^{\mathcal{I}} A$*; and (2) whenever* $\{P_i^\Gamma\}_{i\in\mathbb{N}}$ *approximate* P *and* $P_i \models^{\mathcal{I}} A$ *for each* i*, we have* $P^\Gamma \models^{\mathcal{I}} A$*.* A *is* admissible *if it is so under each well-typed* $\mathcal{I}$*.*

Some (non) examples of admissible formulae follow.

Example 4. (admissible formulae) Below let x (of an appropriate type) be the only primary name of each formula. Processes are considered up to $\cong$.

1. F is admissible.
2. $\langle\!\langle x \bullet \varepsilon/z \rangle\!\rangle \mathsf{F}$ is admissible. This is immediate since only Ω_x satisfies this formula.
3. $\langle\!\langle x \bullet \varepsilon/z \rangle\!\rangle z = \mathtt{in}_1()^\uparrow$ is admissible. This is again easy since only Ω_x and $!x(c).\overline{c}\mathtt{in}_1$ satisfy the formula.
4. Let $A \stackrel{\text{def}}{=} \forall y.(\langle\!\langle y \bullet \varepsilon/w \rangle\!\rangle w = \mathtt{in}_i()^\uparrow \ \supset\ \langle\!\langle x \bullet y/z \rangle\!\rangle w = \mathtt{in}_{2\times i}()^\uparrow)$. Then A is admissible. This formula is satisfied by those behaviours which either diverge or return the double of a given argument (it diverges whenever the argument diverges). If $\{P_i\}$ approximates P and each P_i satisfies A, then either P is among $\{P_i\}$, in which case $P \models A$ is vacuous; or if P is not among $\{P_i\}$, then the "graph" of P is completely covered by P_i collectively. so that the description by this formula cannot contradict the behaviour of P.
5. $\neg(\langle\!\langle x \bullet \varepsilon/z \rangle\!\rangle \mathsf{F})$ is *not* admissible: note Ω_x does not satisfy it. In fact this formula indicates **total correctness** (or **liveness**) so we may as well exclude it. More interestingly, $\exists y.(\langle\!\langle x \bullet y/z \rangle\!\rangle \mathsf{F})$ (assuming the typing as (4) above) is not admissible. In this case, Ω_x does satisfy the formula: but we can take a series of partial behaviours $\{P_i\}$ which are all proper subgraphs of total P. Each P_i does satisfy this formula, but P does not.

Remark 5. (1) We can also use the intensional preorder based on well-typed traces (as defined in [13]), in which case the l.u.b. for approximation is given by the union of traces. (2) By requiring the least defined behaviour of a given action type should always satisfy an admissible formula, we can exclude F from the collection of admissible formulae. This repeats an old practice in denotational semantics. We retain F since having it is often convenient in logical specifications. (3) A logic for partial correctness should assert on safety properties and nothing else. Concretely this means we want each assertion to describe that a certain property never holds in any possible computational paths of a process; and, in this context, the notion of "less defined" in approximation means nothing but "having less paths to reach an undesirable state." This is why we should exclude non-admissible formulae in the partial logic.

For reasoning, a syntactically characterised subset of admissible formulae is useful.

Definition 7. Let $\Gamma;;\Theta \vdash A$ with $\mathsf{dom}(\Gamma) \cap \mathsf{fn}(A) = \{\vec{x}\}$. Then A^Γ is *s-admissible* if the following inductive condition holds for A (assuming the standard bound name convention).

- T, F, $e_1 = e_2$ and $a_1 = a_2$ are s-admissible.
- $\langle\!\langle u \bullet \vec{v}/z \rangle\!\rangle A$ is s-admissible if A is s-admissible.
- $A_1 \wedge A_2$ and $A_1 \vee A_2$ are s-admissible when $A_{1,2}$ are s-admissible.
- $A_1 \supset A_2$ is s-admissible when $\mathsf{fn}(A_1) \cap \{\vec{x}\} = \emptyset$ and A_2 is s-admissible.
- $\forall y.A$ is s-admissible when A is s-admissible.
- $\exists y.A$ and $\neg A$ are s-admissible when no name from $\vec{x}$ occurs in A.

Note s-admissibility of A essentially means it does not use partial substitution on primary names contravariantly. For $\exists y.A$, if we know there are only finite instances satisfying y of A, then (since it is equivalent to a finite disjunction) the s-admissibility of A suffices. We leave the proof of the following result to [53]. Below A is *I-satisfiable* if we have $\xi \models^{\mathcal{I}} A$ for some $\Gamma \vdash \xi$.

Proposition 3. *Given $\Gamma;;\Delta \vdash A$, if A is s-admissible and $\mathcal{I}$-satisfiable for some $\Theta \vdash I$, then A is admissible under $\mathcal{I}$.*

From our experience so far, the restricted set of formulae are expressive enough for describing various partial correctness properties of affine processes, even though it can be too restrictive in some situations. Leaving a further exploration of syntactic characterisations of admissibility to future occasions, we here stipulate:

Convention 2. *From now on we only consider s-admissible formulae in assertions (under given primary names).*

$$\frac{(\mathsf{In}^!)\quad (A \supset \forall \vec{y}.\,(B_1 \supset \langle\!\langle x\bullet\vec{y}/z\rangle\!\rangle B_2) \supset B) \qquad P \vdash \textbf{rely}\ A^{\text{-}\vec{y}} \wedge B_1^{\vec{y}}\ \textbf{guar}\ B_2}{!x(\vec{y}z).P \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

$$\frac{(\mathsf{Out}^?)\quad (A \supset A_1 \supset \langle\!\langle x\bullet\vec{y}/z\rangle\!\rangle A_2) \qquad R \vdash \textbf{rely}\ A\ \textbf{guar}\ A_1 \qquad P \vdash \textbf{rely}\ A \wedge A_2\ \textbf{guar}\ B}{\overline{x}(\vec{y}z)(R|P) \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

$$\frac{(\mathsf{Rec})\quad (A \supset \exists \vec{x}.B \text{ where } \mathsf{prim}(B) = \{\vec{x}\}) \qquad P \vdash \textbf{rely}\ A^{\text{-}\vec{y}} \wedge B[\vec{y}/\vec{x}]\ \textbf{guar}\ B}{\mu\vec{y}{=}\vec{x}.P \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

Fig. 8. Altered Proof Rule (partial correctness)

5.4 Partial Affine Logic (4): Proof Rules

The proof rules for the partial correctness are precisely the same as those for the total logic (which are in turn identical with those for the linear process logic but for the rule of recursion) except for the following two points:

1. For the replicated input and its dual, we use the partial substitution rather than the syntactic substitution.
2. The rule for recursion no longer uses mathematical induction, but is directly based on circularity of names and properties.

The updated proof rules vis-a-vis the total logic are summarised in Figure 8, which we illustrate in the following one by one (we are working under Convention 1, §3.5, as always). The original (In$^!$) rule in Figure 1 now becomes:

$$(\mathsf{In}^!)\ \frac{P \vdash \textbf{rely}\ A^{\text{-}\vec{y}} \wedge B_1^{\vec{y}}\ \textbf{guar}\ B_2\ ,\quad A \supset \forall \vec{y}.(B_1 \supset \langle\!\langle x\bullet\vec{y}/z\rangle\!\rangle B_2) \supset B}{!x(\vec{y}z).P \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

Note $\forall \vec{y}.(B_1 \supset \langle\!\langle x\bullet\vec{y}/z\rangle\!\rangle B_2)$ is admissible if B_2 is since the primary names in B_1, which are $\vec{y}$, are now quantified. Note also $\forall \vec{y}.(B_1 \supset \langle\!\langle x\bullet\vec{y}/z\rangle\!\rangle B_2)$ is logically equivalent to $\forall \vec{y}.\langle\!\langle x\bullet\vec{y}/z\rangle\!\rangle(B_1 \supset B_2)$. A is mentioned in the side condition solely for the case $A \equiv \mathsf{F}$. Similarly (Out$^!$) rule becomes:

$$(\mathsf{Out}^?)\ \frac{S \vdash \textbf{rely}\ A\ \textbf{guar}\ A_1,\quad P \vdash \textbf{rely}\ A^{\text{-}z} \wedge A_2\ \textbf{guar}\ B,\quad A \supset A_1 \supset \langle\!\langle x\bullet\vec{y}/z\rangle\!\rangle A_2}{\overline{x}(\vec{y}z)(S|P) \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

Finally we replace the recursion rule with the following one, which no longer uses mathematical induction. Below $\mathsf{prim}(B)$ denotes the set of primary free names in B.

$$\frac{(\mathsf{Rec})\quad (A \supset \exists \mathsf{prim}(B).B) \qquad P \vdash \textbf{rely}\ A^{\text{-}y} \wedge B[y/x]\ \textbf{guar}\ B}{\mu y{=}x.P \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

While we no longer need two assertions in the premise as in the original (Rec), we need an additional side condition. The condition $A \supset \exists \mathsf{prim}(B).B$ says that the constraints which A has on its auxiliary names should be stronger than that of B. Without this, we can easily derive inconsistent assertions. For example, from the following sound statement:

$$[x \to y] \vdash \ \mathbf{rely}\ \mathsf{F} \wedge \mathsf{F}\ \mathbf{guar}\ \mathsf{F} \tag{5.8}$$

we can derive, if (Rec) lacks the side condition above, the following statement.

$$\mu y = x.[x \to y] \vdash \ \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{F} \tag{5.9}$$

Noting the recursive process above is nothing but Ω_x, this leads us to the trivial logic from a semantic viewpoint. As can be seen from this example, the soundness of the (Rec) rule above crucially relies on its side condition. While the side condition is semantic, the use of admissibility means it usually suffices to assign the least defined behaviours to $\mathsf{prim}(B)$ for checking the condition, as we shall indeed see in the next subsection.

5.5 Partial Affine Logic (5): Examples of Inference

Copycats. We present simple examples of inferences for the affine logic. First we infer the property of the copy-cat, as we have done before for the total logic. The properties are identical with before except for replicated output.

$$[x \to x']^{\tau^!} \vdash \mathbf{rely}\ A[x'/x]\,\mathbf{guar}\ A \tag{5.10}$$

$$[x \to y]^{\tau^\downarrow} \vdash \mathbf{rely}\ A[x/y]\ \mathbf{guar}\ A \tag{5.11}$$

$$\overline{x}\langle \vec{y}z\rangle^{\vec{\rho}\tau} \ \vdash \mathbf{rely}\ \langle\!\langle x \bullet \vec{y}/z\rangle\!\rangle A\ \mathbf{guar}\ A \tag{5.12}$$

$$\overline{x}\mathtt{in}_i\langle\vec{y}\rangle^{\vec{\rho}} \vdash \mathbf{rely}\ \langle\!\langle \mathtt{in}_i(\vec{y})/x\rangle\!\rangle A\ \mathbf{guar}\ A \tag{5.13}$$

We only show the inference for (5.10) and (5.12). Others are inferred precisely as in Section 3, including the n-ary version of (5.10). First we infer (5.13).

$$\begin{array}{lll}
1. & [\vec{y}' \to \vec{y}] \vdash \mathbf{rely}\ \langle\!\langle x \bullet \vec{y}/z\rangle\!\rangle A\,\mathbf{guar}\ \langle\!\langle x \bullet \vec{y}'/z\rangle\!\rangle A & (\mathsf{from\ (5.10)}) \\ \hline
2. & [z' \to z] \vdash \mathbf{rely}\ \langle\!\langle x \bullet \vec{y}/z\rangle\!\rangle A \wedge A[z'/z]\,\mathbf{guar}\ A & (\mathsf{from\ (5.11),\ (Conseq)}) \\ \hline
3. & \overline{x}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ \langle\!\langle x \bullet \vec{y}/z\rangle\!\rangle A\,\mathbf{guar}\ A & (\mathsf{1,\ 2,\ (Out)})
\end{array}$$

For (5.10), let $A(x) \stackrel{\text{def}}{=} \forall \vec{y}\vec{i}.(\vec{y} = \vec{i} \supset \langle\!\langle \vec{x} \bullet \vec{y}/z\rangle\!\rangle E)$ such that $\{x\vec{y}\} \cap \mathsf{fn}(E) = \emptyset$.

$$\begin{array}{lll}
1. & \overline{x'}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ \langle\!\langle x \bullet \vec{y}/z\rangle\!\rangle E\,\mathbf{guar}\ E & (\mathsf{5.12}) \\ \hline
2. & \overline{x'}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ A(x') \wedge \vec{y} = \vec{i}\,\mathbf{guar}\ E & (\mathsf{Conseq}) \\ \hline
3. & !x(\vec{y}z).\overline{x}\langle\vec{y}z\rangle \vdash \mathbf{rely}\ A(x')\,\mathbf{guar}\ \forall\vec{y}\vec{i}.(\vec{y} = \vec{i} \supset \langle\!\langle x \bullet \vec{y}/z\rangle\!\rangle E) & (\mathsf{In}\text{-}\forall).
\end{array}$$

The form of the sequent does not lose generality by the following lemma. The proof uses s-admissibility, which we omit from the present version.

Lemma 3. *Let $x : (\vec{\tau}\rho)^!;;\Theta \vdash A$, A is s-admissible and is satisfiable. Then $A \equiv \forall\vec{y}\vec{i}.(\vec{y} = \vec{i} \supset \langle\!\langle x \bullet \vec{i}/z\rangle\!\rangle A')$ for some A' in which no name in $x\vec{y}$ occurs.*

Diverging Agents. Noting $\Omega_x \stackrel{\text{def}}{=} \mu x' = x.[x \to x']$ we infer:

$$\frac{1.\quad [x \to x'] \vdash \mathbf{rely}\ \mathsf{T} \wedge \forall \vec{y}.\langle\!\langle x' \bullet \vec{y}/z \rangle\!\rangle \mathsf{F}\ \mathbf{guar}\ \forall \vec{y}.\langle\!\langle x \bullet \vec{y}/z \rangle\!\rangle \mathsf{F} \quad \text{(from (5.10))}}{2.\quad \Omega_x \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \forall \vec{y}.\langle\!\langle x \bullet \vec{y}/z \rangle\!\rangle \mathsf{F} \quad \text{(Rec)}}$$

Note the side condition for the (Rec) rule, $\mathsf{T} \supset \exists x.\langle\!\langle x \bullet \vec{y}/z \rangle\!\rangle \mathsf{F}$, is immediate by taking the least defined behaviour for x. It is notable that we *cannot* replace $\forall \vec{y}.\langle\!\langle x' \bullet \vec{y}/z \rangle\!\rangle \mathsf{F}$ by F precisely because this side condition no longer holds.

From the lazy divergent agent Ω_x, we can construct an immediately diverging agent ω_x, for which we infer as follows. Let $\vdash P \triangleright y : \tau^{\uparrow}$ below. Below note $P \vdash \mathbf{rely}\ \mathsf{F}\ \mathbf{guar}\ \mathsf{F}$, which we call (falsity), is also inferable for any well-typed P in the partial logic.

$$\begin{array}{lll}
1. & \Omega_x^{((()^{\uparrow})^!} \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \forall \vec{y}.\langle\!\langle x \bullet \vec{y}/z \rangle\!\rangle \mathsf{F} & \text{(2 above)} \\ \hline
2. & c.P \vdash \mathbf{rely}\ \mathsf{F}\ \mathbf{guar}\ \mathsf{F} & \text{(falsity)} \\ \hline
3. & \overline{x}(c)c.P \vdash \mathbf{rely}\ \mathsf{T} \wedge \forall \vec{y}.\langle\!\langle x \bullet \vec{y}/z \rangle\!\rangle \mathsf{F}\ \mathbf{guar}\ \mathsf{F} & \text{(Out)} \\ \hline
4. & \Omega_x | \overline{x}(c)c.P \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{F} & \text{(Par)} \\ \hline
5. & (\boldsymbol{\nu}\, x)(\Omega_x | \overline{x}(c)c.P) \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{F} & \text{(Res).}
\end{array}$$

Factorial. Finally we treat the recursive factorial. We write down the property formally:

$$B'(f) \stackrel{\text{def}}{=} \forall x, n.(x \bullet \varepsilon = \mathtt{in}_n()^{\uparrow} \supset \langle\!\langle f \bullet x/z \rangle\!\rangle (z = \mathtt{in}_{n!}()^{\uparrow})) \tag{5.14}$$

which can also be written as $\forall x, n.(x \bullet \varepsilon \mathrel{\mathring{=}} \mathtt{in}_n()^{\uparrow} \supset f \bullet x \mathrel{\mathring{=}} \mathtt{in}_{n!}()^{\uparrow})$ using the notation for partial equality (cf. Remark 4, Page 53). The inference follows.

$$\begin{array}{lll}
1. & \overline{z}\mathtt{in}_0 \vdash \mathbf{rely}\ \mathtt{in}_0()^{\uparrow} = \mathtt{in}_0()^{\uparrow}\ \mathbf{guar}\ z = \mathtt{in}()^{\uparrow}_{0!} & \text{(Zero, Sel)} \\ \hline
2. & \overline{z}\mathtt{in}_{(i+1)\times j} \vdash \mathbf{rely}\ j = i!\ \mathbf{guar}\ z = \mathtt{in}()^{\uparrow}_{(i+1)!} & \text{(Zero, Sel)} \\ \hline
3. & z'[\&_j \overline{z}\mathtt{in}_{i\times j}] \vdash \mathbf{rely}\ z' = \mathtt{in}_{i!}()^{\uparrow}\ \mathbf{guar}\ z = \mathtt{in}_{(i+1)!}()^{\uparrow} & \text{(2, Bra)} \\ \hline
4. & [\![i]\!]_{x'} \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ x' \bullet \varepsilon = \mathtt{in}_i()^{\uparrow} & \text{(Zero, Sel, Rep)} \\ \hline
5. & \overline{g}(x'z')([\![i]\!]_{x'} | z'[\&_j \overline{z}\mathtt{in}_{i\times j}]) \vdash \mathbf{rely}\ B'(g)\ \mathbf{guar}\ z = \mathtt{in}()^{\uparrow}_{(i+1)!} & \text{(3, 4, Out)} \\ \hline
6. & c[\&_i P_i] \vdash \mathbf{rely}\ B'(g) \wedge c = \mathtt{in}_i()^{\uparrow}\ \mathbf{guar}\ z = \mathtt{in}()^{\uparrow}_{i!} & \text{(1, 5, Bra)} \\ \hline
7. & \overline{x}(c)c[\&_i P_i] \vdash \mathbf{rely}\ B'(g) \wedge x \bullet \varepsilon = i\ \mathbf{guar}\ z = \mathtt{in}()^{\uparrow}_{(i+1)!} & \text{(6, Out)} \\ \hline
8. & !f(xz).\overline{x}(c)c[\&_i P_i] \vdash \mathbf{rely}\ B'(g)\ \mathbf{guar}\ B'(f) & \text{(7, In)} \\ \hline
9. & \mu g = f.!f(xz).\overline{x}(c)c[\&_i P_i] \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ B'(f) & \text{(8, Rec),}
\end{array}$$

as required.

5.6 General Affine logic (1): Syntax and Semantics of Formulae

In the general affine logic, an assertion can now talk about not only terminating behaviour but also diverging ones: if an assertion says a given process terminates (resp. diverges), the process terminates (resp. diverges). We use the same logical language as that of the total logic. The semantics of formulae and assertions is given precisely as in the total logic except for a small, but crucial, change: we now allow models to be partial, i.e. they are now well-typed finite maps from names to any abstract processes. Thus even the formula $x \Downarrow$ (cf. Section 4.5) with $x^{\tau^{\uparrow}}$ being a primary name, which is equivalent to T in the total logic, has a non-trivial meaning. So does $x \Uparrow$, which is, when x is primary, defined to be $\neg(x \Downarrow)$, hence has always been F in the total logic.

Let us see how this logic specifies both totality and partiality, and how it compares with the two affine logics we have seen so far. We start from a rather trivial assertion which would, however, makes clear the distinction between the total logic and the general logic. Below for clarity we annotate the turnstile of a sequent with the name of the logic.

$$P^{\overline{\Gamma};\Delta} \models^{\mathcal{I}}_{\mathtt{total}} \textbf{rely}\,\mathsf{T}\,\textbf{guar}\,\mathsf{T} \tag{5.15}$$

$$P^{\overline{\Gamma};\Delta} \models^{\mathcal{I}}_{\mathtt{general}} \textbf{rely}\,\mathsf{T}\,\textbf{guar}\,\mathsf{T} \tag{5.16}$$

Having the same textual presentation, (5.15) and (5.16) have quite different meanings if Γ and Δ contain linear types. (5.15) means P would terminate whenever the environment terminates. In contrast, (5.16) means P would have any well-typed affine behaviour, including divergence, relying on nothing about the environment except well-typedness. Note if Γ and Δ are both replicated, then (since replicated processes are by definition total), no difference comes about. Another typical distinguishing examples follow.

$$y[\&_{i\in\mathbb{N}}\overline{x}\mathtt{in}_1] \models^{\mathcal{I}}_{\mathtt{total}} \textbf{rely}\,\mathsf{T}\,\textbf{guar}\,x = \mathtt{in}_1()^{\uparrow}. \tag{5.17}$$

$$y[\&_{i\in\mathbb{N}}\overline{x}\mathtt{in}_1] \models^{\mathcal{I}}_{\mathtt{general}} \textbf{rely}\,\mathsf{T}\,\textbf{guar}\,x = \mathtt{in}_1()^{\uparrow}. \tag{5.18}$$

(5.17) is valid but (5.18) is not, since, as noted above, T in the general logic stands for all (well-typed) behaviours including divergence, which, in this case, means the input y may not be fired, so the output x may not be fired either. We can however transform (5.18) to an assertion equivalent to (5.17) as follows.

$$y[\&_{i\in\mathbb{N}}\overline{x}\mathtt{in}_1] \models^{\mathcal{I}}_{\mathtt{general}} \textbf{rely}\,y\Downarrow\,\textbf{guar}\,x = \mathtt{in}_1()^{\uparrow} \tag{5.19}$$

which is indeed valid in the generalised logic. It is easy to see each valid assertion in the total affine logic can be translated into a valid assertion in the general affine logic by augmenting formulae on linear actions with the condition for convergence appropriately.

The same is true with the partial logic. A simple assertion in the partial logic:

$$P \models^{\mathcal{I}}_{\mathtt{partial}} \textbf{rely}\,\mathsf{T}\,\textbf{guar}\,\langle\!\langle u \bullet \varepsilon/z\rangle\!\rangle \exists i^{\mathbb{N}}.\ z = \mathtt{in}_{2*i}()^{\uparrow} \tag{5.20}$$

$$\frac{(\mathsf{Bra}^{\downarrow}\Downarrow)\quad (B\Downarrow_z \supset A\Downarrow_x) \qquad \forall i.\ P_i^{\Gamma_i, z:\tau^{\uparrow}} \vdash \mathbf{rely}\ A[\mathtt{in}_i(\vec{y}_i)^{\uparrow}/x]\,\mathbf{guar}\ B}{x[\&_i(\vec{y}_i).P_i] \vdash \mathbf{rely}\ A\,\mathbf{guar}\ B}$$

$$\frac{(\mathsf{Out}^{?})\quad (A \supset A_1 \supset A_2[x \bullet \vec{y}/z]) \qquad P \vdash \mathbf{rely}\ A^{\text{-}z} \wedge A_2^{z}\ \mathbf{guar}\ A_1^{\vec{y}} \wedge B^{\text{-}\vec{y}}}{\overline{x}(\vec{y}z)P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}$$

$$\frac{(\mathsf{Rec}\text{-}\Uparrow)\quad (B \text{ admissible, } A \supset \exists \mathsf{prim}(B).B) \qquad P\ \vdash \mathbf{rely}\ A^{\text{-}y} \wedge B[y/x]\ \mathbf{guar}\ B}{\mu y = x.P\ \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}$$

Fig. 9. Updated and Added Proof Rules for General Affine Logic

says that, when invoked at u, *either* P diverges, *or* it would return an even number. This assertion has no counterpart in the total logic, but does have one in the general affine logic, simply by making the disjunction explicit:

$$P \models^{\mathcal{I}}_{\mathtt{general}} \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ u \bullet \varepsilon \Uparrow \vee \exists i^{\mathbb{N}}.\ u \bullet \varepsilon = \mathtt{in}_{2\times i}()^{\uparrow} \qquad (5.21)$$

In this way we can turn any assertion in the partial logic into the one for the general affine logic, by adding a disjunction with possible divergence at each appropriate place. The general affine logic thus subsumes both the total logic and the partial logic, though it does need verbose assertions for expressing the same kinds of assertions in each of them, in order to make explicit what is (arguably conveniently!) left implicit in each simpler logic.

5.7 General Affine logic (2): Proof Rules

The proof system for the general affine logic uses the identical rules as the total logic except for the following two changes and one addition (listed in Figure 9).

- $(\mathsf{Bra}^{\downarrow})$ has to be refined so that it demands the environment to terminate when the specification requires the process itself also to terminate. In the rule, "$A\Downarrow_x$" ("*A ensures convergence at x*") stands for "$A \supset x\Downarrow$."
- In turn, $(\mathsf{Out}^{?})$ no longer needs the additional condition on termination, since the process does not have to be always terminating any more. Indeed, suppose, in the rule, B says P terminates, i.e. P outputs at some free linear output channel, say w. In that case, the linear part of P (which receives at z and outputs at w) satisfies B under A_2 (and A), so that, as can be seen from the side condition of $(\mathsf{Bra}^{\downarrow})$, A_2 surely says there is an input at z (if not it cannot guarantee B: note that this was *not* necessary in the linear input rule for the total logic since termination is the default behaviour even for the environment). Now the side condition $A \supset A_1 \supset A_2[x \bullet \vec{y}/z]$ means the conditions on x and $\vec{y}$ in A_1 and A_2 are sufficiently strong to ensure the termination of $x \bullet \vec{y}$, so the rule needs no further condition on termination.

– $(\mathsf{Rec}\text{-}\Uparrow)$ additionally treats the case when the formula is admissible, where the admissibility is defined precisely as in Definition 6, Section 5.3. For the same reason as in the partial logic, this allows us to do a circular reasoning.

For $(\mathsf{Bra}^{\downarrow})$, we can easily derive an unsound assertion without the added side condition. For example:

$$\dfrac{(\mathsf{Sel})\ \dfrac{-}{\overline{z} \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ z \Downarrow}}{(\mathsf{Bra})\ \ x.\overline{z} \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ z \Downarrow}$$

whose conclusion is obviously invalid (cf. §5.17). For $(\mathsf{Out}^{?})$, note the shape of the rule thus returns to the original rule for linear processes, even though the underlying semantics is quite different. For $(\mathsf{Rec}\text{-}\Uparrow)$, the rule indicates the extra generality which the incorporation of divergence gives to the logic, even though admissibility should be semantically given (syntactic characterisation in the general logic may become awkward). An interesting question is whether there are reasonable proof rules for recursion other than (Rec) and $(\mathsf{Rec}\text{-}\Uparrow)$.

5.8 Use of General Affine Logic

As we already noted, all provable assertions in the two affine logics are provable in the general affine logic by augmenting formulae. What does this logic *additionally* give us in analysing behaviours? In what situations can we use the present logic, not the other two ones, especially the total one? In the total logic, we can prove, for arbitrary well-typed P and A:

$$P^{c:\tau^{\uparrow}} \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ A \ \supset\ \overline{x}(b)b[.P\ \&\ .P] \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ A \qquad (5.22)$$

This essentially means there is no information flow from x to the observable actions of the process, since anything provable about P is provable for this process. It is clear that this entailment relies on the assumption that, when it invokes at x, the environment surely returns.

In the general affine logic, (5.22) does not hold. Instead, however, we can prove the following (derivable for each P and A by the shape of the proof rules):

$$P^{c:\tau^{\uparrow}} \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ A \ \supset\ \overline{x}(b)b[.P\ \&\ .P] \vdash \mathbf{rely}\ x \Downarrow\ \mathbf{guar}\ A \qquad (5.23)$$

where $x \Downarrow$ stands for $x \bullet \varepsilon \Downarrow$. The refined statement says that, as far as the environment is assumed to return a value via x, there is no dependency of P on the environment's behaviour.

The conditional notion of independence in behaviour, as exemplified in (5.23), is precisely what we need for justifying some of the well-known type-based information flow analyses [109]. Further standard information flow analyses *can* analyse the case when P is diverging, as expressed in (5.23). In fact, many compositional program analyses are able to soundly analyse programs without having to assume or prove the termination of whole programs, and are rightly so: if the soundness of an analysis relies on termination, then either the set of analysable programs is very restricted or, alternatively, the analysis is likely to be infeasible. Because the general affine logic can compositionally reason about arbitrary (observable) properties of process behaviour, it offers a powerful tool for developing. and reasoning about, various compositional program analyses, especially in its incarnation as derived program logics (discussed in Section 6).

In summary, when we specify/analyse programs' behaviour, it is sometimes essential that we take divergent behaviours into account, including that of the environment. Thus, for giving a logical analysis in such situations, we need a logic which naturally treats divergent behaviour as part of its universe of discourse. The general affine logic precisely gives such a framework. From a theoretical viewpoint, the properties representable in the general affine logic would cover most of the (finitely describable) interesting behavioural properties of affine processes. This leads to a natural question: for each process, can we find a formula (judgement) which precisely characterises its behaviour up to the contextual equality? If there is, can we always derive that judgement using our proof rules?

6 Program Logics via Process Logics (1): Total Correctness for PCF_v

6.1 Overview of Sections 6, 7 and 8

The purpose of this section and the next two is to illustrate how the affine logic can be used for guiding the development and analyses of the logics for programs, taking basic functional languages as examples. Concretely we discuss:

1. A program logic for total correctness of PCF_v, the call-by-value version of PCF [32], with several basic extensions such as letrec, sums and products (this section).
2. Logics for partial and generalised total correctness for PCF_v as well as program logics for the total and partial correctness of call-by-name PCF (Section 7).
3. Logics for advanced type structures, including two forms of polymorphism and recursive types, preceded with the corresponding extensions in the process logic (Section 8).

In each case, the soundness of the derived program logic is established through the soundness of the underlying process logic via embedding. In Sections 10 and 12, we extend these process logics to further incorporate stateful computation. One of the emphasis throughout these discussions is the clean logical treatment of various data types in the derived logics, which comes from uniform treatment of various algorithmic entities via their decomposition into name passing processes in the process logic.

A note is due concerning connection between the program logics thus derived and the corresponding "original" process logics. A program logic articulates the semantics of a programming language, so that, ideally, it may as well reflect nothing but the intrinsic nature of dynamics and algebra of its target language, rather than something else. If the typed process logics are to play the rôle of a truly language-neutral medium, as we envisaged in Introduction, then they should be as transparent as possible as a tool for analysis, so that it can guide the development of a program logic which reflects the inherent nature of the language, without imposing any unnecessary restriction on the shape of logical articulation of a given programming language. While it is still too early to conclude this is in general possible for a wide variety of programming languages, we hope that the instances we treat in the following discussions. Among others this section and the next demonstrate how two distinct paradigms of functional computation, call-by-name and call-by-value, are reflected on their respective program logics.

6.2 A Summary of PCF_v and its Process Encoding

Below we summarise the syntax of PCF_v, on which our subsequent development mainly focus (for its call-by-name counterpart, see §7.1).

$$M \ ::= \ \mathbf{c} \ \mid \ x \ \mid \ \lambda x.M \ \mid \ MN \ \mid \ \mu x.\lambda y.M \ \mid \ \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 \ \mid \ \mathbf{op}(\vec{M})$$

$$[Var]\ \frac{-}{\Gamma, x\!:\!\alpha \vdash x : \alpha} \quad [Num]\ \frac{-}{\Gamma \vdash \mathbf{n} : \mathbb{N}} \quad [Succ]\ \frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \mathtt{succ}(M) : \mathbb{N}} \quad [Eq]\ \frac{\Gamma \vdash M_{1,2} : \mathbb{N}}{\Gamma \vdash M_1 = M_2 : \mathbb{B}}$$

$$[Abs]\ \frac{\Gamma, x\!:\!\alpha \vdash M : \beta}{\Gamma, x\!:\!\alpha \vdash \lambda x^{\alpha}.M : \alpha \Rightarrow \beta} \qquad [App]\ \frac{\Gamma \vdash M : \alpha \Rightarrow \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta}$$

$$[Rec]\ \frac{\Gamma, x : \alpha \Rightarrow \beta \vdash \lambda y^{\alpha}.M : \alpha \Rightarrow \beta}{\Gamma \vdash \mu x^{\alpha \Rightarrow \beta}.\lambda y^{\alpha}.M : \alpha \Rightarrow \beta} \qquad [If]\ \frac{\Gamma \vdash M : \mathbb{B} \quad \Gamma \vdash N_{1,2} : \alpha}{\Gamma \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : \alpha}$$

Fig. 10. Typing Rules for PCF_v

Binding etc. are standard. Bound variables are later annotated by types. $\mathbf{c}$ indicates a constant, while $\mathbf{op}(M_0, ..., M_{n-1})$ denotes an n-ary arithmetic and boolean operation. Among others we consider:

- Non-negative numerals, written $\mathbf{n}$ etc., and booleans, written $\mathbf{b}$ etc.
- the unary operations $\mathtt{succ}(M)$ (the successor) and $\neg M$ (the negation) and binary operations $M+N$, $M-N$, $M*N$, $M = N$ (equality of two numbers), $M \wedge N$ and $M \vee N$.

These are intended neither to be minimal in terms of expressiveness nor to be convenient for writing algorithms, but should be enough for illustrating the treatment of various operations in our logical language. *Values* $(V, V', \ldots)$ are variables, constants and λ-abstractions. We use the standard weak call-by-value one-step reduction [32, 115], written $M \longrightarrow M'$.

Types, ranged over by $\alpha, \beta, \ldots$, are generated from:

$$\alpha \quad ::= \quad \mathbb{B} \quad | \quad \mathbb{N} \quad | \quad \alpha \Rightarrow \beta.$$

$\mathbb{B}$ and $\mathbb{N}$ are called *atomic types* (or sometimes *base types*). The typing judgements have the form $\Gamma \vdash M : \alpha$, where Γ is a *base*, which is a finite map from variables to types. Well-typed terms are derived from the rules in Figure 10 (only part of constants and operators are treated, which are easily extended to other cases). When $\Gamma \vdash M_{1,2} : \alpha$, we write $\Gamma \vdash M_1 \cong M_2 : \alpha$ if, for each closing $C[M_i]$ of type $\mathbb{N}$, we have $C[M_1] \Downarrow$ iff $C[M_2] \Downarrow$.

PCF_v can be fully abstractly embedded into the affine π-calculus [13] using a simple encoding by Milner [58, 80]. Let us present the encoding of types:

$$\mathbb{N}^{\circ} \stackrel{\text{def}}{=} ([\oplus_{i \in \mathbb{N}}]^{\uparrow_{\mathrm{L}}})^{!} \quad \mathbb{B}^{\circ} \stackrel{\text{def}}{=} ([\oplus_{i \in \mathbb{B}}]^{\uparrow_{\mathrm{L}}})^{!} \quad (\alpha \Rightarrow \beta)^{\circ} \stackrel{\text{def}}{=} (\overline{\alpha^{\circ}}(\beta^{\circ})^{\uparrow})^{!}$$

In the mapping above, the symbol $\uparrow_{\mathrm{L}}$ indicates that the output is linear rather than affine (which formally means it cannot be suppressed by an affine input and its enclosing replication cannot be recursed. This partial mixing of linearity allows us to have a more uniform encoding than in [13] while keeping the key semantic properties (even though the encoding in [13] is terser). The main part of the process encoding is given below (the remaining parts should easily follow), which closely follows that of types. The encoding in fact maps a fully typed term $\Gamma \vdash M : \alpha$ into a typed process, even though we omit type information in the encoding except when they are absolutely necessary. Below $\mathbf{n}'$ is the successor of $\mathbf{n}$ and u is chosen to be fresh.

$$
\begin{aligned}
\langle\!\langle \mathbf{n} \rangle\!\rangle^{\mathrm{val}}_m &\stackrel{\text{def}}{=} !m(c).\overline{c}\mathtt{in}_n \\
\langle\!\langle x^\alpha \rangle\!\rangle^{\mathrm{val}}_m &\stackrel{\text{def}}{=} [m \to x]^{\alpha^\circ} \\
\langle\!\langle \lambda x^\alpha.M \rangle\!\rangle^{\mathrm{val}}_m &\stackrel{\text{def}}{=} !m(xc).\langle\!\langle M \rangle\!\rangle_c \\
\langle\!\langle \mu x^{\alpha \Rightarrow \beta}.V \rangle\!\rangle^{\mathrm{val}}_m &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\, x)(\langle\!\langle V \rangle\!\rangle^{\mathrm{val}}_m | [x \to m]^{(\alpha \Rightarrow \beta)^\circ}) \\
\langle\!\langle MN \rangle\!\rangle_u &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle\!\langle M \rangle\!\rangle_m | m(c).(\boldsymbol{\nu}\, n)(\langle\!\langle N \rangle\!\rangle_n | n(e).\overline{c}\langle eu \rangle)) \\
\langle\!\langle \mathtt{succ}(M) \rangle\!\rangle_u &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle\!\langle M \rangle\!\rangle_m | m(c).\overline{c}(e)e[\&_n \langle\!\langle \mathbf{n}' \rangle\!\rangle_u]) \\
\langle\!\langle \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 \rangle\!\rangle_u &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle\!\langle M \rangle\!\rangle_m | m(c).\overline{c}(e)e[\&_{i \in \{1,2\}} \langle\!\langle N_i \rangle\!\rangle_u]) \\
\langle\!\langle V \rangle\!\rangle_u &\stackrel{\text{def}}{=} \overline{u}(m)\langle\!\langle V \rangle\!\rangle^{\mathrm{val}}_m.
\end{aligned}
$$

We can check that the encoding is type-preserving in the following way. Below let u be fresh and $\overline{\Gamma^\circ}$ denotes the result of pointwise mapping each channel type by $(\cdot)^\circ$ and then dualising it.

$$\Gamma \vdash M : \alpha\,, \quad u \quad \mapsto \quad \vdash \langle\!\langle M \rangle\!\rangle_u \triangleright \overline{\Gamma^\circ},\, u\!:\!(\alpha^\circ)^\uparrow \tag{6.1}$$

The following properties are essentially what are established in [13]. (2) and (3) say the universe of processes of translated types is isomorphic to that of PCF_v in a rather strong sense (the clause 2 is stronger than [13] in that not only finite processes are definable but also all typable processes are, which is because of the computablity assumption for the syntax of processes, cf. Appendix A.2). Below $\cong_\pi$ is the maximal contextual congruence for the affine π-calculus taking observables at $()^\uparrow$ (that is: $\cong_\pi$ is the maximum congruence which satisfies, for each $P_{1,2}$ of type $x : ()^\uparrow$, $P_1 \cong_\pi P_2$ if and only if we have $P_1 \Downarrow \equiv P_2 \Downarrow$).

Proposition 4. *1.* (computational adequacy) *Let $M : \alpha$ be closed. Then $\langle\!\langle M \rangle\!\rangle \Downarrow$ iff $M \Downarrow V$ for some V.*

2. (definability) *For each $\vdash P \triangleright u : \alpha^\bullet$, there exists $\vdash M : \alpha$ such that $\langle\!\langle M : \alpha \rangle\!\rangle_u \cong_\lambda P$.*

3. (full abstraction) *For each $\Gamma \vdash M_{1,2} : \alpha$, $\Gamma \vdash M_1 \cong_\lambda M_2 : \alpha$ iff $\langle\!\langle M_1 : \alpha \rangle\!\rangle_u \cong_\pi \langle\!\langle M_2 : \alpha \rangle\!\rangle_u$.*

6.3 Assertions for PCF_v (1): Syntax

The significance of type preservation in the mapping (6.1) in the present context is that the translation guides us to develop a simple program logic for PCF_v which is justifiable via the process logic. As we stressed in §6.1, the resulting logical language is not an automatic translation of that of the process logic:

Yet the type-preserving embedding offers enough information using which we can arrive at a logical language of fairly good shape. The encoding also leads to simple proof rules for the logic, whose soundness is automatically ensured via the embeddability of the PCF_v-logic into the affine process logic. In the following we illustrate how this is possible in the context of the logic for total correctness of PCF_v-programs, using the affine process logic discussed in Section 4 as a basic analytical tool. In the next section we shall discuss how the same framework extends to partial and general affine logics.

We start from an assertion in the process logic for an encoding of $\Gamma \vdash M : \alpha$. By (6.1), we know it has the following form:

$$\langle\!\langle M \rangle\!\rangle_u \vdash \mathbf{rely}\ A^{\overline{\Gamma^\circ}}\ \mathbf{guar}\ E^{\,u:(\alpha^\circ)^\uparrow} \tag{6.2}$$

For clarity we annotate A and E with respective primary names. A key observation which leads to a simple assertion language for PCF_v is that, since the logic is total, as far as A is satisfiable (and if not, i.e. if $A \equiv \mathsf{F}$, the assertion is vacuously valid for any E), we can set E as follows without loss of generality.

$$E \quad \equiv \quad \exists m.\,(u = (m)^\uparrow \wedge B) \tag{6.3}$$

Note totality of the logic plays a fundamental role in this restriction: if not, the process may as well be diverging at u. We further observe that each primary name, say x_i, in A is typed as the $(\,\cdot\,)^\circ$-encoding of $\Gamma(x_i)$, which is in accordance with the type of m (which is typed as β°). This allows us to "decode" this assertion back to the one for M in the following way.

$$A' \vdash M :_u B' \quad \leftarrow\!\!\dashv \quad \langle\!\langle M \rangle\!\rangle_u \vdash \mathbf{rely}\ A^{\overline{\Gamma^\circ}}\ \mathbf{guar}\ E^{u:(\alpha^\circ)^\uparrow}. \tag{6.4}$$

where A' and B' are a suitably defined pre-image of A and B. Inheriting from the process encoding, the assertion uses the name, u, called *anchor*, in addition to the variables in the (base of the) program and auxiliary names. An anchor is used for representing the point of operation (hence specification) of M. As we mentioned in Introduction, a similar idea is found in the treatment of returned values of procedures in Hoare Logic [50, 69]: a difference is that we start from names as central elements of the logic rather than add them, leading to the uniform treatment of higher-order procedures. Note the "rely" formula is placed in the position of the type environment (where variables are primary names), while the original "guarantee" formula is placed in the position of the type of M (in which the anchor is the sole primary name).

Observe both positive and negative names in an assertion for the encoding are in replicated modes (due to the totality of the type of the anchor). Further, in translation, all $\uparrow$-types are unary except for the atomic types, so the result of invocation, $x \bullet y$, can always be "pealed off" to a replicated behaviour, unless it diverges. This means we can in effect neglect all indirections at linear types, arriving at the following simple syntax of assertions.

$$\begin{array}{lcl} e & ::= & x^\alpha \mid \mathbf{n} \mid \bot^\alpha \mid e_1 \bullet e_2 \mid e_1 + e_2 \mid e_1 \times e_2 \mid \ldots \\ A & ::= & e_1 = e_2 \mid \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \supset A_2 \mid \forall x.A \mid \exists x.A \end{array}$$

$\perp^\alpha$ denotes divergence (which is seldom used in specifications but is needed for semantic completeness), often written $\perp$. We write $e \Downarrow$ for $e \neq \perp$, which corresponds to $a \Downarrow$ in the process logic (given in (4.4), §4.4, Page 44).

The well-typedness of formulae are easily defined, which we omit. We thus arrive at the shape of the sequents of the logic, of which there are two kinds:

- $A \vdash M^{\Gamma;\alpha} :_u B$ (for provability),
- $A \models M^{\Gamma;\alpha} :_u B$ (for validity).

In both sequents, we assume the following well-formedness conditions:

- $\Gamma \vdash M : \alpha$.
- $u \notin \mathsf{fn}(A) \cup \mathsf{fn}(\Gamma)$ and $\mathsf{fn}(\Gamma) \cap \mathsf{fn}(B) = \emptyset$.
- For some Θ, A (resp. B) are well-typed under $\Gamma \cdot \Theta$ (resp. under $u:\alpha \cdot \Theta$).

The *primary names in* A are those occurring in Γ, while *the primary name of* B is the anchor u. Those names which are not primary are *auxiliary*. We often omit type annotation in examples.

Let us look at a couple of simple assertions (some have already appeared in Section 2). First, an assertion for the identity function.

$$\mathsf{T} \vdash \lambda x^\alpha.x :_u \forall y^\alpha.\ u \bullet y = y \tag{6.5}$$

which says that the given program, when it is applied to any well-typed argument y of type α, will return that argument as a result (we in fact should annotate "$\lambda x^\alpha.x$" as "$(\lambda x^\alpha.x)^{\Gamma;\alpha\Rightarrow\alpha}$", as well as, in the formula, u as $u^{\alpha\Rightarrow\alpha}$ and each y as y^α, even though omitting these annotations is harmless). It is instructive to show how this sequent is encoded into the process logic. Since the program under consideration is a value, we first take the mapping $\langle\!\langle \cdot \rangle\!\rangle^{\mathrm{val}}$.

$$!u(xm).\overline{m}(c)[c \to x]^{\alpha^\circ} \vdash \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ \forall y.\ u \bullet y = (y)^\uparrow. \tag{6.6}$$

Note the linear output $(y)^\uparrow$ has become simply y in (6.5), which is a simplification by the totality assumption. From this we reach the encoding via $\langle\!\langle \cdot \rangle\!\rangle$:

$$\overline{e}(u)!u(xm).\overline{m}(c)[c \to x]^{\alpha^\circ} \vdash \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ \exists u.\ (e = (u)^\uparrow \wedge \forall y.\ u \bullet y = (y)^\uparrow) \tag{6.7}$$

which may be considered as the arch-form of (6.5).

Another simple example follows, which uses a non-trivial assumption on a higher-order variable.

$$\forall x.Even(f \bullet x) \vdash (f3) + 1 :_u Odd(u) \tag{6.8}$$

where $Even(n)$ (resp. $Odd(n)$) says n is even (resp. odd), e.g. $Even(x) \stackrel{\text{def}}{=} \exists n.(x = 2 * n)$. We again omit the type information: to be precise, $(f3) + 1$ may be written as $((f3) + 1)^{f:\mathbb{N}\Rightarrow\mathbb{N},\ \mathbb{N}}$, while u should be $u^{\mathbb{N}}$. In the process logic this should become (with some sound simplification of the encoding):

$$\begin{aligned}\overline{f}(xc)([\![3]\!]_x | c(y).\overline{y}(e)e[\&_n \overline{u}\mathtt{in}_{n+1}] \vdash\ &\textbf{rely}\ \forall x.\ \exists i.\ (f \bullet x = \mathtt{in}_i()^\uparrow \wedge Even(i)) \\ &\textbf{guar}\ \exists j.\ (u = \mathtt{in}_j()^\uparrow \wedge Odd(j))\end{aligned}$$

The indirection via linear output complicates the formula;,yet we can precisely read out the high-level assertion from the low-level one. More examples of assertions in the PCF$_v$-logic would appear in Section 6.6.

6.4 Assertions for PCF$_v$ (2): Semantics

In this subsection we briefly illustrate the semantics of the PCF$_v$-logic. There are several ways to define its semantics, all of which arrive at the same satisfaction relation $A \models M :_u B$. Here we present the one which closely follows, hence elucidates the connection with, that of the affine process logic. An alternative method, which uses familiar CPOs for call-by-value computation, is given in Appendix B.

Intuitively, an assertion in the total PCF$_v$-logic, $A \models M^{\Gamma;\alpha} :_u B$, means that, for any closed values of types Γ satisfying A, the result of substituting them for variables in M leads to the behaviour satisfying B, under an arbitrary interpretation of auxiliary names in A and B. We substantiate this informal reading in the following in a simplest possible way.

Let $\lambda, \kappa, \ldots$ range over the $\cong_\lambda$-congruence classes of typed closed terms, which we call *behaviours*. We write λ^α if λ is of type α. For each type α, there is a unique congruent class of the diverging terms, which we write $\bot^\alpha$ or simply $\bot$. If $\lambda \neq \bot$, we say λ is a *total behaviour*.

Given $\lambda^{\alpha \Rightarrow \beta}$ and κ^α, $\lambda \bullet \kappa$ is given as the congruence class including MN for $M \in \lambda$ and $N \in \kappa$. Arithmetic operations etc. are similarly defined. A *model* ξ maps names to total behaviours. An *interpretation* $\mathcal{I}$ maps names to both total and non-total behaviours (as we have discussed in §4.4, Page 44, this generality is necessary for semantic completeness of assertions, even though names for immediately divergent behaviours are seldom needed in practice). Given a model ξ and an interpretation $\mathcal{I}$, the map $[\![e]\!]_{\mathcal{I}\cdot\xi}$ is defined in the obvious way. The satisfaction relation $\xi \models^{\mathcal{I}} A$ is standard. We then write $\lambda \models^{\mathcal{I}}_u A$ (read: *λ with anchor u satisfies A under $\mathcal{I}$*) when $u : \lambda \models^{\mathcal{I}} A$. Finally with M closed, $M \models^{\mathcal{I}}_u A$ stands for $\exists\lambda.\ (M \in \lambda \wedge \lambda \models^{\mathcal{I}}_u A)$. We also write $\vec{x} : \vec{V} \models^{\mathcal{I}} A$ for $\vec{x} : \vec{\lambda} \models^{\mathcal{I}} A$ with $V_i \in \lambda_i$. We can now define:

Definition 8. (semantics of assertions) $A \models M^{\vec{x}:\vec{\beta}\,;\,\alpha} :_u B$ *iff, under each well-typed $\mathcal{I}$ and for each $\vec{V}^{\vec{\beta}}$ such that $\vec{x}:\vec{V} \models^{\mathcal{I}} A$, we have $M[\vec{V}/\vec{x}] \models^{\mathcal{I}}_u B$.*

The validity in the PCF$_v$-logic is faithfully embeddable into that of the process logic. The translation of a formulae from the total PCF$_v$-Logic to the total process logic is written $\langle\!\langle A \rangle\!\rangle^{\vec{v}}$, where $\vec{v}$ are primary names in A (which need separate treatment since they always represent total behaviours). The base case of the translation follows, using the map $\langle\!\langle e \rangle\!\rangle^{\vec{y}}_x$ defined later. Below and henceforth we always choose fresh names for newly introduced names.

$$\langle\!\langle e_1 = e_2 \rangle\!\rangle^{\vec{y}} \stackrel{\text{def}}{=} \exists x_1 x_2.\ (x_1 = x_2\ \wedge\ \langle\!\langle e_1 \rangle\!\rangle^{\vec{y}}_{x_1}\ \wedge\ \langle\!\langle e_2 \rangle\!\rangle^{\vec{y}}_{x_1})$$

The rest of the translation of formulae is compositional with respect to each logical connective and quantifier. The map $\langle\!\langle e \rangle\!\rangle^{\vec{y}}_x$ translates a term in the PCF$_v$-logic to a formula[3], defined inductively as follows. Below $a\Uparrow$ stands for $\neg\,(a\Downarrow)$;

[3] Terms in the PCF$_v$-logic include composite values, which in general cannot be expressed as terms in the process logic.

while $\underline{u \text{ gives } e}$ stands for $\exists c.\,(u = (c)^{\uparrow} \wedge c \bullet \varepsilon = \mathtt{in}_e()^{\uparrow})$.

$$\langle\!\langle x \rangle\!\rangle_u^{\vec{v}} \stackrel{\text{def}}{=} \begin{cases} u = x & (x \notin \{\vec{v}\}) \\ u = (x)^{\uparrow} & (x \in \{\vec{v}\}) \end{cases}$$

$$\langle\!\langle \mathbf{op}(\vec{e}) \rangle\!\rangle_u^{\vec{v}} \stackrel{\text{def}}{=} \exists \vec{x}.\,(\wedge_i \langle\!\langle e_i \rangle\!\rangle_{x_i}^{\vec{v}} \;\wedge\; (\vee_i x_i \Uparrow \supset u \Uparrow) \;\wedge$$
$$\forall \vec{y}.(\wedge_i \underline{x_i \text{ gives } y_i} \supset \underline{u \text{ gives } \mathbf{op}(\vec{y})}\,))$$

$$\langle\!\langle e_1 \bullet e_2 \rangle\!\rangle_u^{\vec{v}} \stackrel{\text{def}}{=} \exists x_1 x_2.\,(\wedge_i \langle\!\langle e_i \rangle\!\rangle_{x_i}^{\vec{v}} \;\wedge\; (\vee_i x_i \Uparrow \supset u \Uparrow) \;\wedge$$
$$\forall y_1 y_2.\,(\wedge_i x_i = (y_i)^{\uparrow} \supset u = y_1 \bullet y_2))$$

$$\langle\!\langle \bot \rangle\!\rangle_u^{\vec{v}} \stackrel{\text{def}}{=} u \Uparrow$$

Above **op** is assumed to be total and strict in each argument (when **op** is nullary the map is for a constant). The translation shows the process of decomposing the coalesced PCF$_v$-types into their affine (partial) and replicated (total) parts. This process makes the mapping complex. The resulting complexity in turn suggests abstraction and convenience in specification we obtain when we move from the fine-grained process logic to a logic tailored for a specific programming language. Below we observe $\exists m.(u = (m)^{\uparrow} \wedge \langle\!\langle B \rangle\!\rangle^u[m/u]) \supset \langle\!\langle B \rangle\!\rangle^{\varepsilon}$, allowing us to simplify the guarantee formula.

Theorem 5. (logical full abstraction) *Let $\Gamma = \vec{x}:\vec{\tau}$. Then $A \models M^{\Gamma;\alpha} :_m B$ iff $\langle\!\langle M \rangle\!\rangle_u^{\overline{\Gamma^\circ};u:(\alpha^\circ)^{\uparrow}} \models$ **rely** $\langle\!\langle A \rangle\!\rangle^{\vec{x}}$ **guar** $\langle\!\langle B \rangle\!\rangle^{\varepsilon}$ for each well-typed $\Gamma \vdash M : \alpha$.*

Proof. Given a total behaviour λ, let $\langle\!\langle \lambda \rangle\!\rangle^{\text{val}}$ denote the result of translating a term in λ into a process, say P, and then taking the abstract process including P. Further write $\langle\!\langle \xi \rangle\!\rangle^{\text{val}}$ for the point-wise extension of $\langle\!\langle \lambda \rangle\!\rangle^{\text{val}}$ (resulting in a model in the process logic), similarly for $\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}$. We first observe, for each well-typed, closed M, that:

$$M \models_u^{\mathcal{I}} A \quad \equiv \quad M \Downarrow V \wedge \langle\!\langle V \rangle\!\rangle_u^{\text{val}} \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}} A^\circ. \tag{6.9}$$

For this we show: **(a)** $M \models_{\mathsf{b}} \lambda$ iff $\langle\!\langle M \rangle\!\rangle \models_{\mathsf{b}} u : \langle\!\langle \lambda \rangle\!\rangle$, and **(b)** $\xi \models^{\mathcal{I}} A$ iff $\langle\!\langle \xi \rangle\!\rangle^{\text{val}} \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}} A$ (hence $\lambda \models_u^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}} A$ iff $\langle\!\langle \lambda \rangle\!\rangle_u^{\text{val}} \models^{\mathcal{I}} A^\circ$). (a) is easy by induction on types, while (b) is by induction on formulae in the PCF$_v$-logic, using Proposition 4 (3) (full abstraction) when A is $e_1 = e_2$. We can now infer as follows. For the "if" direction, let $\langle\!\langle M \rangle\!\rangle_u \models$ **rely** A° **guar** B' with $B' \stackrel{\text{def}}{=} \exists m.(u = (m)^{\uparrow} \wedge B^\circ)$. Below we write $\langle\!\langle \vec{V} \rangle\!\rangle_{\vec{x}}^{\text{val}}$ for the n-fold parallel composition of $\langle\!\langle V_i \rangle\!\rangle_{x_i}^{\text{val}}$.

$$\begin{array}{lll}
\vec{x}:\vec{V} \models^{\mathcal{I}} A & \supset \;\; \langle\!\langle \vec{V} \rangle\!\rangle_{\vec{x}}^{\text{val}} \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}} A^\circ & \text{((6.9) above)} \\
& \supset \;\; (\boldsymbol{\nu}\,\vec{x})(\langle\!\langle M \rangle\!\rangle_u | \langle\!\langle \vec{V} \rangle\!\rangle_{\vec{x}}^{\text{val}}) \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}} B' & \text{(Assumption)} \\
& \supset \;\; \langle\!\langle M[\vec{V}/\vec{x}] \rangle\!\rangle_u \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}} B' & \text{(Replication Theorem)} \\
& \supset \;\; M[\vec{V}/\vec{x}] \Downarrow W \wedge \langle\!\langle W \rangle\!\rangle_u^{\text{val}} \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}} B^\circ & \text{(Proposition 4 (1))} \\
& \supset \;\; M[\vec{V}/\vec{x}] \models_m^{\mathcal{I}} B & \text{((6.9) above).}
\end{array}$$

For the "only if" direction, assume $A \models M^{\Gamma;\alpha} :_m B$ with $\Gamma = \vec{x}:\vec{\tau}$. If $A \equiv \mathsf{F}$, the statement is vacuous. Assume not. By Proposition 4 (3) (definability), we can set $R_i \cong_\pi \langle\!\langle V_i \rangle\!\rangle^{\mathrm{val}}_{x_i}$ for some V_i.

$$
\begin{array}{lcll}
\Pi_i R_i^{x_i} \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\mathrm{val}}} A^\circ & \supset & \langle\!\langle \vec{V} \rangle\!\rangle^{\mathrm{val}}_{\vec{x}} \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\mathrm{val}}} A^\circ & (R_i \cong \langle\!\langle V_i \rangle\!\rangle^{\mathrm{val}}_{x_i}) \\
 & \supset & \vec{x}:\vec{V} \models^{\mathcal{I}} A & ((6.9) \text{ above}) \\
 & \supset & M[\vec{V}/\vec{x}] \Downarrow W \wedge \langle\!\langle W \rangle\!\rangle^{\mathrm{val}}_u \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\mathrm{val}}} B' & (\text{assumption}) \\
 & \supset & (\boldsymbol{\nu}\, \vec{x})(\langle\!\langle M \rangle\!\rangle_u | \langle\!\langle \vec{V} \rangle\!\rangle^{\mathrm{val}}_{\vec{x}}) \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\mathrm{val}}} B' & (\text{Replication}) \\
 & \supset & (\boldsymbol{\nu}\, \vec{x})(\langle\!\langle M \rangle\!\rangle_u | \Pi_i R_i) \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\mathrm{val}}} B'. & (R_i \cong \langle\!\langle V_i \rangle\!\rangle^{\mathrm{val}}_{x_i}).
\end{array}
$$

Note the definability is essential for the second direction. □

6.5 Proof Rules for $\mathbf{PCF}_v$

Below we show how the correspondence between the assertions in the PCF$_v$-logic and those in the process logic leads to simple and natural proof rules for PCF$_v$, including its basic extensions such as `letrec`, products and sums. *Henceforth we always assume all assertions occurring in each rule are well-typed and a primary name and an auxiliary name are never mixed in the antecedent (cf. Convention 1, §3.5, Page 31). $i, j, \ldots$ exclusively range over auxiliary names.*

$$
[\mathit{Var}]\ \frac{-}{A[x/u] \vdash x :_u A} \qquad [\mathit{Num}]\ \frac{-}{A[\mathbf{n}/u] \vdash \mathbf{n} :_u A} \qquad [\mathit{Succ}]\ \frac{A \vdash M :_v B[v{+}1/u]}{A \vdash \mathsf{succ}(M) :_u B}
$$

We can read these rules naturally from the viewpoint of PCF$_v$-computation. For example, $[Var]$ says that, if something can be said about what x denotes in the environment, then the same thing can be said about x as a term, named as u. They also precisely correspond to the proof rules for the total process logic via encoding. For example, $[Var]$ is direct from the assertion (3.12) (cf. §3.7) for a replicated copycat. For arithmetical/boolean operations, we have, for example:

$$
[\mathit{Add}]\ \frac{A \vdash M :_v B_1 \quad A \vdash N :_w B_2 \quad A \wedge (B_1 \wedge B_2) \supset B[v+w/u]}{A \vdash M + N :_u B}
$$

Similarly for the equality operator:

$$
[\mathit{Eq}]\ \frac{A \vdash M^{\mathbb{N}} :_{v_1} B_1 \quad A \vdash N^{\mathbb{N}} :_{v_2} B_2 \quad A \wedge (B_1 \wedge B_2) \supset B[v_1{=}v_2/b]}{A \vdash M{=}N :_b B}
$$

From these rules one can guess the general form of the proof rule for an n-ary operator (of which constants and unary/binary operators are special cases), as given below. In the following rule, we assume op also appears as an operator in the logical language. Let i range over $\{1, .., n\}$ below.

$$
[\mathit{Op}]\ \frac{A \vdash M_i :_{m_i} B_i \quad A \wedge (\wedge_i B_i)\ \supset\ B[\mathsf{op}(m_1, .., m_n)/u]}{A \vdash \mathsf{op}(M_1, .., M_n) :_u B}
$$

As far as op is a first-order total operation (mapping first-order values to a first-order value), each instance of this rule is sound via its process encoding.

Next we move to the proof rule for the conditional, which says that, under A, if a boolean term M (named as b) satisfies A', and B holds for:

$$[Var]\ \frac{-}{A[x/u] \vdash x :_u A} \qquad [Num]\ \frac{-}{A[\mathbf{n}/u] \vdash \mathbf{n} :_u A} \qquad [Boolean]\ \frac{-}{A[\mathbf{b}/u] \vdash \mathbf{b} :_u A}$$

$$[Op]\ \frac{A \vdash M_i :_{m_i} B_i \ \ (1 \le i \le n) \quad A \wedge (\wedge_i B_i) \supset B[\mathsf{op}(m_1,..,m_n)/u]}{A \vdash \mathsf{op}(M_1,..,M_n) :_u B}$$

$$[If]\ \frac{A \vdash M :_b A' \quad A \wedge A'[\mathsf{T}/b] \vdash N_1 :_u B \quad A \wedge A'[\mathsf{F}/b] \vdash N_2 :_u B}{A \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 :_u B}$$

$$[Abs]\ \frac{A^{\text{-}x\vec{i}} \wedge A_1^x \vdash M :_m A_2 \quad \forall x\vec{i}.(A_1 \supset A_2[u \bullet x/m]) \supset B}{A \vdash \lambda x.M :_u B}$$

$$[App]\ \frac{A \vdash M :_m B_1 \quad A \vdash N :_x B_2 \quad B_1 \supset B_2 \supset B[m \bullet x/u] \wedge m \bullet x \Downarrow}{A \vdash MN :_u B}$$

$$[Rec]\ \frac{A \vdash \lambda y.M :_u B(0) \qquad A \wedge B(i)[x/u] \vdash \lambda y.M :_u B(i+1)}{A \vdash \mu x.\lambda y.M :_u \forall i.B(i)}$$

$$[Consequence]\ \frac{A \supset A_0 \quad A_0 \vdash M :_u B_0 \quad B_0 \supset B}{A \vdash M :_u B}$$

Fig. 11. Proof Rules for PCF$_v$: total correctness (main rules)

(1) N_1 under the assumption that A' holds; and
(2) N_2 under the assumption that A' does not hold,

then, again under A, surely B holds for $\mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2$.

$$[If]\ \frac{A \vdash M :_b A' \quad A \wedge A'[\mathsf{T}/b] \vdash N_1 :_u B \quad A \wedge A'[\mathsf{F}/b] \vdash N_2 :_u B}{A \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 :_u B}$$

Note the rule keeps clean symmetry, as in the corresponding rule in Hoare logic. Observe also $b \notin \mathsf{fn}(A)$ by the well-formedness of $A \vdash M :_b A'$. Another observation is that $A \vdash M :_b A'$ indicates (as far as A is non-trivial) M terminates. Thus, in a closed term, the conditional branch can surely be evaluated, reaching one of N_i, which in turn is guaranteed to terminate and satisfies B.

We now move to the three key rules for PCF$_v$-logic, abstraction, application and recursion. Below we recall A^x indicates the only primary name in A is x, while $A^{\text{-}x\vec{i}}$ says no names from $x\vec{i}$ occur in A.

$$[Abs]\ \frac{A^{\text{-}x\vec{i}} \wedge A_1^x \vdash M :_m A_2 \quad \forall x\vec{i}.(A_1 \supset A_2[u \bullet x/m]) \supset B}{A \vdash \lambda x.M :_u B}$$

The rule says that, whenever M named as m satisfies A_2 relying on A (which is not about x) and A_1 (which is about x), then $\lambda x.M$ named u has the behaviour such that, whenever u is fed with an argument x satisfying A_1, returns the result

$(u \bullet x)$ which satisfies A_2 for m. The rule is based on $(\mathsf{In}^!\text{-}\forall)$ discussed in §3.6, 35, via the encoding $\langle\!\langle \lambda x.M \rangle\!\rangle \stackrel{\text{def}}{=} \overline{u}(c)!c(xm).\langle\!\langle M \rangle\!\rangle_m$. As in $(\mathsf{In}^!\text{-}\forall)$, and as we shall see with concrete examples, the quantification on $\vec{i}$ is often convenient in practice. As in §3.6, we can decompose this rule into:

$$[\mathit{Abs\text{-}sub}]\frac{A \vdash M :_m B,\ i \text{ fresh}}{A[i/x] \vdash \lambda x.M :_u B[m \bullet i/u]} \quad [\supset]\frac{A \wedge B \vdash M^{\Gamma;\alpha} :_u C,\ \mathsf{fn}(B) \cap \mathsf{fn}(\Gamma) = \emptyset}{A \vdash M^{\Gamma;\alpha} :_u B \supset C}.$$

However we prefer $[\mathit{Abs}]$ because of their convenience in reasoning. The provability does not differ in the presence of $[\supset]$ and $[\mathit{Aux}\text{-}\forall]$ (the latter follows the rule with the same name for the process logics, cf. §3.5, Page 31), and is given later).

The next rule is the one for application.

$$[\mathit{App}]\ \frac{A \vdash M :_m B_1 \quad A \vdash N :_x B_2 \quad B_1 \supset B_2 \supset B[m \bullet x/u] \wedge m \bullet x \Downarrow}{A \vdash MN :_u B}$$

The rule says that, if M, named as m, satisfies B_1, and N, named as x, satisfies B_2, and B_1 and B_2 say that m applies to x converges, then MN named as u satisfies B whenever B is a consequence of B_1 and B_2 where, for making B speak about m and x as in $B_{1,2}$, u in B is replaced by $m \bullet x$.

It is instructive to decompose $[\mathit{App}]$ into inferences in the process logic via encoding. For brevity, assume M and N are values, so that the encoding becomes:

$$\langle\!\langle MN \rangle\!\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\langle\!\langle M \rangle\!\rangle^{\text{val}}_m | \overline{m}(xu').(\langle\!\langle N \rangle\!\rangle^{\text{val}}_x | [u' \to u])).$$

Note m of $\langle\!\langle M \rangle\!\rangle^{\text{val}}_m$ is typed as $(\alpha \Rightarrow \beta)^\circ$, x of $\langle\!\langle N \rangle\!\rangle^{\text{val}}_x$ is typed as α°, and u is typed as $(\beta^\circ)^\uparrow$. We can now derive the proof rule for application as follows.

1. $\langle\!\langle M \rangle\!\rangle^{\text{val}}_m \vdash \textbf{rely}\ \langle\!\langle A \rangle\!\rangle^{\vec{x}}\ \textbf{guar}\ \langle\!\langle B_1 \rangle\!\rangle^m$	(Ant 1, IH)
2. $\langle\!\langle N \rangle\!\rangle^{\text{val}}_x \vdash \textbf{rely}\ \langle\!\langle A \rangle\!\rangle^{\vec{x}}\ \textbf{guar}\ \langle\!\langle B_1 \rangle\!\rangle^x$	(Ant 2, IH)
3. $[u' \to u] \vdash \textbf{rely}\ \langle\!\langle B \rangle\!\rangle^\varepsilon[u'/u]\ \textbf{guar}\ \langle\!\langle B \rangle\!\rangle^\varepsilon$	(copycat law)
4. $\vdash \overline{m}(xu')(\langle\!\langle N \rangle\!\rangle^{\text{val}}_x | [u' \to u])$ $\textbf{rely}\ \langle\!\langle B_1 \rangle\!\rangle^m \wedge \langle\!\langle A \rangle\!\rangle^{\vec{x}}\ \textbf{guar}\ \langle\!\langle B \rangle\!\rangle^\varepsilon$	(2, 3, Ant 3, (Out))
5. $\vdash \langle\!\langle M \rangle\!\rangle^{\text{val}}_m | \overline{m}(xu').(\langle\!\langle N \rangle\!\rangle^{\text{val}}_x | [u' \to u])$ $\textbf{rely}\ \langle\!\langle A \rangle\!\rangle^{\vec{x}}\ \textbf{guar}\ \langle\!\langle B \rangle\!\rangle^\varepsilon$	(1, 4, (Par))
6. $\langle\!\langle MN \rangle\!\rangle_u \vdash \textbf{rely}\ \langle\!\langle A \rangle\!\rangle^{\vec{x}}\ \textbf{guar}\ \langle\!\langle B \rangle\!\rangle^\varepsilon$	(Res)

Note the inference involves both (Out) and (Par)/(Res) non-trivially. In fact, these two elements of $[\mathit{App}]$ can be decoupled at the level of PCF$_v$, using the proof rule for the `let` construct:

$$[\mathit{Let}]\ \frac{A \vdash M :_x A' \quad A \wedge A' \vdash N :_u B}{A \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N :_u B}.$$

The rule can be derived from $[\mathit{Abs}]$ and $[\mathit{App}]$ via the standard translation: $\mathtt{let}\ x = M\ \mathtt{in}\ N \stackrel{\text{def}}{=} (\lambda x.N)M$. As a process, however, $\mathtt{let}\ x = V\ \mathtt{in}\ N$ can be directly translated into:

$$\langle\!\langle \mathtt{let}\ x = V\ \mathtt{in}\ N \rangle\!\rangle_u \quad \stackrel{\text{def}}{=} \quad (\boldsymbol{\nu}\, c)(\langle\!\langle V \rangle\!\rangle_c | c(x).\langle\!\langle N \rangle\!\rangle_u) \quad \approx \quad (\boldsymbol{\nu}\, x)(\langle\!\langle V \rangle\!\rangle_x^{\text{val}} | \langle\!\langle N \rangle\!\rangle_u),$$

Thus [*Let*] only needs (Par)/(Res) to infer in the process logic. We can then decompose MN as $\mathtt{let}\ x = M\ \mathtt{in}\ xN$, where the inference for $\langle\!\langle xN \rangle\!\rangle_u$ only uses (Out) substantially.

We turn to the rule for recursion. Below n is a fresh name of $\mathbb{N}$-type.

$$[Rec]\ \frac{A^{\text{-}x} \vdash \lambda y.M :_u B(0) \qquad A \wedge B(n)[x/u] \vdash \lambda y.M :_u B(n+1)}{A \vdash \mu x.\lambda y.M :_u \forall n.B(n)}$$

Under A, the term satisfies, without assuming anything about x, already $B(0)$ (where $B(\ \cdot\ )$ is a formula with a hole). The term named as u also satisfies $B(n+1)$, if we assume $B(n)[x/u]$ on x in addition to A. So we expect it satisfies $B(n)$ for each n. The [*Rec*] rule closely follows (Rec) in the total process logic (cf. Section 4.5, Page 46). The latter further suggests that the body of recursion can be any value of non-atomic types (recursion on atomic types is prohibited, remembering they are encoded as linear types, cf. §6.3, Page 65). Relatedly, the `letrec`, with the standard operational semantics [32], has a clean proof rule.

$$[Letrec]\ \frac{\begin{array}{c} A \vdash V[y/x]^{\Gamma;\alpha} :_x E(0) \\ A \wedge E(n)[y/x] \vdash V[y/x] :_x E(n+1) \\ A \wedge \forall n.E(n) \vdash M :_u B \end{array}}{A \vdash \mathtt{letrec}\ x = V\ \mathtt{in}\ M :_u B}$$

The `letrec` construct has a natural representation using wires [41], which is in a direct correspondence with recursion $\mu\vec{x} = \vec{y}.P$ in the process. We show this for a generalised version of the above rule, the proof rule for multiple recursion. Below let each V_i have an arrow type.

$$[Mletrec]\ \frac{\begin{array}{c} A \vdash V_i[y_i/x_i] :_x E_i(0) \\ A \wedge E(n)[\vec{y}/\vec{x}] \vdash V_i :_{x_i} E_i(n+1) \\ A \wedge \forall n.E(n) \vdash M :_u B \end{array}}{A \vdash \mathtt{letrec}\ \vec{x} = \vec{V}\ \mathtt{in}\ M :_u B}\,(E(n) \stackrel{\text{def}}{=} \wedge_i E_i(n))$$

We justify the rule above using (Rec) in Section 4. We first observe:

$$\langle\!\langle \mathtt{letrec}\ \vec{x} = \vec{V}\ \mathtt{in}\ M \rangle\!\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,\vec{x})(\langle\!\langle M \rangle\!\rangle_u \mid \mu\vec{y}{=}\vec{x}.\,\Pi_i \langle\!\langle V_i[y_i/x_i] \rangle\!\rangle_{x_i}^{\text{val}}) \qquad (6.10)$$

From the encoding of the antecedents in [*Mletrec*], (Rec) in Figure 5 directly gives us $\mu\vec{y}{=}\vec{x}.\,\Pi_i \langle\!\langle V_i[y_i/x_i] \rangle\!\rangle_{x_i}^{\text{val}} \vdash \mathbf{rely}\ A^\circ\ \mathbf{guar}\ \forall n.E(n)$, to which we apply (Par) and (Res) to get the process (6.10), arriving at the conclusion of [*Mletrec*].

Finally we list the consequence rule and other structural rules (the last four rules may not add provability). Their names follow the corresponding rules in

$$
\begin{array}{lrcl}
\textbf{(ext)} & \forall y.\, x_1 \bullet y = x_2 \bullet y & \supset & x_1 = x_2 \\
\textbf{(}\bot\textbf{-left)} & \bot^{\alpha \Rightarrow \beta} \bullet x^{\alpha} = \bot^{\beta} & & \\
\textbf{(}\bot\textbf{-right)} & x^{\alpha \Rightarrow \beta} \bullet \bot^{\alpha} = \bot^{\beta} & &
\end{array}
$$

Fig. 12. Axioms for Strict $\bullet$

the process logic, cf. §3.6, Page 36.

$$[\mathit{Conseq}]\ \frac{A \supset A' \quad A' \vdash M :_u B' \quad B' \supset B}{A \vdash M :_u B}$$

$$[\mathit{Aux}\text{-}\forall]\ \frac{A^{\text{-}i} \vdash M :_u B}{A \vdash M :_u \forall i.B} \qquad [\mathit{Aux}\text{-}\exists]\ \frac{A \vdash M :_u B^{\text{-}i}}{\exists i.A \vdash M :_u B}$$

$$[\mathit{Aux}\text{-}\wedge,\vee]\ \frac{A \vdash M^{\Gamma;\alpha} :_u B \quad \mathsf{fn}(C) \cap (\mathsf{dom}(\Gamma) \cup \{u\}) = \emptyset}{A \star C \vdash M :_u B \star C} \quad (\star \in \{\wedge, \vee\})$$

$$[\mathit{Conj}]\ \frac{A \vdash M :_u B_{1,2}}{A \vdash M :_u B_1 \wedge B_2} \qquad [\mathit{Disj}]\ \frac{A_{1,2} \vdash M :_u B}{A_1 \vee A_2 \vdash M :_u B}$$

To put the consequence rule to real use (as well as all other rules which use validity of formulae as their side conditions), we may as well use inference rules for the underlying partial applicative structure, in addition to the standard rules for the predicate calculus with equality and number theory. Three basic rules are listed in Figure 12, all easily justifiable from the underlying model (similar rules exist for first-order operations).

We conclude the presentation of the proof system with its key property. Below we write $B^{\bullet}_u$ for $\exists x.(u = (x)^{\uparrow} \wedge B^{\circ}[x/u])$.

Proposition 5. *Let $\Gamma \vdash M : \alpha$ with $\Gamma = \vec{x} : \vec{\alpha}$. Then $A \vdash M :_u B$ implies $\langle\!\langle M \rangle\!\rangle_u \vdash$* **rely** $\langle\!\langle A \rangle\!\rangle^{\vec{x}}$ **guar** $\langle\!\langle B \rangle\!\rangle^{\varepsilon}$.

Proof. Mechanical induction, as shown above for (App). □

Theorem 6. (soundness of PCF$_v$-logic) $A \vdash M :_u B$ *implies* $A \models M :_u B$.

Proof. Assume $A \vdash M^{\vec{x}:\vec{\tau}\,;\,\alpha} :_u B$.

$$
\begin{array}{lcll}
\vec{x}{:}\vec{V} \models^{\mathcal{I}} A & \supset & \langle\!\langle \vec{V} \rangle\!\rangle^{\text{val}}_{\vec{x}} \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}} A^{\circ} & \text{(Theorem 5)} \\
& \supset & (\boldsymbol{\nu}\, \vec{x})(\langle\!\langle M \rangle\!\rangle_u | \langle\!\langle \vec{V} \rangle\!\rangle^{\text{val}}_{\vec{x}}) \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}} B^{\bullet}_u & \text{(Prop. 5, Theorem 4)} \\
& \supset & \langle\!\langle M[V/\vec{x}] \rangle\!\rangle_u \ \models^{\langle\!\langle \mathcal{I} \rangle\!\rangle^{\text{val}}} B^{\bullet}_u & \text{(replication)} \\
& \supset & M[V/\vec{x}] \models^{\mathcal{I}} B & \text{(Theorem 5).}
\end{array}
$$

Note both directions of Theorem 5 (logical full abstraction) are used. □

The converse of Theorem 6 may not hold for the same reason as in the affine process logic (cf. Remark 2, §4.5, Page 46).

$$[Pair]\,\frac{A \vdash M_i :_{m_i} B_i \quad B_1 \wedge B_2 \supset B[\langle m_1, m_2\rangle/u]}{A \vdash \langle M_1, M_2\rangle :_u B} \qquad [Proj_1]\,\frac{A \vdash M :_m B[\pi_1(m)/u]}{A \vdash \pi_1(M) :_u B}$$

$$[In_1]\,\frac{A \vdash M :_m B_i[\mathtt{in}_1(m)/u]}{A \vdash \mathtt{in}_1(M) :_u B} \qquad [Case]\,\frac{A \vdash M :_m E \quad A^{\text{-}\vec{x}} \wedge E[\mathtt{in}_i(x_i)/m] \vdash N_i :_u B}{A \vdash \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).N_i\} :_u B}$$

Fig. 13. Proof Rules for Product and Sum

Products and Sums. In the following we briefly discuss how we can simply incorporate products and sums in our program logic. For completeness we also add the unit type, an atomic type with a single constant. We write $\alpha \times \beta$ and $\alpha + \beta$ for the product and sum of α and β, respectively. `Unit` denotes the unit type. The grammar of programs is extended as follows, with i ranging over $\{1, 2\}$.

$$M \ ::= \ () \ \mid \ \langle M_1, M_2\rangle \ \mid \ \pi_i(M) \ \mid \ \mathtt{in}_i(M) \ \mid \ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).N_i\}_{i\in\{1,2\}}$$

For brevity we usually write $\mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).N_i\}$ for the case construct, omitting the indexing set. We omit the standard reduction and typing rules [32, 96].

The logical language adds the following terms. Again i ranges over $\{1, 2\}$.

$$e \ ::= \ \ldots \ \mid \ () \ \mid \ \langle e_1, e_2\rangle \ \mid \ \pi_i(e) \ \mid \ \mathtt{in}_i(e)$$

We can dispense with $\pi_i(e)$, but having it is often convenient for reasoning. The semantics are given as the standard call-by-value sums and products [32]. From the viewpoint of models, these operators are strict and the equality on them includes extensional laws, which may as well be used as axioms for calculating validity as in Figure 13.

The semantics of the unit becomes vacuous in the total logic (since it has a unique total element), that is $e^{\mathtt{Unit}} = ()$ is equivalent to T. The encoding of these constructs in the total affine logic is easy: we only add the following clauses to the map given in Section 6.4.

$$\begin{aligned}
\langle e_1, e_2\rangle^{\circ}_u &\stackrel{\text{def}}{=} \exists xy.\ (u \bullet \varepsilon = (xy)^{\uparrow} \ \wedge \ {e_1}^{\circ}_x \ \wedge \ {e_2}^{\circ}_y) \\
\pi_1(e)^{\circ}_u &\stackrel{\text{def}}{=} \exists xr.\ (x \bullet \varepsilon = (ur)^{\uparrow} \ \wedge \ e^{\circ}_x) \\
\mathtt{in}_1(e)^{\circ}_u &\stackrel{\text{def}}{=} \exists x.\ (u \bullet \varepsilon = \mathtt{in}(x)^{\uparrow} \ \wedge \ e^{\circ}_x)
\end{aligned}$$

This translation directly suggests the proof rules for products and sums, as given in Figure 13. The rule for () is simply another constant. The soundness is mechanically checked via its mapping to the process logic.

Products and sums are naturally extended to their labelled versions, records and variants. For example, records add the term of the form $\{l_i : e_i\}$ and $e.l$, interpreted as (labelled) products and their projection. The proof rules become:

$$[Record]\frac{A \vdash M_i :_{m_i} B_i \quad \wedge B_i \supset B[\{l_i : m_i\}/u]}{A \vdash \{l_i : M_i\} :_u B} \qquad [Sel_l]\frac{A \vdash M :_m B[m.l/u]}{A \vdash M.l :_u B}$$

whose soundness can again be checked by mapping the derivation to that of the total process logic.

Remark 6. (the use of free variables in guarantee formulae) Following our discussions in Section 3 (Remark 1, Page 35), we may as well use free variables of a program in a guarantee formula, with the obvious refinement in semantics of judgements. The proof rules stay as they are. This convention tends to makes assertions and their proofs more concise. For example, the following assertion for a variable:

$$x = i \vdash x :_u u = i \tag{6.11}$$

can be simply written as:

$$\mathsf{T} \vdash x :_u u = x \tag{6.12}$$

which may express more straightforwardly what the original assertion in (6.11) indicates. As in the process logic, this does not result in any additional expressiveness, since we can mechanically translate extended assertions into those in the original format. For example, when given the assertion in (6.12), we first decompose $u = x$ into $u = i$ and $i = x$ using a fresh i, leave $u = i$ in the guarantee formula and add $i = x$ to the rely formula, resulting in (6.11). When we should deal with complex assertions, the convenience given by this extension is often quite valuable.

6.6 Reasoning Examples in PCF$_v$-Logic

We conclude this section with a couple of simple inference examples. We show the derivation of the following three assertions.

$$\mathsf{T} \vdash \lambda x.x :_u \forall x.(u \bullet x = x). \tag{6.13}$$

$$\mathsf{T} \vdash \mathit{Fact} :_u \forall n.\ u \bullet n = n!. \tag{6.14}$$

$$\mathsf{T} \vdash \mathit{Gcd} :_u \forall x, y \geq 1.gcd(u \bullet (x, y), x, y) \tag{6.15}$$

where we set:

$$\begin{aligned} \mathit{Fact} &\stackrel{\text{def}}{=} \mu f.\lambda x.\mathtt{if}\ x = 0\ \mathtt{then}\ 1\ \mathtt{else}\ x * f(x-1). \\ \mathit{Gcd} &\stackrel{\text{def}}{=} \mu f.\lambda(x, y).\ \mathtt{if}\ x \lneq y\ \mathtt{then}\ f(y-x, x)\ \mathtt{else} \\ &\qquad\qquad \mathtt{if}\ x \gneq y\ \mathtt{then}\ f(x-y, y)\ \mathtt{else}\ x\ . \end{aligned}$$

Fact computes the factorial when fed with a non-negative integer, similarly *Gcd* computes the greatest common divisor when fed with two arguments which should be positive integers. In *Gcd*, we are using a function with multiple parameter for brevity, for which we assume the standard operational semantics.

Accordingly, $\bullet$ in its specification is used with multiple parameters. The predicate $gcd(m, x, y)$ (read: *m is the g.c.d. of x and y*) is given as:

$$gcd(m,x,y) \stackrel{\text{def}}{=} Div(m,x) \wedge Div(m,y) \wedge \forall n.\ (Div(n,x) \wedge Div(n,y) \supset n \leq m).$$

where $Div(n, x)$ (read: *n divides x*) stands for $\exists i.\ n = x * i$.

Deriving the assertion (6.13) is easy. Below and henceforth we shall freely use trivial logical equivalences.

$$\begin{array}{ll}
1.\quad \mathsf{T} \wedge x = l \vdash x :_m m{=}l & \text{(Var)} \\
\hline
2.\quad \mathsf{T} \vdash \lambda x.x :_u \forall xl.(x = l \supset u \bullet x = l) & \text{(Abs)} \\
\hline
3.\quad \mathsf{T} \vdash \lambda x.x :_u \forall x.(u \bullet x = x) & \text{(Conseq)}
\end{array}$$

Recalling Remark 6, we also show how the above inference changes when we allow free variables to occur into guarantee formulae.

$$\begin{array}{ll}
1.\quad \mathsf{T} \vdash x :_m m{=}x & \text{(Var)} \\
\hline
2.\quad \mathsf{T} \vdash \lambda x.x :_u \forall x.(u \bullet x = l) & \text{(Abs)}
\end{array}$$

We thus save one step. While we do not employ this relaxed usage of names (at least in this section), accumulation of such small savings can lead to a substantial simplification of reasoning as a whole.

As the next example, we tackle the assertion (6.14). Let

$$B(f)(n) \stackrel{\text{def}}{=} f \bullet n = n!$$

as well as

$$M \stackrel{\text{def}}{=} \texttt{if}\ x{=}0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1).$$

We also use the following convention, which is often useful in practice.

Convention 3. *In the following inferences, we often use a boolean-typed expression $e^{\mathbb{B}}$ as a formula, which is convenient and which does not lose precision* as far as we know e is total *(since in that case e indeed denotes either the truth or the falsity). For example, if b is a primary name in a derived judgement, we automatically know b is total. Under this assumption, each occurrence of $b^{\mathbb{B}}$ as a formula is equivalent to $b = \mathsf{T}$ (with an implicit outermost conjunction with $e \Downarrow$, which should be implied from the preceding derivation).*

The convention needs be used with care: if e is a boolean typed term which is by itself not a primary name (especially if it is a composite term including $\bullet$ in it), e can have one of the three values, $\mathbf{t}$, $\mathbf{f}$ and $\perp$, so that we cannot use it as a formula (note however $e = \mathbf{t}$ is a well-defined formula). However, as far as it is used with care, this convention is quite useful, as we shall see below.

We start from the base case (the first three steps are somewhat trivial, but we give some detail for illustration).

1.	$x = 0 \vdash x :_m m = 0$	(Var)
2.	$x = 0 \vdash 0 :_n n = 0$	(Num)
3.	$m = 0 \wedge n = 0 \supset b[m = n/b]$	(PRED)
4.	$x = 0 \vdash x = 0 :_b b$	(1, 2, 3, Conv 3, Eq)
5.	$x = 0 \wedge \mathsf{T} \vdash 1 :_v v = 1$	(Num, Conseq)
6.	$x = 0 \wedge \mathsf{F} \vdash x * f(x-1) :_v v = 1$	(falsity)
7.	$x = 0 \vdash \texttt{if } x{=}0 \texttt{ then } 1 \texttt{ else } x * f(x-1) :_v v = 1$	(4, 5, 6, If)
8.	$\mathsf{T} \vdash \lambda x.M :_u \forall x.(x = 0 \supset u \bullet x = 1)$	(6, Abs)
9.	$\mathsf{T} \vdash \lambda x.M :_u B(u)(0)$	(Conseq)

In Line 2, (PRED) means it is derivable in the predicate calculus with equality. In Line 6, (falsity) stands for the following lemma:

$$[\mathit{falsity}]\ \frac{-}{\mathsf{F} \vdash M :_m A}$$

(as always we assume $\mathsf{F} \vdash M :_m A$ is a well-typed judgement) which can easily derive the lemma is easily y checked by rule induction.

For the induction step, remembering n ranges over non-negative integers:

1.	$B(f)(n) \wedge x = n+1 \vdash x :_m \neg(m = 0)$	(Var)
2.	$B(f)(n) \wedge x = n+1 \vdash 0 :_n n = 0$	(Num)
3.	$\neg(m = 0) \wedge n = 0 \supset \neg b[m{=}n/b]$	(PRED)
4.	$B(f)(n) \wedge x = n+1 \vdash x = 0 :_b \neg b$	(1,2,3,Conv 3,(Eq))
5.	$B(f)(n) \wedge x = n+1 \wedge \mathsf{F} \vdash 1 :_v v = (n+1)!$	(Num, Conseq)
6.	$B(f)(n) \wedge x{=}n+1 \wedge \mathsf{T} \vdash x :_m m{=}n+1$	(Var, Conseq)
7.	$B(f)(n) \wedge x{=}n+1 \wedge \mathsf{T} \vdash f :_l B(l)(n)$	(Var, Conseq)
8.	$B(f)(n) \wedge x{=}n+1 \wedge \mathsf{T} \vdash x - 1 :_k k{=}n$	(6. Subt)
9.	$B(f)(n) \wedge x = n+1 \vdash f(x-1) :_v v{=}(n-1)!$	(4, 5, App)
10.	$B(f)(n) \wedge x = n+1 \vdash x * f(x-1) :_v v{=}n!$	(3, 6, Mult)
11.	$B(f)(n) \wedge x = n+1 \vdash M :_v v{=}n!$	(1, 2, 6, If)
12.	$\mathsf{T} \wedge B(f)(n) \vdash \lambda x.M :_u B(u)(n+1)$	(7, Abs, Conseq)

Above we have used (Mult) and (Subt), which are instances of (Op) before. We can now combine the two conclusions (let them be $(*)$ and $(**)$), reaching:

1.	$\mathsf{T} \vdash \lambda x.M :_u B(u)(0)$	$(*)$
2.	$\mathsf{T} \wedge B(f)(n) \vdash \lambda x.M :_u B(u)(n+1)$	$(**)$
3.	$\mathsf{T} \vdash \mathit{Fact} :_u \forall n.B(u)(n)$	(1, 2, (Rec)) .

We next move to (6.15), an assertion on GCD. A reasoning for essentially the same property is given in [11] with elucidating illustration, but without compositional proof rules. We let:

$$G(m,i,j)(n) \stackrel{\text{def}}{=} i,j \geq 1 \wedge i+j \leq n+1 \;\supset\; gcd(m,i,j)$$

$$M \stackrel{\text{def}}{=} \texttt{if } x \lneq y \texttt{ then } f(y-x,x) \texttt{ else if } x \gneq y \texttt{ then } f(x-y,y) \texttt{ else } x \;.$$

The base case follows.

1.	$x=y=i=1 \wedge \mathsf{F} \vdash f(y-x,x) :_m G(m,i,i)(0)$	(falsity)
2.	$x=y=i=1 \wedge \mathsf{F} \vdash f(x-y,y) :_m G(m,i,i)(0)$	(falsity)
3.	$x=y=i=1 \wedge \mathsf{T} \vdash x :_m G(m,i,i)(0)$	(Var, Conseq)
4.	$x=y=i=1 \vdash M :_m G(m,i,i)(0)$	(If, If, Conseq)
5.	$\mathsf{T} \vdash \lambda(x,y).M :_u \forall x,y,i.\ (x=y=i=1 \supset G(u \bullet (x,y),i,i)(0))$	(Mabs)
6.	$\mathsf{T} \vdash \lambda(x,y).M :_u \forall x,y.\ G(u \bullet (x,y),x,y)(0)$	(Conseq) .

In Line 3, note $G(m,i,j)(1)$ is equivalent to $gcd(m,1,1))$, which in turn is equivalent to $m=1$, In Line 5, $[\mathit{Mabs}]$ is the following extension of $[\mathit{Abs}]$ (the rule is easily justifiable by combining $[\mathit{Abs}]$ with $[\mathit{Proj}]$).

$$[\mathit{Mabs}]\ \frac{A^{\text{-}\vec{x}\vec{i}} \wedge A_1^{\vec{x}} \vdash M :_m A_2 \qquad \forall \vec{x}\vec{i}.\,(A_1 \supset A_2[u \bullet (\vec{x})/m]\,) \supset B}{A \vdash \lambda(\vec{x}).M :_u B}$$

Next we turn to the induction. Let:

$$G'(f)(n) \stackrel{\text{def}}{=} \forall x,y.\ G(f \bullet (x,y),x,y)(n),$$

which says, for each positive x and y such that $x+y \leq n+1$, $f \bullet (x,y)$ is the g.c.d. of x and y. We also let $C \stackrel{\text{def}}{=} (x=i \wedge y=j \wedge i+j \leq n+2)$, $E_1 \stackrel{\text{def}}{=} i \lneq j$, $E_2 \stackrel{\text{def}}{=} i \gneq j$ and $E_3 \stackrel{\text{def}}{=} i=j$. In the following inferences, we often do not mention trivial applications of (Conseq).

1.	$G'(f)(n) \;\wedge\; C \;\wedge\; E_1 \vdash f :_p G'(p)(n)$	(Var)
2.	$G'(f)(n) \;\wedge\; C \;\wedge\; E_1 \vdash y - x :_q q = j - i$	(Var, Var, Subt)
3.	$G'(f)(n) \;\wedge\; C \;\wedge\; E_1 \vdash x :_r r = i$	(Var)
4.	$G'(f)(n) \;\wedge\; C \;\wedge\; E_1 \vdash f(y - x, x) :_m G(m, i, j)(n + 1)$	(1,2,3,Mapp)
5.	$G'(f)(n) \;\wedge\; C \;\wedge\; E_2 \vdash f :_p G'(p)(n)$	(Var)
6.	$G'(f)(n) \;\wedge\; C \;\wedge\; E_2 \vdash x - y :_q q = i - j$	(Var, Var, Subt)
7.	$G'(f)(n) \;\wedge\; C \;\wedge\; E_2 \vdash y :_r r = j$	(Var)
8.	$G'(f)(n) \;\wedge\; C \;\wedge\; E_2 \vdash f(x - y, y) :_m G(m, i, j)(n + 1)$	(5,6,7, Mapp)
9.	$G'(f)(n) \;\wedge\; C \;\wedge\; E_3 \vdash x :_m G(m, i, j)(n + 1)$	(Var, Conseq)
10.	$G'(f)(n) \;\wedge\; C \vdash M :_m G(m, i, j)(n + 1)$	(4,8,9, If, If)
11.	$G'(f)(n) \vdash \lambda(x, y).\, M :_u G'(u)(n + 1)$	(10, Mabs)

where $[Mapp]$ is the following extension of $[App]$ with $[Pair]$:

$$[Mapp]\ \frac{A \vdash M :_m B_1 \quad A \vdash N_i :_{x_i} B_{2i}\ (1 \leq i \leq n) \quad A \supset B_1 \supset \wedge_i B_{2i} \supset B[m \bullet (\vec{x})/u]}{A \vdash M(N_1, .., N_n) :_u B}.$$

Further, in Lines 4, 8 and 9, the following implications (all elementarily justifiable) are used.

- In Line 4, $(E_1 \wedge G'(f)(n) \wedge q = j - i \wedge r = i) \supset G(f \bullet (q, r), i, j)(n + 1)$, which holds since, under the condition, $gcd(m, i.j) = gcd(m, j - i, i)$, and because $i + j - i \lneq i + j \leq n + 1$ (which allows the use of $G'(f)(n)$ in the assumption).

- In Line 8, $(E_2 \wedge G'(f)(n) \wedge q = i - j \wedge r = j) \supset G(f \bullet (q, r), i, j)(n + 1)$, which holds for the symmetric reason.

- In Line 9, $(E_3 \wedge C[m/x]) \supset G(m, i, j)(n+1)$, which holds since, if $E_3 \wedge C[m/x]$, then $m = i = j$, hence $gcd(m, i, j)$ from which we conclude $G(m, i, j)(n + 1)$.

By combining two conclusions by $[Rec]$, we have now reached the required statement (6.15).

7 Program Logics via Processes (2): Call-by-Name and Call-by-Value

7.1 Partial Logic for $\mathbf{PCF}_v$

Following our development in Section 5 for the affine process logics, there are two other program logics for PCF_v, one for partial correctness and the other for the generalised total correctness. The partial logic allows us simpler reasoning in many cases; while the generalised total logic gives a general form of reasoning which subsumes other two. In particular, we shall show how the latter can be presented for a decomposed form of PCF_v which cleanly articulates the behaviours of PCF_v in an analytical way. After outlining these logics, we speculate how three different logics for PCF_v can be used for different purposes.

We start from the logic for partial correctness. As in the partial affine logic (cf. Section 5.1), we add the new formula for partial substitution.

$$A \quad ::= \quad ... \mid \langle\!\langle e \bullet e'/z \rangle\!\rangle A$$

The formula "$\langle\!\langle e \bullet e'/z \rangle\!\rangle A$" means either $e \bullet e'$ diverges or, if it converges, its result, named z, satisfies A. Here e and e' may soundly contain applications, even though restricting them to names does not lead to loss of expressiveness. Accordingly, we do not include the application $e \bullet e'$ in the grammar of terms (alternatively we use partial equality $\mathrel{\mathring{=}}$ on terms involving applications, where $e \mathrel{\mathring{=}} e'$ means "either one of e and e' diverges, or if both ever converge they should be equal," cf. §5.2, Remark 4, Page 53). The partial logic only uses the set of *s-admissible formulae*, which are defined following essentially the same clauses as in §5.3, Definition 7, Page 55. For clarity we reintroduce the conditions for formulae in the present system. Let $\vec{x} = x_0..x_{n-1}$ be the primary names in A. Then we say A is *s-admissible* if the following conditions hold for all of its subformulae, fixing its primary names $\vec{x}$ and assuming the standard bound name convention.

- T, F and $e_1 = e_2$ are s-admissible.
- $\langle\!\langle e \bullet e'/z \rangle\!\rangle A$ is s-admissible if A is s-admissible.
- $A_1 \wedge A_2$ and $A_1 \vee A_2$ are s-admissible when $A_{1,2}$ are s-admissible.
- $A_1 \supset A_2$ is s-admissible when $\mathsf{fn}(A_1) \cap \{\vec{x}\} = \emptyset$ and A_2 is s-admissible.
- $\forall y.A$ is s-admissible when A is s-admissible.
- $\exists y.A$ and $\neg A$ are s-admissible when no name from $\vec{x}$ occurs in A.

As we mentioned in Section 5, $\exists y.A$ is treatable even when primary names occur in A, if we know there are only finite instances of y which satisfies A. Then s-admissibility of A suffices, since in that way $\exists y.A$ is equivalent to a finite disjunction. From now on we simply say *A is s-admissible* by fixing its primary names, and only treat s-admissible formulae. From our experience so far, the s-admissibility is not overly restrictive for reasoning about partial correctness. The s-admissibility again satisfies the (semantic) admissibility condition, defined based on process-representation of PCF_v-terms or, more extensionally, using the term model of PCF_v and its standard ordering.

$$[\mathit{Abs\text{-}Partial}]\ \frac{A^{\text{-}x\vec{i}} \wedge A_1^x \vdash M :_m A_2 \quad \forall x\vec{i}.\,(A_1 \supset \langle\!\langle u \bullet x/m \rangle\!\rangle A_2\,) \supset B}{A \vdash \lambda x.M :_u B}$$

$$[\mathit{App\text{-}Partial}]\ \frac{A \vdash M :_m B_1 \quad A \vdash N :_x B_2 \quad B_1 \supset B_2 \supset \langle\!\langle m \bullet x/u \rangle\!\rangle B}{A \vdash MN :_u B}$$

$$[\mathit{Rec\text{-}Partial}]\ \frac{A \wedge B[x/u] \vdash \lambda y.M :_u B \quad A \supset \exists u.B}{A \vdash \mu x.\lambda y.M :_u B}$$

Fig. 14. Altered Proof Rules for Partial Correctness of PCF_v

The semantics of terms is precisely as we gave for the total PCF_v-logic (except we no longer have applications as terms). For the semantics of formulae, the satisfaction relation of the partial substitution is given as:

$$\xi \models^{\mathcal{I}} \langle\!\langle e \bullet e'/x \rangle\!\rangle A \quad \equiv \quad \begin{cases} [\![e]\!]_{\mathcal{I}\cdot\xi}([\![e']\!]_{\mathcal{I}\cdot\xi}) = \omega, \text{ or} \\ [\![e]\!]_{\mathcal{I}\cdot\xi}([\![e']\!]_{\mathcal{I}\cdot\xi}) = \lambda \neq \omega \text{ and } \xi \models^{\mathcal{I}\cdot x:\lambda} A. \end{cases} \tag{7.1}$$

The definition precisely mirrors the semantics of partial substitution in the process logic. The remaining clauses are precisely as in the total PCF_v-logic (which are simply the defining clauses for the satisfaction relation in the predicate calculus with equality).

Finally the semantics of judgement is given precisely as that of the total PCF_v-logic. To wit:

$$A \models M^{\vec{x}:\vec{\beta}\,;\,\alpha} :_u B \quad \equiv \quad \forall \mathcal{I}, \vec{V}^{\vec{\beta}}.\ (\ \vec{x}:\vec{V} \models^{\mathcal{I}} A \ \supset\ M[\vec{V}/\vec{x}] \models^{\mathcal{I}}_u B\ ).$$

The proof rules for the partial logic for PCF_v are given in Figure 14, listing only those rules which differ from those for total PCF_v-logic in Figure 11. The altered points are:

(1) The replacement of the textual substitutions in the rules for abstraction and applications with the partial ones; and
(2) The replacement of the recursion rule for total correctness with the one for partial correctness.

These changes precisely mirror the ones which we needed in moving from the total affine logic to the partial one in Section 5. It is notable that the proof rules preserve s-admissibility. For example, in [*Abs-Partial*] which reverses the variance of a formula, if the formula $A^{\text{-}x\vec{i}} \wedge A_1^x$ and the formula A_2 are both s-admissible with respect to respective primary names, then $\forall x\vec{i}.\,(A_1 \supset \langle\!\langle u \bullet x/m \rangle\!\rangle A_2\,)$ is again s-admissible with respect to the new primary names, because of the quantification of $x\vec{i}$. As we discussed in Section 5, the side-condition in [*Rec-Partial*], $A \supset \exists x.B$, is usually trivial to verify when B is s-admissible.

We present an example of inference for a partial correctness assertion, using the recursive factorial, cf. (6.14). First, the property of a factorial with respect to the partial correctness is given by:

$$B'(u) \stackrel{\text{def}}{=} \forall n.\ \langle\!\langle u \bullet n/z \rangle\!\rangle\ (z = n!) \tag{7.2}$$

Alternatively the same property can be written as $\forall n.\ u \bullet n \mathrel{\mathring{=}} n!$ using the partial equality. Our aim is to prove:

$$\mathsf{T} \vdash \mathit{Fact} :_u B'(u). \tag{7.3}$$

The inference follows.

1.	$B'(f) \wedge x = i \vdash x = 0 :_b i = 0 \supset b$	(Eq)
2.	$B'(f) \wedge x = i \wedge i = 0 \vdash 1 :_v v = i!$	(Num, Conseq)
3.	$B'(f) \wedge x = i \wedge i \geq 1 \vdash x :_v v = i$	(Var)
4.	$B'(f) \wedge x = i \wedge i \geq 1 \vdash f :_l B'(l)$	(Var, Conseq)
5.	$B'(f) \wedge x = i \wedge i \geq 1 \vdash x - 1 :_k k{=}i - 1$	(3, Subt)
6.	$B'(f) \wedge x = i \wedge i \geq 1 \vdash f(x-1) :_v v{=}(i-1)!$	(4, 5, App-Partial)
7.	$B'(f) \wedge x = i \wedge i \geq 1 \vdash x * f(x-1) :_v v{=}i!$	(3, 6, Mult)
8.	$B'(f) \wedge x = i \vdash \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1) :_v v{=}i!$	(2, 7, If)
9.	$\mathsf{T} \wedge B'(f) \vdash \lambda x.M :_u B'(u)$	(8, Abs-Partial)
10.	$\mathsf{T} \vdash \mu f.\lambda x.M :_u B'(u)$	(9, Rec-Partial)

In Line 10, $\mathsf{T} \supset \exists f.B'(f)$ is verified by assigning the least defined function to f, by which $B'(f)$ is trivially satisfied.

7.2 Generalised Total Logic for PCF$_v$

Generalised Total Logic with Decomposed Types. In the logic for generalised total correctness, an assertion $A \vdash M :_u B$ incorporates divergent behaviour. $\mathsf{T} \vdash M :_u \mathsf{T}$ no longer means M is terminating: it now means M can have any (well-typed) behaviour, including the divergent one. Yet if an assertion says the program converges, it ensures standard total correctness.

We first formulate this logic by faithfully following the corresponding general affine process logic via process encoding of PCF$_v$. This gives rise to an analytical, or decomposed, form of the general total logic for PCF$_v$. The type structure underlying the logic uses *lifting*, a familiar idea from the denotational semantics of partial computation [34, 90]. Let $\alpha, \beta, \ldots$ range over the original set of types for PCF$_v$. To them we add *lifted types*, written $(\alpha)^\uparrow$ for each standard PCF$_v$-type α, which operationally denote programs that may not necessarily be values. In contrast, the original set of types denote only values. We hereafter call lifted

$$[Var]\ \frac{-}{\Gamma, x:\alpha \vdash_{\mathrm{val}} x : \alpha} \qquad [Num]\ \frac{-}{\Gamma \vdash_{\mathrm{val}} \mathbf{n} : \mathbb{N}} \qquad [Op]\ \frac{\Gamma \vdash_{\mathrm{val}} V_i : \alpha_i}{\Gamma \vdash \mathsf{op}(\vec{V}) : (\beta)^{\uparrow}}$$

$$[Abs]\ \frac{\Gamma, x:\alpha \vdash M : (\beta)^{\uparrow}}{\Gamma, x:\alpha \vdash_{\mathrm{val}} \lambda x^{\alpha}.M : \alpha \Rightarrow \beta} \qquad [App]\ \frac{\Gamma \vdash_{\mathrm{val}} V : \alpha \Rightarrow \beta \quad \Gamma \vdash_{\mathrm{val}} W : \alpha}{\Gamma \vdash VW : (\beta)^{\uparrow}}$$

$$[Rec]\ \frac{\Gamma, x : \alpha \Rightarrow \beta \vdash_{\mathrm{val}} \lambda y^{\alpha}.M : \alpha \Rightarrow \beta}{\Gamma \vdash_{\mathrm{val}} \mu x^{\alpha \Rightarrow \beta}.\lambda y^{\alpha}.M : \alpha \Rightarrow \beta} \qquad [If]\ \frac{\Gamma \vdash_{\mathrm{val}} V : \mathbb{B} \quad \Gamma \vdash N_{1,2} : (\alpha)^{\uparrow}}{\Gamma \vdash \mathtt{if}\ V\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : (\alpha)^{\uparrow}}$$

$$[Let]\ \frac{\Gamma \vdash M : (\alpha)^{\uparrow} \quad \Gamma, x : \alpha \vdash N : (\beta)^{\uparrow}}{\Gamma \vdash \mathtt{let}\ x =\ M\ \mathtt{in}\ N : (\beta)^{\uparrow}} \qquad [Lift]\ \frac{\Gamma \vdash_{\mathrm{val}} V : \alpha}{\Gamma \vdash V : (\alpha)^{\uparrow}}$$

Fig. 15. Typing Rules for PCF$_v$ (decomposed)

types *partial types*, and non-lifted types *total types*. One may consider total types as replicated types and partial types as linear output types.

Due to the need to distinguish values and non-values in proof rules, we use the following decomposed presentation of the syntax of PCF$_v$.

$$\begin{aligned} M \ ::= \ & \mathbf{c} \mid x \mid \lambda x.M \mid VW \mid \mu x.\lambda y.M \mid \mathtt{if}\ V\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 \mid \mathbf{op}(\vec{V}) \mid \\ & \mathtt{let}\ x =\ M\ \mathtt{in}\ N. \end{aligned}$$

where $V, W, \ldots$ range over values, which are, as before, abstractions (including recursions), variables, and constants. In the decomposed grammar, we restrict terms in the "strict" positions (i.e. those which should be immediately evaluated) to values. This restriction is however compensated by the introduction of the `let` construct. For example, "$\mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2$" in the original syntax becomes, with x fresh, "$\mathtt{let}\ x =\ M\ \mathtt{in}\ \mathtt{if}\ x\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2$." Thus we lose no generality. The typing rules of this decomposed syntax is given in Figure 15, using the following two kinds of sequents (in both, Γ is a finite map from variables to total types).

- $\Gamma \vdash_{\mathrm{val}} V : \alpha$, which is only for values.
- $\Gamma \vdash M : (\alpha)^{\uparrow}$, which is for both values and non-values.

By folding the let construct and regarding $\Gamma \vdash M : (\alpha)^{\uparrow}$ as $\Gamma \vdash M : \alpha$, all typed terms in the original PCF$_v$ can be recovered. One may consider $\Gamma \vdash_{\mathrm{val}} V : \alpha$ is for a replicated process (via the map $\langle\!\langle V \rangle\!\rangle_u^{\mathrm{val}}$) while $\Gamma \vdash M : (\alpha)^{\uparrow}$ is for a linearly typed process (via the map $\langle\!\langle M \rangle\!\rangle_u$).

In accordance with the decomposed type structure, the logic now uses terms of the form $(e)^{\uparrow}$ in addition to those of the total PCF$_v$-logic. For clarity we present the whole grammar.

$$e \ ::= \ x^{\alpha} \ \mid \ \bot^{\alpha} \ \mid \ e_1 \bullet e_2 \ \mid \ \mathsf{op}(\vec{e}) \ \mid \ (e)^{\uparrow}$$

These terms are naturally typed with decomposed types. First, in $(e)^{\uparrow}$, e should have a total type, say α, while the expression itself has type $(\alpha)^{\uparrow}$. Similarly $e \bullet e'$ and $\mathsf{op}(e_1..e_n)$ (where e, e' and e_i have total types) have partial types. The grammar of formulae remains the same, even though there is one simplification: thanks to their totality, all terms of type $\mathbb{B}$ can always be used as formulae (cf. Convention 3, §6.6, Page 77). Given e of type $(\alpha)^{\uparrow}$, we write $e \Downarrow$ for $\exists x.(e = (x)^{\uparrow})$, and $e \Uparrow$ for its negation.

Following two kinds of typing sequents, the judgement also uses two sequents, one for values and one for general terms.

- $A \models_{\text{val}} V^{\Gamma;\alpha} :_u B$, in correspondence with $\Gamma \vdash_{\text{val}} V ; \alpha$
- $A \vdash M^{\Gamma;\alpha} :_u B$ in correspondence with $\Gamma \vdash M ; \alpha$.

The typing of primary names in these judgements follows the given base and type of a program. For the semantics of judgements, let $\lambda, \ldots$ range over *behaviours*, which are, as in §6.4, Page 68, simply the congruent classes of closed programs modulo the standard contextual equivalence. As before, for each type, there is a unique divergent behaviour, which we write $\bot$. If $\lambda \neq \bot$, then we say λ is *total.* Each total behaviour is inhabited by a value, so a total behaviour may have a total type. But a total behaviour may also have a partial type by lifting. Thus, for a total behaviour λ of type α, we consider two type assignments, written λ^{α} and $\lambda^{(\alpha)^{\uparrow}}$. For $\bot$, we only assign the lifted type, writing $\bot^{(\alpha)^{\uparrow}}$.

Let $\xi, \ldots$ range over *models*, which are a finite map from variables to behaviours. ξ is *total* if its range only consist of behaviours of total types. An *interpretation* $\mathcal{I}$ is also a map from variables to behaviours, which cam have both total and partial types. We can now define the two satisfaction relations as follows (we assume well-typedness while leaving the typing implicit).

- $\xi \models^{\mathcal{I}}_{\text{val}} A$ with ξ total, whose defining clauses precisely follow §6.4, Page 68. We also write $\lambda^{\alpha} \models^{\mathcal{I}}_{\text{val},\, u} A$ which stands for $u : \lambda^{\alpha} \models^{\mathcal{I}}_{\text{val}} A$
- $\lambda^{(\alpha)^{\uparrow}} \models^{\mathcal{I}}_{u} A$, again precisely following §6.4, assigning λ to u.

As in §6.4, we write $\vec{x} : \vec{V} \models^{\mathcal{I}}_{\text{val}} A$ for $\vec{x} : \vec{\lambda} \models^{\mathcal{I}}_{\text{val}} A$ such that $V_i \in \lambda_i$ for each i, similarly for $V \models^{\mathcal{I}}_{\text{val},\, u} A$ and $M \models^{\mathcal{I}}_{u} A$. Note the two satisfaction relations are distinct because of their typings: $\models^{\mathcal{I}}_{\text{val}}$ is about totally typed behaviours, while $\models^{\mathcal{I}}$ is about partially typed behaviours. Since the latter is in effect absent from the satisfaction for the total logic given in §6.4, it in effect adds the satisfaction relation for possibly partial behaviours, leading to the logic which include both total and partial behaviours in its universe of discourse.

Finally we set the semantics of judgement as follows, with $\Gamma = \vec{x} : \vec{\alpha}$, and again leaving the typing implicit.

- $A \models_{\text{val}} V^{\Gamma;\alpha} :_u B$ iff for each $\mathcal{I}$ and $\vec{W}$, whenever $\vec{x} : \vec{W} \models^{\mathcal{I}} A$ we have $V[\vec{w}/\vec{x}] \models^{\mathcal{I}}_{\text{val},\, u} B$.
- $A \models M^{\Gamma;\alpha} :_u B$ iff for each $\mathcal{I}$ and $\vec{W}$, whenever $\vec{x} : \vec{W} \models^{\mathcal{I}} A$ we have $M[\vec{w}/\vec{x}] \models^{\mathcal{I}}_{u} B$.

$$[Op]\ \frac{A \vdash_{\mathrm{val}} V_i :_{m_i} B_i \ \ (1 \le i \le n) \quad (\wedge_i B_i) \supset B[\mathsf{op}(m_1,..,m_n)/u]}{A \vdash \mathsf{op}(V_1,..,V_n) :_u B}$$

$$[If]\ \frac{A \vdash_{\mathrm{val}} V :_b A' \quad A \wedge A'[\mathsf{T}/b] \vdash N_1 :_u B \quad A \wedge A'[\mathsf{F}/b] \vdash N_2 :_u B}{A \vdash \mathtt{if}\ V\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 :_u B}$$

$$[Abs]\ \frac{A^{\text{-}x\vec{i}} \wedge A_1^x \vdash M :_m A_2 \quad \forall x\vec{i}.\,(A_1 \supset A_2[u \bullet x/m]) \supset B}{A \vdash_{\mathrm{val}} \lambda x.M :_u B}$$

$$[App]\ \frac{A \vdash_{\mathrm{val}} V :_m B_1 \quad A \vdash_{\mathrm{val}} W :_x B_2 \quad B_1 \supset B_2 \supset B[m \bullet x/u]}{A \vdash VW :_u B}$$

$$[Rec]\ \frac{A \vdash_{\mathrm{val}} \lambda y.M :_u B(0) \qquad A \wedge B(i)[x/u] \vdash_{\mathrm{val}} \lambda y.M :_u B(i+1)}{A \vdash_{\mathrm{val}} \mu x.\lambda y.M :_u \forall i.B(i)}$$

$$[Rec\text{-}\Uparrow]\ \frac{A^{\text{-}x} \wedge B[x/u] \vdash_{\mathrm{val}} \lambda y.M :_u B \quad B \text{ admissible, } A \supset \exists x.B}{A \vdash_{\mathrm{val}} \mu x.\lambda y.M :_u B}$$

$$[Let]\ \frac{A \vdash M :_m A' \quad A \wedge A'[(x)^{\uparrow}/m] \vdash N :_u B \quad B \Downarrow_u \supset A' \Downarrow_m}{A \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N :_u B}$$

$$[Lift]\ \frac{A \vdash_{\mathrm{val}} V :_x B[(x)^{\uparrow}/u]}{A \vdash V :_u B}$$

Fig. 16. Proof Rules for General Total Logic of PCF_v (decomposed version)

Figure 16 lists the proof rules for the generalised total logic, which precisely follow the typing rules in Figure 15 (we omit the rules for variables and constants, since they are identical with those in the total PCF_v-logic, cf. Figure 11, §6.5, Page 71). Some illustration of the proof rules follows.

- $[Op]$ and $[If]$ are as before, except for the use of value sequents. In $[If]$, the condition part us a value (accordingly b has the total type $\mathbb{B}$, allowing its use as a formula), so that the property B for $N_{1,2}$ named as u, which may say u converges, can be directly inherited in the conclusion.
- $[Abs]$ introduces a value, while $[App]$ introduces a non-value. In $[Abs]$, note x and u have total types, which makes $u \bullet x$ well-defined, while m has a partial type, which conforms to the substitution of $u \bullet x$ for m. Similarly for $[App]$. Note how types of names and the kinds of sequents used precisely conform.
- In $[Let]$, "$A \Downarrow_x$" denotes, as in Section 5.7, "$A \supset x \Downarrow$". Thus the side condition says: *if we wish the whole term to terminate, we need to assume M also terminates.* With the translation of $\mathtt{let}\ x = M\ \mathtt{in}\ N$ as $(\boldsymbol{\nu}\, m)(\langle\!\langle M \rangle\!\rangle_m | m(x).\langle\!\langle N \rangle\!\rangle)$, the side condition precisely corresponds to (Bra) in Figure 9, Page 60. $[Lift]$ turns an assertion on a value to an assertion about a general program.

These proof rules are (not only based on but also) justifiable by the general affine logic (cf. Figure 9, §5.7, Page 60), via logical full abstraction.

$$[Op]\ \frac{A \vdash M_i :_{m_i} B_i \ \ (1 \le i \le n) \quad (\wedge_i B_i) \supset B[\mathsf{op}(m_1, .., m_n)/u] \quad B \Downarrow_u \ \supset \ \wedge_i B_i \Downarrow_{m_i}}{A \vdash \mathsf{op}(V_1, .., V_n) :_u B}$$

$$[\mathit{If}]\ \frac{A \vdash M :_b A' \quad A \wedge A'[\mathsf{T}/b] \vdash N_1 :_u B \quad A \wedge A'[\mathsf{F}/b] \vdash N_2 :_u B \quad B \Downarrow_u \ \supset \ A' \Downarrow_b}{A \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 :_u B}$$

$$[Abs]\ \frac{A^{\text{-}x\vec{i}} \wedge A_1^x \vdash M :_m A_2 \quad u \Downarrow \supset \forall x\vec{i}.\, (A_1 \supset A_2[u \bullet x/m]\,) \ \supset \ B}{A \vdash \lambda x.M :_u B}$$

$$[App]\ \frac{A \vdash M :_m B_1 \quad A \vdash N :_x B_2 \quad A \supset B_1 \supset B_2 \supset B[m \bullet x/u] \quad B \Downarrow_u \supset (B_1 \Downarrow_m \wedge B_2 \Downarrow_x)}{A \vdash M N :_u B}$$

Fig. 17. Proof Rules for General Total Logic for PCF_v (single type version)

Generalised Total Logic without Decomposition. The rules in Figure 16 follow a clean type structure of the decomposed PCF_v. From the viewpoint that proofs for programs may as well faithfully follow the syntax of programs as they are programmed (except for structural rules), these rules may not be fully satisfactory. To obtain the proof rules for generalised total correctness which are based on PCF_v rather than on its decomposed form, a natural idea (which we do take in the following) is to coalesce lifted types and non-lifted types, implicitly converting between these two types. The proof rules are given in Figure 17 (under Convention 1, §3.5, Page 31, as always). Note the reasoning precisely does follow the syntax of PCF_v, and that we only need one sequent: a drawback is that we should mention the conditions for convergence at each rule involving evaluation of terms. In $[Abs]$, we also explicitly note that abstraction is total. While each rule becomes more complex, the added side condition in each rule should read naturally, motivated from operational semantics of term forms. It is also notable that this presentation directly subsumes all provable formulae in the total logic discussed in Section 6, where the latter is the result of inductively assuming termination.

It is easy to justify the proof rules in Figure 17 via the encoding into the previous decomposed rules. We also believe, up to the translation, the derivability of two systems coincides on the original PCF_v-programs.

7.3 On the Use of Different Affine Program Logics

Of the three different program logics for PCF_v which we have seen so far, we believe the logic for total correctness is best suited when the program logic is used during the design/programming stage from the bottom up, especially when the resulting software is assumed to run in a well-defined environment. In this case, ensuring the correctness including termination for each piece of software is natural, which is precisely what the total logic helps us to do.

On the other hand, the partial correctness gives a simpler platform for reasoning about programs, as well as offering an alternative basic criterion for program correctness. As has been practised traditionally, one may as well prove the properties of programs first using proof rules for partial correctness, which would then be augmented with the proof of termination, the latter can be done either by the logical method as presented here or by a different means. As another significant usage of the partial correctness logic, it would be necessary to use this logic when one cannot assume programs used as the environment of her/his own software will surely terminate.

Finally the logic for generalised total correctness serves the same purpose as the logic for partial correctness. Further, since it can stipulate arbitrary properties for both the environment and the program, it can be used for specification of programs in general environments with different degrees of reliability. Another significant use of the generalised total logic is, as we discussed in §4.3, for the verification of correctness of compositional program analyses. In many such cases, an analysis may not ensure termination, but still guarantees significant properties of interest. In such situations, the generalised total logic would particularly be useful.

7.4 Call-by-Name PCF

When a program calls a procedure, there are two basic ways to treat evaluation of actual parameters (i.e. expressions actually passed at the time of a call). In *call-by-value*, we evaluate the actual parameters *before* calling, so that evaluated expressions, or values, are passed to the callee. In *call-by-name*, we pass actual parameters as they are without evaluating them, substituting for formal parameters literally (usually avoiding capture of names): then they will be evaluated when it becomes necessary to do so. The distinction arises conspicuously when passing compound functions as an actual parameter, where call-by-value imposes more sequencing than call-by-name. For example, when we call p with an expression $f(g(h(0)))$, adopting call-by-value means we first evaluate $h(0)$, then $g(v)$ where v is the result of evaluating $h(0)$, then finally $f(w)$ where w is the result of evaluating $g(v)$, while call-by-name does not impose such sequencing, which can lead to a greater room for optimisation when all these functions are side-effect free. Sequencing is however useful when these functions do have side effects, since in that case sequencing gives important information. Another place where their difference arises is when evaluation of actual parameters leads to nontermination. For example, $(\lambda x.1)\Omega$ diverges in call-by-value, but converges to 1 in call-by-name.

These two modes of calling procedures, or in fact of calling any named entity with a parameter passing mechanism, can be given two distinct, and equally canonical, models both in theories of functions (where call-by-name is modelled as total continuous functions between pointed CPOs and call-by-value as partial continuous functions between bottomless CPOs, or alternatively total strict continuous functions between pointed CPOs [32]) and in theories of processes (where the difference between call-by-name and call-by-value arises as two distinct forms of flow of interaction [58, 62, 80]). Originally it is in Algol-60 that the call-by-name is adopted as a parameter passing mechanism, concerning which

Peter Landin [70] pointed out how an applied form of the call-by-name λ-calculus can model many of its features. Since then, extensive studies have been carried out on the semantics of programming languages focussing on call-by-name calculi. More recently, several modern functional programming languages are based on call-by-name evaluation, including Haskell[42].

In the following we shall briefly discuss the logics for call-by-name PCF, derived from the underlying process logics via standard process encoding of call-by-name sequential functions. Our main interest is how the difference between call-by-name and call-by-value may be reflected onto the program logics. Do process logics give us program logics which cleanly reflect the intrinsic nature of call-by-name computation, distinct from their call-by-value counterpart? We shall see this is indeed so, both for total correctness and for partial one.

The syntax of call-by-name PCF, which we shall hereafter simply call PCF_n, is the same as that of PCF_v, except that the recursion now has the form $\mu x.M$ for an arbitrary M. The dynamics is given by the standard (weak) call-by-name reduction [32], again written $\longrightarrow$. The call-by-name evaluation means that, among others, when we reduce MN, we do reduce M but not N. We use the same grammar for types and typing judgements. The typing rules only change for $\mu x.M$, which now becomes:

$$[Rec]\ \frac{\Gamma, x : \alpha \vdash M : \alpha}{\Gamma \vdash \mu x^{\alpha}.M : \alpha}. \tag{7.4}$$

Thus we have a diverging term $\mu x^{\alpha}.x$, often written Ω^{α}, at each type α. For semantics, we take the view that the only observables are termination at atomic types, based on which we obtain the contextual congruence $\cong_{\mathrm{cbn}}$. While we can equally treat termination at each type, this is *the* semantics of PCF_n which has been studied traditionally, has a good practical footing, and makes the distinction from the call-by-value most conspicuous.

7.5 Program Logics for Call-by-Name PCF (1): Total Correctness

We first present a logic for total correctness for PCF_n, then discuss how they can be understood via the total affine logic. We use exactly the same syntax for our logical language as in the total logic for PCF_v except for limiting the divergent constants to atomic types. Below we list the syntax in full.

$$e \ ::= \ x^{\alpha} \ | \ \bot^{\mathbb{B}} \ | \ \bot^{\mathbb{N}} \ | \ \mathbf{n} \ | \ e_1 \bullet e_2 \ | \ e_1 + e_2 \ | \ e_1 \times e_2 \ | \ \ldots \tag{7.5}$$

We may note, while we use the same notation in two logics, the meaning of the application operator, $e_1 \bullet e_2$, in the call-by-name logic is substantially different from the one in the call-by-value logic. For example, $\exists x^{\mathbb{N}\Rightarrow\mathbb{N}}.\ \forall y^{\mathbb{N}}.\ x \bullet y = 1$ is false in the call-by-value logic but is true in the present logic.

The divergence at higher-order types is not explicitly present since it can now be defined as followed. Below we assume e has type $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_{n-1} \Rightarrow \beta$ $(n \geq 0)$ where β is atomic, and $\vec{y} = y_1..y_n$ with each y_i fresh and distinct. $e \bullet \vec{y}$ means the obvious n-fold applications (which becomes e itself when $n = 0$).

$$e \Uparrow^{\mathsf{cbn}} \qquad \equiv \qquad \forall \vec{y}.\, e \bullet \vec{y} = \bot^{\beta}. \tag{7.6}$$

Thus $e \Uparrow^{\mathsf{cbn}}$ means e is the least defined element of that type, either in base types or in arrow types. We also write $e \Downarrow^{\mathsf{cbn}}$ for $\neg(e \Uparrow^{\mathsf{cbn}})$.

Finally, in spite of having these divergent elements, the logic now treats all expressions including divergent ones as (total) values. In particular, a variable of type $\mathbb{N}$ can denote $\perp$ even if it is a primary name, since $\perp$ is a total element. Some observations:

1. From an operational viewpoint, the framework reflects the demand-driven nature of call-by-name, in the sense that, in call-by-name, we are interested in the content of expressions, including its convergence, only when it is used. Thus even $\Omega^{\mathbb{N}}$ has a non-trivial valid specification — since it starts diverging only when they are called.
2. From the viewpoint of traditional denotational semantics, this corresponds to the standard interpretation of types as pointed CPOs: atomic types are augmented with least elements to become pointed, while the least element in a function type can be characterised extensionally.

The semantics of terms is given just as we did in the previous section (an alternative, and equivalent, semantics can be given using pointed CPOs and total continuous functions, just as we did Appendix B.3 for the call-by-value logic). We first take the $\cong_{\mathrm{cbn}}$-congruent classes of PCF_n-terms, which we again call *behaviours* ($\lambda, \ldots$, or with type annotation, $\lambda^{\alpha}, \ldots$). $\lambda \bullet \lambda'$ is defined by reflecting the application MN in PCF_n onto the congruent classes. A *model* ($\xi, \ldots$ or, with type annotation, $\xi^{\Gamma}, \ldots$) is a finite map from variables to behaviours. *We have no distinction between total models and non-total ones.* The satisfaction relation $\xi \models^{\mathcal{I}} A$ is defined as in the standard predicate logic with equality, from which we define $\vec{x} : \vec{M} \models^{\mathcal{I}} A$ (note the terms in the environment are not values) and $M \models^{\mathcal{I}}_{u} A$.

We use two forms of judgements for the total correctness of PCF_n.

$$A \models M^{\Gamma,\alpha} :_u B, \qquad A \models_{\mathrm{run}} M^{\Gamma,\alpha} :_u B.$$

where, in the second sequent, we restrict α to a base type. The validity of the first assertion is given in the now familiar manner:

$$A \models M^{\Gamma,\alpha} :_u B \quad \equiv \quad \forall \mathcal{I}. \, (\vec{x} : \vec{N} \models^{\mathcal{I}} A \supset M \models^{\mathcal{I}}_{u} B \,)$$

while that of the latter is given as:

$$A \models_{\mathrm{run}} M^{\Gamma,\alpha} :_u B \quad \equiv \quad \forall \mathcal{I}. \, (\vec{x} : \vec{N} \models^{\mathcal{I}} A \; \supset \; M \models^{\mathcal{I}}_{u} B \wedge B \Downarrow^{\mathsf{cbn}})$$

where $B \Downarrow^{\mathsf{cbn}}$ stands for $B \supset u \Downarrow^{\mathsf{cbn}}$. The judgement $A \models_{\mathrm{run}} M^{\Gamma,\alpha} :_u B$ compensates the treatment of $\perp$ as a total value since the validity of $A \models_{\mathrm{run}} M :_u B$ guarantees that M does not diverge with respect to the call-by-name reduction. But the natural sequent for the use in the compositional reasonoing is $A \models M^{\Gamma,\alpha} :_u B$ (as we shall see in the next section), using which we would reach the desired conclusion which, for a program of a base type, may as well take the shape $A \models_{\mathrm{run}} M^{\Gamma,\alpha} :_u B$.

$$[Op]\ \frac{A \vdash M_i :_{m_i} B_i \ \ (1 \le i \le n) \quad (\wedge_i B_i) \supset B[\mathsf{op}(m_1,..,m_n)/u] \quad B \Downarrow_u^{\mathsf{cbn}} \supset \wedge_i \mathcal{B}_i \Downarrow_{m_i}^{\mathsf{cbn}}}{A \vdash \mathsf{op}(M_1,..,M_n) :_u B}$$

$$[If]\ \frac{A \vdash M :_b A' \quad A \wedge A'[\mathsf{T}/b] \vdash N_1 :_u B \quad A \wedge A'[\mathsf{F}/b] \vdash N_2 :_u B \quad B \Downarrow_u^{\mathsf{cbn}} \supset A' \Downarrow_b^{\mathsf{cbn}}}{A \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 :_u B}$$

$$[Abs]\ \frac{A^{-x\vec{i}} \wedge A_1^x \vdash M :_m A_2 \quad \forall x\vec{i}.\,(A_1 \supset A_2[u \bullet x/m]) \supset B}{A \vdash \lambda x.M :_u B}$$

$$[App]\ \frac{A \vdash M :_m B_1 \quad A \vdash N :_x B_2 \quad A \supset B_1 \supset B_2 \supset B[m \bullet x/u]}{A \vdash MN :_u B}$$

$$[Rec]\ \frac{A \vdash M :_u B(0) \qquad A \wedge B(i)[x/u] \vdash M :_u B(i+1)}{A \vdash \mu x.M :_u \forall i.B(i)}$$

$$[Run]\ \frac{A \vdash M^{\Gamma;\alpha} :_u B \quad B \Downarrow_u^{\mathsf{cbn}} \quad \alpha \in \{\mathbb{B}, \mathbb{N}\}}{A \vdash_{\mathrm{run}} M :_u B}$$

Fig. 18. Updated/Added Rules for Call-by-Name PCF: total correctness

7.6 Proof Rules of Total PCF$_n$-Logic

The proof rules for the PCF$_n$-logic are given in Figure 18 (we omit the rules for variable/constant as well as the consequence rule, since they are identical with those in the total PCF$_v$-logic, given in Figure 11, §6.5, Page 71), always under Convention 1, §3.5, Page 31. Practically, one may consider the aim of the proof system as being the derivation of $A \vdash_{\mathrm{run}} M :_u B$ if M has an atomic type; and $A \vdash M :_u B$ if M has a higher-order type. Some observations:

- [*Op*] demands that, if any of the atomic values used as arguments diverges, then the whole term should either diverge if it has a base type, or is the least defined function if it has a higher-order type, under the assumption that the arithmetic/boolean operations are strict at each argument.
- [*If*] is understood as [*Op*].
- In [*App*], we need no requirement that the result of application converges, since any resulting element, higher-order or atomic, is legitimately total.
- [*Rec*] simply follows the call-by-name typing of recursion (7.4). Note, thus, we can derive $\mathsf{T} \vdash \Omega^{\mathbb{N}} :_u \mathsf{T}$, which is indeed valid.
- [*Run*] "runs" the term of an atomic type and only allows to assert when it does not diverge.

For establishing the soundness of proof rules, one can again use the encoding into the underlying process logic, on which we discuss next.

7.7 Process Encoding of Total PCF$_n$-Logic

We use the encoding due to Hyland and Ong [62], which is closely related with that of Milner [80]. We first observe any PCF$_n$-type can be written $\alpha \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ ($n \geq 1$) where α_n is a base type. We write this as $[\alpha_1, \alpha_2, .., \alpha_n]$, following [62]. Types are mapped as:

$$\begin{aligned}
[\alpha_1, \ldots, \alpha_{n-1}, \beta]^\circ &\stackrel{\text{def}}{=} (\overline{\alpha_1^\circ}, \ldots, \overline{\alpha_{n-1}^\circ}, \beta^\bullet)^! \\
\mathbb{N}^\bullet &\stackrel{\text{def}}{=} [\oplus_{i \in \mathbb{N}}]^\uparrow \\
\mathbb{B}^\bullet &\stackrel{\text{def}}{=} [\oplus_{i \in \mathbb{B}}]^\uparrow.
\end{aligned}$$

(The mappings α° and $\alpha^\bullet$ are exclusively used with the above meaning in this subsection). Thus $[\alpha_1, \alpha_2, .., \alpha_{n-1}, \beta]$ abstracts the behaviour which asks at each α_i and returns the answer at β. Note also a base type is encoded with an affine (not linear) type, unlike in PCF$_v$, in correspondence with the formation of recursion at base types. Following the encoding of types, the translation of terms is given as follows. Below let $\alpha = [\beta_1, \beta_2, .., \beta_{n-1}, \gamma]$, $\vec{y} = y_1..y_{n-1}$ where each y_i is fresh and distinct, and all newly introduced names are fresh.

$$\begin{aligned}
[\![M : \alpha]\!]_u &\stackrel{\text{def}}{=} !u(\vec{y}z).[\![M : \alpha]\!]^{\text{run}}_{\vec{y}z} \\
[\![x : \alpha]\!]^{\text{run}}_{\vec{y}z} &\stackrel{\text{def}}{=} \overline{x}\langle\vec{y}z\rangle^{\overline{\alpha_1^\circ}..\overline{\alpha_{n-1}^\circ}\gamma^\bullet} \\
[\![n : \mathbb{N}]\!]^{\text{run}}_{z} &\stackrel{\text{def}}{=} \overline{z}\mathtt{in}_n \\
[\![\lambda x^\alpha.M : \theta \Rightarrow \alpha]\!]^{\text{run}}_{x\vec{y}z} &\stackrel{\text{def}}{=} [\![M : \alpha]\!]^{\text{run}}_{\vec{y}z} \\
[\![MN : \alpha]\!]^{\text{run}}_{\vec{y}z} &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\, x)([\![M : \theta \Rightarrow \alpha]\!]^{\text{run}}_{x\vec{y}z} | [\![N : \theta]\!]_x) \\
[\![\mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : \alpha]\!]^{\text{run}}_{\vec{y}z} &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\, c)([\![M : \mathbb{B}]\!]^{\text{run}}_c | c[\&_{i \in \mathbb{B}}[\![N_i : \alpha]\!]^{\text{run}}_{\vec{y}z}]) \\
[\![\mu x.M]\!]^{\text{run}}_{\vec{y}z} &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\, xu)([\![M : \alpha]\!]^{\text{run}}_u \,|\, [x \to u] \,|\, \overline{u}\langle\vec{y}z\rangle)
\end{aligned}$$

In the Hyland-Ong encoding, all arguments are passed in one go, so that, at the final argument, we can obtain the affine observable (at an atomic type). This idea is reflected in the encoding of terms of our logical language, which is identical with the one presented in Section 6.4 except for (1) adding the divergent constant $\perp$; and (2) altering the map of application, both listed below (we assume well-typedness as always).

$$\perp_u^\circ \quad \stackrel{\text{def}}{=} \quad u \bullet \varepsilon = \omega \tag{7.7}$$

$$(e^{\alpha \Rightarrow \beta} \bullet e'^{\alpha})_u^\circ \quad \stackrel{\text{def}}{=} \quad \exists xy.\, (\forall \vec{z}.\, u \bullet \vec{z} = x \bullet y\vec{z} \wedge e_x^\circ \wedge e'^\circ_y) \tag{7.8}$$

In (7.8), the semantics of an application is given by supplying all remaining arguments to reach the atomic type, which is necessary from the typing (this is reminiscent of logical relations: in the standard presentation they are constructed using binary applications but the resulting meaning is extensional). The encoding of formulae is given precisely as for the total PCF$_v$-logic. Assertions are then mapped as follows.

$$A \vdash M^{\Gamma;\alpha} :_u B \quad \mapsto \quad [\![M]\!]_u^{\overline{\Gamma^\circ};u:\alpha^\circ} \vdash \textbf{rely}\, A^\circ\, \textbf{guar}\, B^\circ$$
$$A \vdash_{\text{run}} M^{\Gamma;\alpha} :_u B \quad \mapsto \quad [\![M]\!]_u^{\text{run}}\ {}^{\overline{\Gamma^\circ};u:\alpha^\bullet} \vdash \textbf{rely}\, A^\circ\, \textbf{guar}\, \exists x.(u = x \bullet \varepsilon \wedge B^\circ[x/u])$$

At the level of validity, this mapping easily gives the logical full abstraction (cf. Theorem 5, §6.4), The encoding of a formula in the total PCF$_n$-logic is meaningful both in the total process logic and in the general affine process logic: however it is more natural to employ the latter as a target. This is because, when we consider provability. the embedding is only meaningful when we consider the general total process logic, mapping the derivation of each rule in the total PCF$_n$-logic into several steps of inferences in the generalised affine process logic. The mapping uses the "run" encoding at each step of rule induction, identifying $[\![M]\!]^{\text{run}}_{\vec{y}z}$ with $(\boldsymbol{\nu}\, m)([\![M]\!]_m|\overline{m}\langle\vec{y}z\rangle)$. As an example of reasoning, $[Abs]$ is sound by the following inference, letting $\vec{i}$ to be fresh and $B' \stackrel{\text{def}}{=} \exists m.(B_2^\circ \wedge m \bullet \vec{i} = z)$:

1.	$[\![M]\!]_m \vdash \textbf{rely}\, A^\circ \wedge B_1^\circ\, \textbf{guar}\, B_2^\circ$	(assumption)
2.	$\overline{m}\langle\vec{y}z\rangle \vdash \textbf{rely}\, A^\circ \wedge B_1^\circ \wedge B_2^\circ \wedge \vec{y}=\vec{i}\, \textbf{guar}\, B'$	(from (3.14) of §3.7)
3.	$[\![M]\!]_m|\overline{m}\langle\vec{y}z\rangle \vdash \textbf{rely}\, A^\circ \wedge B_1^\circ \wedge B_2^\circ \wedge \vec{y}=\vec{i}\, \textbf{guar}\, B'$	(1, 2, (Par))
4.	$(\boldsymbol{\nu}\, m)([\![M]\!]_m|\overline{m}\langle\vec{y}z\rangle) \vdash \textbf{rely}\, A^\circ \wedge B_1^\circ \wedge \vec{y}=\vec{i}\, \textbf{guar}\, B'$	(Res)
5.	$!u(x\vec{y}z).[\![M]\!]^{\text{run}}_{\vec{y}z} \vdash \textbf{rely}\, A^\circ\, \textbf{guar}\, \forall x\vec{y}\vec{i}.((B_1^\circ \wedge \vec{y}=\vec{i}) \supset B'[u \bullet x\vec{y}/z]$	(In)
6.	$!u(x\vec{y}z).[\![M]\!]^{\text{run}}_{\vec{y}z} \vdash \textbf{rely}\, A^\circ\, \textbf{guar}\, \forall x.(B_1^\circ \supset B_2^\circ[u \bullet x/m])$	(Conseq)

In Line 6, we infer:

$$\begin{array}{rcl} \forall x\vec{y}\vec{i}.((B_1^\circ \wedge \vec{y}=\vec{i}) \supset B'[u \bullet x\vec{y}/z]) & \equiv & \forall x.(B_1^\circ \supset \forall \vec{y}.(\exists m.(B_2^\circ \wedge m \bullet \vec{y} = u \bullet x\vec{y}))) \\ & \supset & \forall x.(B_1^\circ \supset \exists m.(B_2^\circ \wedge \forall \vec{y}.(m \bullet \vec{y} = u \bullet x\vec{y}))) \\ & \supset & \forall x.(B_1^\circ \supset (B_2[u \bullet x/m])^\circ) \end{array}$$

where the exchange of $\forall \vec{y}$ and $\exists m$ in the second line is possible because $m \bullet \vec{y} = u \bullet x\vec{y}$ completely determines m by extensionality and because no name from $\vec{y}$ occurs in B_2. Note Line 3 may not be meaningful in the total affine logic, but is so in the general affine logic. Just as Theorem 6, the logical full abstraction and soundness of the embedding lead to the soundness of the proof system with respect to validity.

7.8 Program Logics for Call-by-Name PCF (2): Partial Correctness

In this subsection we discuss a program logic for partial correctness for PCF$_n$. One of the interesting aspects of this logic is that we do not have to use partial substitution, unlike in the partial logic for PCF$_v$. Instead, we use two kinds of expressions, one for numerical/boolean expressions denoting total values (which never contain applications and over which we use the standard equality) and another for expressions denoting programs' behaviour (which may as well contain applications and for which we use the partial equality). While it is more precise to formally distinguish these two kinds of expressions, the present discussion treats them together using the same symbols $e, e', \ldots$, which may not lead to ambiguity by the use of different symbols for equality. Thus we use the syntax:

$$\begin{array}{lcl} e & ::= & x^\alpha \mid \mathbf{n} \mid \bot^\alpha \mid e_1 + e_2 \mid e_1 \times e_2 \mid \ldots \\ A & ::= & e_1 = e_2 \mid e_1 \mathrel{\mathring{=}} e_2 \mid \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \supset A_2 \mid \forall x.A \mid \exists x.A \end{array}$$

In practice we may as well take off $\bot^\alpha$ (which is for the behaviour, so that we always have $\bot = e$ for each e of the same type). It is apt to illustrate the semantics of $\mathring{=}$ (for other terms and formulae, the treatment is standard except for the admissibility condition on formulae discussed later).

- When $e_{1,2}$ are of an atomic type, the equality $e_1 \mathrel{\mathring{=}} e_2$ holds when (1) e_1 and e_2 both denote the same atomic value, or (2) e_1 or e_2 diverges.
- Equations on expressions of higher-order types is then given inductively. For example, $x^{\mathbb{N} \Rightarrow \mathbb{N}} \mathrel{\mathring{=}} y^{\mathbb{N} \Rightarrow \mathbb{N}}$ is true when $\forall z^{\mathbb{N}}.x \bullet z \mathrel{\mathring{=}} y \bullet z$ is true.

The translation into the partial process logic elucidates the ideas underlying these definitions. Let $\alpha = [\beta_1, .., \beta_{n-1}, \mathbb{N}]$. Then $x^\alpha \mathrel{\mathring{=}} y^\alpha$ in the partial PCF$_n$-logic is essentially the same thing as $\forall \vec{z}.x \bullet \vec{z} \mathrel{\mathring{=}} y \bullet \vec{z}$ in the partial process logic, which is a short hand for $\forall \vec{z}.\langle\!\langle x \bullet \vec{z}/w \rangle\!\rangle (w = y \bullet \vec{z})$ with w fresh. As we mentioned in Section 5.2, $\mathring{=}$ does not enjoy transitivity (though it does enjoy reflexivity and symmetry): however it is partially associative w.r.t. $=$, that is we have the following implication:

$$(e \mathrel{\mathring{=}} e'' \;\wedge\; e'' = e') \quad \supset \quad e \mathrel{\mathring{=}} e'. \tag{7.9}$$

which becomes important in practice. Another basic axiom schema is, for any e of type τ:

$$\exists x^\tau.(x \mathrel{\mathring{=}} e) \tag{7.10}$$

The validity is immediate by assigning the least behaviour to x.

Following the development in Section 5 (in particular §5.3 and §5.7), we further demand s-admissibility of formulae, which, in the present context, is given as follows. Let, as before, fix the primary names $\vec{x}$ in A and assume the bound name convention.

- T, F, $e = e'$ and $e \mathrel{\mathring{=}} e'$ are s-admissible.
- $A_1 \wedge A_2$ and $A_1 \vee A_2$ are s-admissible when $A_{1,2}$ are s-admissible.
- $A_1 \supset A_2$ is s-admissible when $\mathsf{fn}(A_1) \cap \{\vec{x}\} = \emptyset$ and A_2 is s-admissible.
- $\forall y.A$ is s-admissible when A is s-admissible.
- $\exists y.A$ and $\neg A$ are s-admissible when no name from $\vec{x}$ occurs in A.

Again we can relax the condition for $\exists y.A$ when we know there are only finite instances of y satisfying A. The conditions as a whole say that equations by $\mathring{=}$ which concern primary names never occur in a negative position.

As an example of s-admissible formulae, the following formula (giving the specification of a factorial in the present setting, corresponding to (6.14) in §6.6 and (7.2) in §7.1), is s-admissible, assuming u is the only primary name.

$$B''(n) \quad \stackrel{\text{def}}{=} \quad \forall n.\ u \bullet n \doteq n\mathbf{!}. \tag{7.11}$$

The assertion does not say the result of application converges: but it does say, if it ever converges, it coincides with the factorial of an argument. We can also write such formulae as:

$$i \gtrsim 0 \supset \exists j.(u \bullet i \doteq j \gtrsim 0). \tag{7.12}$$

Above $i \gtrsim 0$ stands for $\exists h.(i = 1 + h)$. The formula above combines both the standard equality (for non-behaviours) and the partial equality (for u standing for the behaviour).

Before presenting the proof rules, it is illustrative to briefly outline the formal encoding of the call-by-name partial logic in process logics. As a target logic, we use the general affine process logic rather than the partial affine logic (note the former subsumes the latter). Expressions are encoded thus [4]:

$$\begin{aligned}
\mathtt{n}^\circ_u &\stackrel{\text{def}}{=} u \bullet \varepsilon = \mathtt{in}_n()^\uparrow \\
x^\circ_u &\stackrel{\text{def}}{=} u = x \\
(e \bullet e')^\circ_u &\stackrel{\text{def}}{=} \exists xy.\ \forall \vec{z}.(u \bullet \vec{z} = x \bullet y\vec{z} \wedge e^\circ_x \wedge e'^\circ_y).
\end{aligned}$$

The addition of extra arguments $\vec{z}$ in the third line above is necessitated by the encoding of types, which sends all arguments until the base type is reached. For formulae, the partial equality is interpreted as:

$$(e \doteq e')^\circ \stackrel{\text{def}}{=} \exists xy.\ (\forall \vec{z}.(x \bullet \vec{z}\Uparrow \ \vee\ y \bullet \vec{z}\Uparrow \ \vee x \bullet \vec{z} = y \bullet \vec{z}) \ \wedge\ e^\circ_x \ \wedge\ e'^\circ_y)$$

The encoded formula says that $e \doteq e'$ means, when fed with all arguments to reach a base type, either e diverges, e' diverges, or they are equal (since we are in the general affine logic, the last equality also includes the case when both diverge: while this is redundant, it is harmless). At this point, the reason for using the general affine logic as a target logic may become clear. The call-by-name partial logic captures both partial correctness (for behaviours) and total correctness (for expressions): moreover, for the latter, the partiality is captured at the level of equations, not at the level of the whole formula. Thus its exact representation needs to treat both kinds of assertions, as can be done by the general affine logic but not by the partial affine logic.

We omit the semantics of terms and formulae, which precisely follows the encoding just given, except for noting that the resulting logic does not allow a diverging term ($\mu x^{\mathbb{N}}.x$, among others) to have an arbitrary specification, even

[4] While the symbol $(\cdot)^\circ$ is already used in Section 6, it denotes the above call-by-name mapping in the rest of the present section.

$$[\mathit{Var}]\ \frac{-}{A[x/u] \vdash x :_u A} \qquad [\mathit{Num}]\ \frac{-}{A[\mathbf{n}/u] \vdash \mathbf{n} :_u A} \qquad [\mathit{Boolean}]\ \frac{-}{A[\mathbf{b}/u] \vdash \mathbf{b} :_u A}$$

$$[\mathit{Op}]\ \frac{A \vdash M_i :_{m_i} B_i \quad (1 \le i \le n) \quad (\wedge_i B_i) \supset B[\mathsf{op}(m_1, .., m_n)/u]}{A \vdash \mathsf{op}(M_1, .., M_n) :_u B}$$

$$[\mathit{If}]\ \frac{A \vdash M :_b A' \quad A \wedge A'[\mathsf{T}/b] \vdash N_1 :_u B \quad A \wedge A'[\mathsf{F}/b] \vdash N_2 :_u B}{A \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 :_u B}$$

$$[\mathit{Abs}]\ \frac{A^{\text{-}x\vec{i}} \wedge A_1^x \vdash M :_m A_2 \quad \forall x\vec{i}.\,(A_1 \supset A_2[u \bullet x/m]) \supset B}{A \vdash \lambda x.M :_u B}$$

$$[\mathit{App}]\ \frac{A \vdash M :_m B_1 \quad A \vdash N :_x B_2 \quad B_1 \supset B_2 \supset B[m \bullet x/u]}{A \vdash MN :_u B}$$

$$[\mathit{Rec}]\ \frac{A \wedge B[x/u] \vdash M :_u B \quad A \supset \exists u.B}{A \vdash \mu x.M :_u B}$$

$$[\mathit{Consequence}]\ \frac{A \supset A_0 \quad A_0 \vdash M :_u B_0 \quad B_0 \supset B}{A \vdash M :_u B}$$

Fig. 19. Proof Rules for Call-by-Name PCF: partial correctness

though it can be equated with an arbitrary behaviour of the same type. This indicates the basic difference from the standard partial correctness, but is consistent with the process-logical reading of formulae just discussed.

The translation of formulae we have discussed suggests simple proof rules for the partial PCF_n-logic, listed in Figure 19 (we list the rules in full). Some observations:

1. [*Rec*] indicates the inference system is indeed for partial correctness. We can check we can*not* derive $\mathsf{T} \vdash \mu x.x :_u \mathsf{F}$, even though we *can* derive:
$$\mathsf{T} \vdash \mu x.x :_u \forall i^{\mathbb{N}}.u \mathrel{\mathring{=}} i. \tag{7.13}$$
For deriving (7.13), let $A(m) \stackrel{\text{def}}{=} \forall i^{\mathbb{N}}.m \mathrel{\mathring{=}} i$. Then $\mathsf{T} \wedge A(x) \vdash x :_m A(m)$ and the entailment $\mathsf{T} \supset \exists x.A(x)$ (which is an instance of the axiom (7.10)) give us (7.13) via [Rec].
2. In no places do we need to check convergence of applications, nor do we have to use the partial substitution for abstraction and application. The latter is partly because we do not allow, as we noted above, the derivation of $\mathsf{T} \vdash \mu x.x :_u \mathsf{F}$.
3. In both m of [*Abs*] and u of [*App*], the occurring substitutions indicate they are treated as behaviours, so that they cannot occur in either side of the standard equality.

Other rules should now read familiar.

We do not claim that these rules (and the underlying logic) are the most adequate rules for reasoning about the partial correctness of PCF_n. However, as far as the logic given above is accepted as one of the effective ways to assert on the call-by-name partial correctness, these proof rules may as well be regarded as being natural and intuitive. The verification of the soundness of these rules is proved via their process encoding into those in the general affine process logic we just discussed.

Let us present a very simple example of reasoning, which illustrates how the lack of associativity of $\mathrel{\mathring{=}}$ (which surely poses a problem in its use for reasoning) may be circumvented for a practical reasoning.

1.	$B''(f) \wedge x \mathrel{\mathring{=}} i \vdash x = 0 :_b i = 0 \supset b$	(Var, Eq, Conseq)
2.	$B''(f) \wedge x \mathrel{\mathring{=}} i \wedge i = 0 \vdash 1 :_v v \mathrel{\mathring{=}} i!$	(Num, Conseq)
3.	$B''(f) \wedge x \mathrel{\mathring{=}} i \wedge i \geq 1 \vdash x :_v v \mathrel{\mathring{=}} i$	(Var)
4.	$B''(f) \wedge x \mathrel{\mathring{=}} i \wedge i \geq 1 \vdash f :_l B''(l)$	(Var, Conseq)
5.	$B''(f) \wedge x \mathrel{\mathring{=}} i \wedge i \geq 1 \vdash x - 1 :_k k \mathrel{\mathring{=}} i - 1$	(3, Subt)
6.	$B''(f) \wedge x \mathrel{\mathring{=}} i \wedge i \geq 1 \vdash f(x-1) :_v v \mathrel{\mathring{=}} (i-1)!$	(4, 5, App-Partial)
7.	$B''(f) \wedge x \mathrel{\mathring{=}} i \wedge i \geq 1 \vdash x * f(x-1) :_v v \mathrel{\mathring{=}} i!$	(3, 6, Mult)
8.	$B''(f) \wedge x \mathrel{\mathring{=}} i \vdash \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1) :_v v \mathrel{\mathring{=}} i!$	(2, 7, If)
9.	$\mathsf{T} \wedge B''(f) \vdash \lambda x.M :_u B''(u)$	(8, Abs-Partial)
10.	$\mathsf{T} \vdash \mu f.\lambda x.M :_u B''(u)$	(9, Rec-Partial)

Note the inference precisely corresponds to the one given in §7.1 except for the use of partial equality, mixed with the standard equality, which are mediated by the axiom (7.9).

8 Polymorphism and Recursive Types

Recursive types and polymorphism are fundamental building blocks of many modern programming languages [20, 96]. They also play an important role when analysing programs' behaviour, including those of low-level languages and object-oriented ones [20, 87, 96]. In this section we demonstrate how the framework we have developed so far can smoothly cover these refined type structures without any change in its fundamental principles. As applications we show how the resulting process logic guides the development of powerful reasoning principles for the extension of PCF_v with polymorphism and recursive types.

In the rest of the section, we first introduce, in §8.1, the syntax and typing rules for the second-order polymorphism, then presents a logical language for polymorphic behaviour in §8.2. Semantics and proof rules come next (§8.3). The incorporation of recursive types is discussed in §8.4. The resulting process logic is used for deriving a program logic for PCF_v extended with polymorphism and recursive types. The syntax and typing rules of the extended calculus are introduced in §8.5. The syntax and semantics of formulae and the proof rules are presented in §8.6. Finally §8.8 illustrates the use of proof rules with a few non-trivial examples. For simplicity we focus on the total correctness throughout the section: precisely the same extensions are possible for other two affine logics.

8.1 Process Logic for Polymorphism (1): Typing

As in functional calculi (cf. [38, 103]), the polymorphism is incorporated into the π-calculus by adding type variables and their quantifiers into its type system. One difference from polymorphism in functions is that, by reflecting the duality in interaction, two kinds of quantifiers, existential and universal, simultaneously arise: only having one does not make a coherent type structure. Including this point, the following construction appeared in [14], incorporating the ideas from Turner's polymorphic π-calculus [111] into the duality-based types.

$$\tau \quad ::= \quad \ldots \mid \mathrm{X} \mid \overline{\mathrm{X}} \mid \forall \vec{\mathrm{X}}.(\vec{\tau}^{?}\rho^{\uparrow})^{!} \mid \forall \vec{\mathrm{X}}.(\vec{\tau}^{?})^{\downarrow} \mid \exists \vec{\mathrm{X}}.(\vec{\tau}^{!}\rho^{\downarrow})^{?} \mid \exists \vec{\mathrm{X}}.(\vec{\tau}^{!})^{\uparrow}$$

Both X and $\overline{\mathrm{X}}$ are type variables. Their restriction to positive modes (for X) and negative modes (for $\overline{\mathrm{X}}$) are for convenience. Note we only allow guarded forms of quantifications: again this is not necessary, but these restricted forms suit the operational content of polymorphism, which we discuss later. The mode of $\forall \mathrm{X}.\tau$ and $\exists \mathrm{X}.\tau$ is that of τ. Binders, α-conversions etc. are defined in the standard way, except that a quantifier binding X also binds $\overline{\mathrm{X}}$. Similarly, a substitution $[\rho/\mathrm{X}]$ simultaneously substitutes $\overline{\rho}$ for $\overline{\mathrm{X}}$. When $\vec{\mathrm{X}}$ is the null string, we identify $\forall \vec{\mathrm{X}}.\tau$ and $\exists \vec{\mathrm{X}}.\tau$ with τ.

The dynamics associated with these types can be given by introducing a free output into our syntax [14]. For example, a free output $\overline{x}\langle v\rangle$ can be given a type assignment

$$\vdash \overline{x}\langle v\rangle \triangleright x : (\mathrm{X})^{\uparrow},\ v : \overline{\mathrm{X}},$$

which says x is carrying a name v which should be used generically (i.e. only as a name carried by a message: note if it is used as a subject, which the process should do if it uses a bound output at x, it cannot have a type X). If we prefix this process as $!u(vx).\overline{x}\langle v\rangle$, then we obtain a process which behaves as a generic identity, receiving one name and returning it without ever using it, that is the process is generic at the first channel. In fact this process can be given the following typing (in the implicit typing):

$$\vdash !u(vx).\overline{x}\langle v\rangle \triangleright u : \forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}.$$

In this way free outputs and polymorphism are closely tied with each other.

Instead of directly using free outputs, it is more convenient, for the purpose of the present study, to represent the equivalent of free outputs while staying within bound outputs. This can be done using a special kind of copycats, the constants which are called *generic wires*. To understand the underlying idea, we first recall (cf. §3.7, Page 37) that any free output in the first-order type discipline can be equivalently represented using a copy-cat, so that, say, $\overline{x}\langle v\rangle$ can be represented as $\overline{x}(c)[c \to v]^{\tau}$ when v has a type $\overline{\tau}$. Thus all we need to have to represent a free output carrying a generic name is to have a copycat at type X. Operationally, if this generic copycat is to serve the purpose of a copycat at type X then it should also serve the purpose of any (replicated) type, via type instantiation. The generic copycat, which we hereafter call *generic wire* or simply *wire*, is written $[x \triangleright y]$ (for a linear type) and $![x \triangleright y]$ (for a replicated type). The significance of using a generic wire instead of free outputs lies in the clean logical analysis it gives us. Thus we add the following additional syntax to the grammar, which includes generic wires as well as, for having the explicitly typed syntax, refined prefixed terms with type abstractions and concretion.

$$\begin{array}{rcl} P & ::= & ... \mid x(\vec{\mathrm{X}})[\&_{i\in I}(\vec{y}_i).P_i] \mid \overline{x}[\vec{\tau}]\mathtt{in}_i(\vec{y})P \mid !x(\vec{\mathrm{X}})(\vec{y}).P \mid \overline{x}[\vec{\tau}](\vec{y})P \mid \\ & & [x \triangleright y] \mid ![x \triangleright y] \end{array}$$

$(\vec{\mathrm{X}})$ in $x(\vec{\mathrm{X}})[\&_{i\in I}(\vec{y}_i).P_i]$ and $!x(\vec{\mathrm{X}})(\vec{y}).P$ act as binders. While we can continue to work with implicit typing, this partial type elaboration elucidates the operational content of the type structure. We often use a free output notation, so that $!u(vx).\overline{x}(c)![c \triangleright v]$ is usually written $!u(vx).\overline{x}\langle v\rangle$. The dynamics of these prefixed terms now involve not only name-passing but also type-passing [111].

$$x(\vec{\mathrm{X}})[(\vec{y}_i).P_i] | \overline{x}[\vec{\rho}]\mathtt{in}_i(\vec{y}_i)Q \longrightarrow (\boldsymbol{\nu}\,\vec{y}_i)(P_i[[\vec{\rho}/\vec{\mathrm{X}}]/|]Q) \qquad (8.1)$$

$$!x(\vec{\mathrm{X}})(\vec{y}).P | \overline{x}[\vec{\rho}](\vec{y})Q \longrightarrow !x(\vec{\mathrm{X}})(\vec{y}).P | (\boldsymbol{\nu}\,\vec{y})(P[[\vec{\rho}/\vec{\mathrm{X}}]/|]Q) \qquad (8.2)$$

Wires simply forward an output from one name to another name.

$$\begin{array}{rcl} [x \triangleright y] \mid \overline{x}[\vec{\tau}](\vec{z})P & \longrightarrow & \overline{y}[\vec{\tau}](\vec{z})P \\ ![x \triangleright y] \mid \overline{x}[\vec{\tau}](\vec{z})P & \longrightarrow & ![x \triangleright y] \mid \overline{y}[\vec{\tau}](\vec{z})P \end{array}$$

For example, the following reduction including a free output:

$$\overline{x}\langle v\rangle | x(c).\overline{c}(u)P | !v(u)Q \longrightarrow \overline{v}(u)P | !v(u)Q \longrightarrow (\boldsymbol{\nu}\,u)(P|Q),$$

$$\frac{(\mathsf{Bra}^{\downarrow})\quad (\{\vec{\mathrm{X}}\} \cap (\mathsf{ftv}(\Gamma) \cup \mathsf{ftv}(\rho)) = \emptyset) \qquad \forall i.\ \vdash P_i \triangleright ?\Gamma^{-x},\ v:\rho^{\uparrow},\ \vec{y}_i:\vec{\tau}_i}{\vdash x(\vec{\mathrm{X}})[\&_i(\vec{y}_i).P_i] \triangleright \Gamma,\ v:\rho,\ x:\forall\vec{\mathrm{X}}.[\&_i\vec{\tau}_i]^{\downarrow}}$$

$$\frac{(\mathsf{Sel}^{\uparrow}) \qquad \vdash P \triangleright ?\Gamma,\ \vec{y}:\vec{\tau}_i[\rho/\mathrm{X}]}{\vdash \overline{x}[\vec{\rho}]\mathtt{in}_i(\vec{y})P \triangleright \Gamma,\ x:\exists\vec{\mathrm{X}}.[\oplus_i\vec{\tau}_i]^{\uparrow}}$$

$$\frac{(\mathsf{In}^{!})\quad (\{\vec{\mathrm{X}}\} \cap \mathsf{ftv}(\Gamma) = \emptyset) \qquad \vdash P \triangleright ?\Gamma^{-x},\ \vec{y}:\vec{\tau},\ z:\rho}{\vdash !x(\vec{\mathrm{X}})(\vec{y}z).P \triangleright \Gamma,\ x:\forall\vec{\mathrm{X}}.(\vec{\tau}\rho)^{!}}$$

$$\frac{(\mathsf{Out}^{?})\quad (\Gamma(x) = \exists\vec{\mathrm{X}}.(\vec{\tau}\rho)^{?}) \qquad \vdash R \triangleright ?\Gamma,\ \vec{y}:\vec{\tau}[\vec{\gamma}/\vec{\mathrm{X}}] \qquad \vdash P \triangleright ?\Gamma,\ v:\sigma,\ z:\rho[\vec{\gamma}/\vec{\mathrm{X}}]}{\vdash \overline{x}[\vec{\gamma}](\vec{y}z)(R|P) \triangleright \Gamma,\ v:\sigma}$$

$$\frac{(\mathsf{Wire}\text{-}\downarrow) \qquad -}{\vdash [x \triangleright y] \triangleright x:\tau^{\downarrow}, y:\overline{\tau}^{\uparrow}} \qquad \frac{(\mathsf{Wire}\text{-}!) \qquad -}{\vdash ![x \triangleright y] \triangleright x:\tau^{!}, y:\overline{\tau}^{?}}$$

Fig. 20. Typing Rules for Polymorphism (with partial type annotation)

is simulated by its generic wire representation as:

$$\begin{aligned}&\overline{x}(c)![c \triangleright v]|x(c).\overline{c}(u)P|!v(u)Q \ \longrightarrow\ (\boldsymbol{\nu}\, c)(![c \triangleright v]|\overline{c}(u)P)|!v(u)Q \ \longrightarrow \\ &(\boldsymbol{\nu}\, c)![c \triangleright v] \mid (\overline{v}(u)P|!v(u)Q) \ \longrightarrow\ (\boldsymbol{\nu}\, c)![c \triangleright v] \mid (\boldsymbol{\nu}\, u)(P|Q),\end{aligned}$$

where two results are identical up to a semantically negligible $(\boldsymbol{\nu}\, c)![c \triangleright v]$.

In typing, we refine the prefix rules and add one rule for a wire, as given in Figure 20. The introduction of quantification is done at the time of prefixing, following [111]. When universal quantification is introduced, the standard condition on type variables should be obeyed. These rules become the original prefix rules for the linear/affine π-calculus, given in Figure 1, §3.1, Page 25, when the abstracted type variables become null.

As a simple example, we can type the universal identity, which is now explicitly typed as $!u(\mathrm{X})(vx).\overline{x}(c)![c \triangleright v]$ which we still write $\mathbf{id}(u)$, as follows:

$$\cfrac{(\mathsf{Sel})\ \cfrac{(\mathsf{Wire}\text{-}!)\ \cfrac{-}{\vdash ![c \triangleright v] \triangleright c:\mathrm{X}, v:\overline{\mathrm{X}}}}{\vdash \overline{x}(c)![c \triangleright v] \triangleright x:(\mathrm{X})^{\uparrow}, v:\overline{\mathrm{X}}}}{(\mathsf{In}\text{-}\forall)\ \vdash !u(\mathrm{X})(vx).\overline{x}(c)![c \triangleright v] \triangleright u:\forall\mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}}$$

thus obtaining:

$$\vdash \mathbf{id}(u) \triangleright u:\forall\mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}. \tag{8.3}$$

Similarly we can type the following agent dual to $\mathbf{id}(u)$, with $\mathtt{true}_v \stackrel{\text{def}}{=} !v(c).\overline{c}\mathtt{in}_\mathtt{t}$ and $\mathtt{not}_{cw} \stackrel{\text{def}}{=} \overline{c}(e)e[.\overline{w}(g)\mathtt{false}_g \& .\overline{w}(g)\mathtt{true}_g]$:

$$\vdash \overline{u}[\mathbb{B}](vx)(\mathtt{true}_v | x(c).\mathtt{not}_{cw}) \triangleright u : \exists \mathrm{X}.(\mathrm{X}(\overline{\mathrm{X}})^{\downarrow})^{?}, w : (\mathbb{B})^{\uparrow} \tag{8.4}$$

The following ADT uses an existential type for linear output, exporting, via its arguments x, i, q: (1) an opaque integer, (2) an increment operation, and (3) the instantiation of an opaque value.

$$\vdash \overline{u}[\mathbb{N}](xig)([\![0]\!]_x \,|!i(ye).\mathtt{succ}_{ye} |!\mathbf{id}(g)) \triangleright u : \exists \mathrm{X}.(\mathrm{X}(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}(\overline{\mathrm{X}}(\mathbb{N})^{\uparrow})^{!})^{\uparrow} \tag{8.5}$$

where $\mathtt{succ}_{ye} \stackrel{\text{def}}{=} \overline{y}(c)c[\&_n \overline{e}(n')[\![n+1]\!]_{n'}$. Its dual process is typed as:

$$\vdash u(\mathrm{X})(xig)(\mathtt{inc}_{ixgr}) \triangleright u : \forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X}(\overline{\mathrm{X}})^{\downarrow})^{?}(\mathrm{X}(\overline{\mathbb{N}})^{\downarrow})^{?})^{\downarrow}, r : (\mathbb{N})^{\uparrow} \tag{8.6}$$

where $\mathtt{inc}_{ixgr} \stackrel{\text{def}}{=} \overline{i}(x'c)(![x' \triangleright x] | c(y).g(y'r')(![y' \triangleright y] | r'(c).\overline{r}\langle c \rangle))$, which increments an opaque value once and outputs the instantiated value via r.

8.2 Process Logic for Polymorphism (2): Terms and Formulae

The logical language refine that of the total affine logic by allowing the representation of communication (or concretion) of types, and, at the level of formulae, add quantification of types.

$$a \quad ::= \quad \ldots \quad | \quad x \bullet [\vec{\tau}]\vec{y} \quad | \quad [\vec{\tau}]\mathtt{in}^{\rho}_{e}(\vec{y})^{\uparrow}$$

The added constructs should obey the following typing constraints.

- In $x \bullet [\vec{\tau}]\vec{y}$, the name x should be typed as $\forall \vec{\mathrm{X}}.(\vec{\rho}\sigma)^{!}$, the types $[\vec{\tau}]$ (called *type concretion*) should match $\vec{\mathrm{X}}$ in length, and $\vec{y}$ should be (pointwise) typed by $\vec{\rho}[\vec{\tau}/\vec{\mathrm{X}}]$. Then the term as a whole has the type $\sigma[\vec{\tau}/\vec{\mathrm{X}}]$.

- In $[\vec{\tau}]\mathtt{in}^{\rho}_{e}(\vec{y})^{\uparrow}$, the type ρ should have shape $\exists \vec{\mathrm{X}}.[\oplus_i \vec{\sigma}_i]^{\uparrow}$ where $\vec{\tau}$ and $\vec{\mathrm{X}}$ have the same length, and $\vec{y}$ should have a type $\vec{\sigma}_j[\vec{\tau}/\vec{\mathrm{X}}]$ for each valuation j of e. The whole term has the type ρ. $[\vec{\tau}]$ is again called type concretion.

As before (cf. §3.2), regarding $[\vec{\tau}]\mathtt{in}^{\rho}_{e}(\vec{y})^{\uparrow}$, we always assume e is given so that the term is well-typed for its arbitrary valuation, and usually omit ρ. The newly introduced term forms mimic the type passing in reduction, (8.1) and (8.2) in . Page 99. The formulae add type quantifiers:

$$A \quad ::= \quad \ldots \quad | \quad \forall \mathrm{X}.\, A \quad | \quad \exists \mathrm{X}.\, A$$

$\forall \mathrm{X}.\, A$ means for any type α, $A[\mathrm{X}/\alpha]$ is true; while $\exists \mathrm{X}.\, A$ means for some type α, $A[\mathrm{X}/\alpha]$ is true.

As a simple example of formulae with a universal quantification, we give a specification for $\mathbf{id}(u)$, corresponding to the typing in (8.3):

$$\forall \mathrm{X},\, y^{\mathrm{X}}.\ (\ u^{\forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}} \bullet [\mathrm{X}]y = (y)^{\uparrow}\ ), \tag{8.7}$$

which says that, at u, an invocation with any argument of any type returns the argument itself, precisely describing the behaviour of $\mathbf{id}(u)$. Note this formula has a typing $u : \forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$, the same as given in (8.3). By the law of universal quantification, this formula may as well have its instantiation:

$$\forall y^{\mathbb{N}}.\ (\ u^{\forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}} \bullet [\mathbb{N}]y = (y)^{\uparrow}\ ), \tag{8.8}$$

which says whatever natural number is used as an argument of invocation of u, the process always returns that number itself. Note the typing of the formula (8.8) is $u : \forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$, which is the same as (8.7)

The type concretion, as presented in the examples above, underline the significant role types play in specifying polymorphic behaviour. In practice, however, omitting concretion is harmless since they can always be inferred from the typed names. Thus we may as well simplify (8.7) as:

$$\forall \mathrm{X}, y^{\mathrm{X}}.\ (\ u^{\forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}} \bullet y = (y)^{\uparrow}\ ) \tag{8.9}$$

Similarly for (8.8). Further, since we can usually infer the type of a primary name of a given assertion since it comes from a typed process concerned, we may further simplify (8.9) to $\forall \mathrm{X}, y^{\mathrm{X}}.\ u \bullet y = (y)^{\uparrow}$. This simpler presentation would still be clear enough as a specification for $\mathbf{id}(u)$.

Next we present a usage of type concretion together with existential quantification on types, using two specifications for the integer ADT in (8.5). The two specifications we shall present in the following have the same typing, but are quite different, one specifying details of the behaviour on opaque types; while another specifying the behaviour of a process more abstractly within the confines of the abstraction visible to the outside. In the following we let $\tau \stackrel{\mathrm{def}}{=} (\mathrm{X}(\overline{\mathrm{X}}(\mathbb{N})^{\uparrow})^{!}(\overline{\mathrm{X}}(\mathbb{N})^{\uparrow})^{!})^{\uparrow}$ for brevity. First we present a concrete specification.

$$\exists x^{\rho_1}, i^{\rho_2}, g^{\rho_3}.\,(\ u^{\exists \mathrm{X}.\tau} = [\mathbb{N}]^{\exists \mathrm{X}.\tau}(xig)^{\uparrow}\ \wedge\ A_1(x)\ \wedge\ A_2(i)\ \wedge\ A_3(g)\ ) \tag{8.10}$$

where $\rho_1 \stackrel{\mathrm{def}}{=} \mathbb{N}$, $\rho_2 \stackrel{\mathrm{def}}{=} (\overline{\mathbb{N}}(\mathbb{N})^{\uparrow})^{!}$ and $\rho_3 \stackrel{\mathrm{def}}{=} (\overline{\mathbb{N}}(\mathbb{N})^{\uparrow})^{!})^{\uparrow}$ (all of which can indeed be inferred from the typing constraint on type concretion) and $A_{1,2,3}$ are given by, writing "$\underline{u \text{ is } n}$" for "$u \bullet \varepsilon = \mathtt{in}_n()^{\uparrow}$" (cf. §6.4):

- $A_1(x) \stackrel{\mathrm{def}}{=} \underline{x \text{ is } 0}$.
- $A_2(i) \stackrel{\mathrm{def}}{=} \forall yn.\,(\ \underline{y \text{ is } n} \supset \exists y'.\,(i \bullet y = (y')^{\uparrow} \wedge \underline{y' \text{ is } n+1}\,)\,)$.
- $A_3(g) \stackrel{\mathrm{def}}{=} \forall yn.\,(\ \underline{y \text{ is } n} \supset g \bullet \varepsilon = (y)^{\uparrow})$.

Note the specifications A_1, A_2 and A_3 match the given types of x, i and g. In this specification, the name u is typed with the existential type $\exists \mathrm{X}.\tau$. However, except for this point, the specification hides nothing. While it is surely a sound, and in fact most detailed, description of the behaviour of the integer ADT in (8.5), the specification does not treat the opaque datum and operations as such. In other words, we can have the same specification even if $\exists \mathrm{X}.\tau$ is replaced by a concrete type $\tau[\mathbb{N}/\mathrm{X}]$, merely by taking off the type concretion. However the formula (8.10) does have the typing $u : \exists \mathrm{X}.\tau$, which indicates this detailed behaviour is not visible to the outside.

We now move to a more abstract specification, which is gotten by, in brief, existentially abstracting the occurrences of $\mathbb{N}$ which are opaque from the outside, in the specification (8.10). Below we use the same τ as given above and let $\sigma_1 \stackrel{\text{def}}{=} \mathrm{X}$, $\sigma_2 \stackrel{\text{def}}{=} (\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ and $\sigma_3 \stackrel{\text{def}}{=} (\overline{\mathrm{X}}(\mathbb{N})^{\uparrow})^{!})^{\uparrow}$.

$$\exists \mathrm{X}, x^{\sigma_1}, i^{\sigma_2}, g^{\sigma_3}.\, (\, u^{\exists \mathrm{X}.\tau} = [\mathrm{X}]^{\exists \mathrm{X}.\tau}(xig)^{\uparrow} \;\wedge \\ \exists w^{\mathrm{X}} c^{\mathrm{X}}.\, (\, i \bullet x = (w)^{\uparrow} \wedge g \bullet w = (c)^{\uparrow} \wedge \underline{c \text{ is } 1}\ )\ ). \quad (8.11)$$

As reflected in types of x, i and g, the specification no longer reveals neither the initial value nor the increment operation: the only thing it reveals is the final value, which does have the concrete type. The specification (8.11) has the typing $u : \tau$ which is the same as (8.10), but the expressed property is much more abstract. As we did in the universal abstraction, we may again omit type concretions from formulae, which can usually be inferable from types given to free/bound names in formulae (in fact we can even eliminate most of the type annotations for free/bound names in this and many other instances by inferring them from a given specification using the standard technique of type inference).

8.3 Process Logic for Polymorphism (3): Semantics and Proof Rules

In the present discussion we only consider the total correctness. The semantics of terms again use abstract processes, which are, as before, processes (each with a single name) up to $\cong_\pi$ and the injective renaming. Abstract processes are ranged over by $p, q, \ldots$. This gives us the notion of models, for which we again only consider total ones, i.e. those models which do not include the immediately diverging process. The interpretation of terms and the satisfaction relation are as in §4.4,P Page 44, except we add, with τ ranging over closed replicated types:

- $\xi \models^{\mathcal{I}} \forall \mathrm{X}.A \;\equiv\; \forall \tau.\, \xi \models^{\mathcal{I}\cdot x \mapsto \tau} .\, A.$
- $\xi \models^{\mathcal{I}} \exists \mathrm{X}.A \;\equiv\; \exists \tau.\, \xi \models^{\mathcal{I}\cdot x \mapsto \tau} .\, A.$

The fundamental properties of this model include:

1. The operator $\bullet$ defines a function, because of the context lemma for polymorphic affine processes [15].
2. We can use all basic syntactic properties of the equational logic as before, as well as $\forall \mathrm{X}.A \supset A[\rho/\mathrm{X}]$ assuming the action type does not change.

Following the corresponding clauses for the total affine logic, we arrive at the semantics of assertions, whose validity is written $P^{\Gamma} \models \textbf{rely}\, A\, \textbf{guar}\, B$ as before.

The proof system for the extended logic uses the identical rules as those of the total logic except for conservatively refining the four prefix rules and adding one new rule for the wire, using the same sequent $P^{\Gamma} \vdash \textbf{rely}\, A\, \textbf{guar}\, B$ as before. The altered/added rules are given in Figure 21. We observe:

(Bra$^{\downarrow}$)
$$\frac{\forall i.\ P_i \vdash \mathbf{rely}\ A[\mathtt{in}_i[\vec{\mathrm{X}}](\vec{y}_i)^{\uparrow}/x]\ \ \mathbf{guar}\ B^{\text{-}\mathrm{X}}}{x(\vec{\mathrm{X}})[\&_i(\vec{y}_i).P_i] \vdash \mathbf{rely}\ \exists\vec{\mathrm{X}}.A\,\mathbf{guar}\ B}$$

(Sel$^{\uparrow}$)
$$\frac{P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B[\mathtt{in}_i[\vec{\mathrm{X}}](\vec{y}_i)^{\uparrow}/x][\rho/\vec{\mathrm{X}}]}{\overline{x}[\vec{\rho}]\mathtt{in}_i(\vec{y})P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ \exists\vec{\mathrm{X}}.B}$$

(In$^{!}$) $(\forall\vec{\mathrm{X}}.\forall\vec{y}.\,(B_1 \supset B_2[x\bullet[\vec{\mathrm{X}}]\vec{y}/z]) \supset B)$
$$\frac{P \vdash \mathbf{rely}\ A^{\text{-}\vec{\mathrm{X}}\vec{y}} \wedge B_1^{\vec{y}}\ \ \mathbf{guar}\ B_2}{!x(\vec{\mathrm{X}})(\vec{y}z).P \vdash \mathbf{rely}\ A\,\mathbf{guar}\,B}$$

(Out$^{?}$) $(A \supset A_1 \supset A_2[x\bullet[\vec{\rho}]\vec{y}/z] \wedge x\bullet[\vec{\rho}]\vec{y}\Downarrow)$
$$\frac{P \vdash \mathbf{rely}\ \ A^{\text{-}z} \wedge A_2^{z}\ \ \mathbf{guar}\ \ A_1^{\vec{y}} \wedge B^{\text{-}\vec{y}}}{\overline{x}[\vec{\rho}](\vec{y}z)P \vdash \mathbf{rely}\ A\ \ \mathbf{guar}\ B}$$

(Wire-$\downarrow$)
$$\frac{-}{[x \triangleright y] \vdash \mathbf{rely}\ A[y/x]\,\mathbf{guar}\ A}$$

(Wire-$\mathbf{!}$)
$$\frac{-}{![x \triangleright y] \vdash \mathbf{rely}\ A[y/x]\,\mathbf{guar}\ A}$$

Fig. 21. Extended/Added Rules for Polymorphism (for total affine logic)

- Each rule closely mirrors the corresponding typing rules, respecting the polarities of introduced types. In particular, the prefix rules simply augment the original first-order rules with necessary type concretion/abstraction.
- In each of (Bra$^{\downarrow}$) and (Out$^{?}$), the quantifier introduced in the assertion is dual to the one introduced for the term, because the former is present in the rely formula.
- The rule (Wire) is the precise mirror of the property of copycat, saying whatever being assumed for the environment is copied as the specification of the process.

In practice, we may as well eliminate type concretions (of the form $x \bullet [\vec{\mathrm{X}}]$) from assertions, hence from the four prefix rules, by which we obtain the proof rules for simplified formulae without type concretions.

We list two examples of inferences using these proof rules, one for the universal identity and another for its dual agent (appeared in (8.4)). First we show the simple derivation for the specification of the universal identity.

1.	$![c \triangleright v] \vdash \mathbf{rely}\ v = i\,\mathbf{guar}\ c = i$	(Wire)
2.	$\overline{x}(c)![c \triangleright v] \vdash \mathbf{rely}\ v = i\,\mathbf{guar}\ x = (i)^{\uparrow}$	(Sel)
3.	$!u(\mathrm{X})(vx).\overline{x}(c)![c \triangleright v] \vdash \mathbf{rely}\ \mathsf{T}\,\mathbf{guar}\,\forall\mathrm{X}.\,\forall y^{\mathrm{X}}.\,(u \bullet [\mathrm{X}]y = y)$	(In, Conseq)

Thus we obtain:

$$\mathbf{id}(u) \vdash \mathbf{rely}\ \mathsf{T}\,\mathbf{guar}\,\forall\mathrm{X}.\,\forall y^{\mathrm{X}}.\,(u \bullet [\mathrm{X}]y = y) \tag{8.12}$$

Since $\forall\mathrm{X}.\forall y^{\mathrm{X}}.\,(u \bullet [\mathrm{X}]y = y)$ entails $\forall y^{\mathbb{N}}.\,(u \bullet [\mathbb{N}]y = y)$ (via the semantics of $\forall\mathrm{X}$ noted above), we also obtain this more concrete specification for $\mathbf{id}(u)$.

Next we consider the agent given in (8.4), which has a behaviour dual to $\mathbf{id}(u)$, and infer a property of its composition with $\mathbf{id}(u)$. For reference we reproduce this agent.

$$\vdash \overline{u}[\mathbb{B}](vx)(\mathtt{true}_v | x(c).\mathtt{not}_{cw}) \triangleright u : \exists \mathrm{X}.(\mathrm{X}(\overline{\mathrm{X}})^{\downarrow})^{?}, w : (\mathbb{B})^{\uparrow}. \qquad (8.13)$$

where we set:

- $\mathtt{true}_x \stackrel{\text{def}}{=} !x(c).\overline{c}\mathtt{in}_{\mathbf{t}}$ and
- $\mathtt{not}_{cw} \stackrel{\text{def}}{=} \overline{c}(e)e[.\overline{w}(g)\mathtt{false}_g \& .\overline{c}(g)\mathtt{true}_g]$,

respectively representing the truth value and the negation operation.

The process (8.13) invokes u with an opaque truth value as its argument, receives the result and negates it (note this is possible because the process knows, internally, that the returned value is a boolean, though it does not know — at least literally — which value it would receive). For brevity of the inference we use the following lemmas, which are easily derivable, Below $\underline{x \text{ is } \mathbf{t}}$ means $x \bullet \varepsilon = \mathtt{in}_{\mathbf{t}}()^{\uparrow}$ as before, while $\underline{x \text{ gives } \mathbf{f}}$ is defined as $\exists b.(w = (b)^{\uparrow} \wedge \underline{b \text{ is } \mathbf{f}})$.

$$\mathtt{true}_v \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \underline{v \text{ is } \mathbf{t}} \qquad (8.14)$$

$$\mathtt{not}_{cw} \vdash \mathbf{rely}\ \underline{c \text{ is } \mathbf{t}}\ \mathbf{guar}\ \underline{w \text{ gives } \mathbf{f}} \qquad (8.15)$$

The inference follows. Below in Line (4), "(Out-$\exists$)" indicates the higher-order quantification is non-trivially used; if not, we simply write "(Out)", etc. $A(u)$ stands for the formula $\forall \mathrm{X}.\forall y^{\mathrm{X}}.\,(u \bullet [\mathrm{X}]y = y)$.

1.	$\mathtt{true}_x \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \underline{x \text{ is } \mathbf{t}}$	((8.14) above)
2.	$\mathtt{not}_{cw} \vdash \mathbf{rely}\ \underline{c \text{ is } \mathbf{t}}\ \mathbf{guar}\ \underline{w \text{ gives } \mathbf{f}}$	((8.15) above)
3.	$v(c).\mathtt{not}_{cw} \vdash \mathbf{rely}\ \underline{v \text{ gives } \mathbf{t}}\ \mathbf{guar}\ \underline{w \text{ gives } \mathbf{f}}$	(Bra-fo)
4.	$\overline{u}[\mathbb{B}](xv)(\mathtt{true}_x \mid v(c).\mathtt{not}_{cw}) \vdash \mathbf{rely}\ A(u)\ \mathbf{guar}\ \underline{w \text{ gives } \mathbf{f}}$	(Out-$\exists$)
5.	$\mathbf{id}(u) \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ A(u)$	((8.12) above)
6.	$\overline{u}[\mathbb{B}](xv)(\mathtt{true}_x \mid v(c).\mathtt{not}_{cw}) \mid \mathbf{id}(u) \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \underline{w \text{ gives } \mathbf{f}}$	(Par)

In Line 4, the following implications are used:

$$\begin{aligned}(\ \forall \mathrm{X}.\forall y^{\mathrm{X}}.\,(u \bullet [\mathrm{X}]y = y)\ \wedge\ \underline{x \text{ is } \mathbf{t}}\ )\ &\supset\ (\ \forall y^{\mathbb{B}}.\,(u \bullet [\mathbb{B}]y = y)\ \wedge\ \underline{x \text{ is } \mathbf{t}}\ )\\ &\supset\ u \bullet [\mathbb{B}]x = x.\end{aligned}$$

Note these three formulae connected by the entailment in the above reasoning all own the identical action type $u : \forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ (if not the entailment does not hold by the lack of well-typedness). More complex examples of inferences would appear when we treat the corresponding logic for PCF_v extended with polymorphism and recursive types.

(T-Rec)

$$\frac{\vdash P \triangleright \Gamma \quad \Gamma \approx \Delta}{\vdash P \triangleright \Delta}$$

Fig. 22. Typing Rule for Recursive Types (equi-recursive approach)

(T-Rec)

$$\frac{P^{\Gamma'} \vdash \mathbf{rely}\, A' \,\mathbf{guar}\, B' \quad \Gamma' \approx \Gamma \quad A' \approx A \quad B' \approx B}{P^{\Gamma} \vdash \mathbf{rely}\, A \,\mathbf{guar}\, B}$$

Fig. 23. The Proof Rule for Recursive Types

8.4 Treatment of Recursive Types

Recursive types are essential for representing complex data structure, as exemplified by their use in many programming languages. In the present setting, we first extend the syntax of types to:

$$\tau \ ::= \ \ldots \mid \mu \mathrm{X}.\tau$$

where we assume τ itself is not a type variable. As examples of recursive types, $\mu\mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ gives a type for an untyped λ-calculus, while (combining with polymorphism) $\forall\mathrm{X}.\mu\mathrm{Y}.[\varepsilon \oplus (\mathrm{XY})^{\uparrow}]^{\uparrow}$ is a type for polymorphic lists.

There are two ways to treat recursive types, as detailed in the case of functional calculi in [96]. In one, we explicitly fold and unfold recursion in terms and types. This framework is called "iso-recursive" approach in [96].

An alternative approach, called "equi-recursive approach" in [96], identifies a recursive type upto its folding/unfolding. To be sufficiently general, we in fact consider recursive types modulo their standard (regular) tree unfoldings, and substitute a recursive type for another as far as both denote the same tree. With $\approx$ denoting the equivalence on types induced by their tree representations, and $\Gamma \approx \Delta$ meaning its pointwise extension to action types, this type substitution can be made explicit by the rule given in Figure 22. For the proof rule, we add the rule given in Figure 23, where $A \approx B$ indicates the substitutions of types up to $\approx$. The equi-recursive approach is particularly convenient in the present study since it incurs no change in syntax of assertions (except for annotating types). Except the collection of typable processes are enlarged, the semantics of assertions also need no change. Because of its simplicity, we shall take this approach in this paper, even though a more explicit approach would have a different merit.

$$[T\text{-}Abs]\ \frac{\Gamma \vdash V : \alpha \quad \mathrm{X} \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda \mathrm{X}.V : \forall \mathrm{X}.\alpha}$$

$$[T\text{-}App]\ \frac{\Gamma \vdash M : \forall \mathrm{X}.\alpha}{\Gamma \vdash M\beta : \alpha[\beta/\mathrm{X}]}$$

$$[T\text{-}Pack]\ \frac{\Gamma \vdash V : \alpha[\beta/\mathrm{X}]}{\Gamma \vdash \mathtt{pack}\,\langle \beta, V\rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha : \exists \mathrm{X}.\alpha}$$

$$[T\text{-}Unpack]\ \frac{\Gamma \vdash M : \exists \mathrm{X}.\alpha \quad \Gamma \cdot x\!:\!\alpha \vdash N : \beta \quad \mathrm{X} \notin \mathsf{ftv}(\Gamma) \cup \mathsf{ftv}(\beta)}{\Gamma \vdash \mathtt{unpack}\, M\,\mathtt{as}\,\langle \mathrm{X}, x\rangle\,\mathtt{in}\, N : \beta}$$

$$[T\text{-}Rec]\ \frac{\Gamma \vdash M : \alpha \quad \alpha \approx \beta}{\Gamma \vdash M : \beta}$$

Fig. 24. Added Typing Rules for $\mathrm{PCF}_v^{\forall\exists\mu}$

8.5 A Logic for $\mathbf{PCF}_v^{\forall\exists\mu}$ (1): Types and Terms

Polymorphism and recursive types offer powerful abstraction mechanisms for programming languages. In the rest of the section, we explore how we can develop general program logics for these advanced data structures guided by the underlying process logic, on the basis of the total logic for PCF_v with unit, sums and products (these data types, together with their logics, are discussed at the end of §6.5, Page 70).

We first extend PCF_v with the second-order polymorphism (including both universals and existentials) and recursive types. Our presentation follows [96] except for a couple of minor points. The types and terms are extended as follows.

$$\begin{array}{rcl} \alpha & ::= & * \ldots \mid \mathrm{X} \mid \forall \mathrm{X}.\alpha \mid \exists \mathrm{X}.\alpha \mid \mu \mathrm{X}.\alpha \\ M & ::= & \ldots \mid \Lambda \mathrm{X}.V \mid M\beta \mid \mathtt{pack}\,\langle \beta, V\rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha \mid \mathtt{unpack}\, M\,\mathtt{as}\,\langle \mathrm{X}, x\rangle\,\mathtt{in}\, N. \end{array}$$

The set of values are inductively extended to include those terms of the shape $\Lambda \mathrm{X}.V$ and $\mathtt{pack}\,\langle \beta, V\rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha$ (we restrict universal abstraction to values since this gives a slightly simpler encoding, though this is not essential). The reduction rules for these constructs are:

$$\begin{array}{l} (\Lambda \mathrm{X}.V)\beta \ \longrightarrow\ V[\beta/\mathrm{X}] \\ \mathtt{unpack}\,(\mathtt{pack}\,\langle \beta, V\rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha)\,\mathtt{as}\,\langle \mathrm{X}, x\rangle\,\mathtt{in}\, N \ \longrightarrow\ N[\beta/\mathrm{X}][V/x] \end{array}$$

The typing rules are given in Figure 24. In the rules for polymorphism, the constructors (type abstraction and pack) and the destructors (type concretion and unpack) compensate with each other. Later we shall uncover the duality underlying these two pairs of rules by translation into processes. In the rule for recursive type (again based on the equi-recursive approach), $\approx$ denotes the tree equivalence of two types, which in particular includes the standard (un)folding, $\mu X.\alpha \approx \alpha[\mu X.\alpha/X]$. We shall call the resulting calculus $\mathrm{PCF}_v^{\forall\exists\mu}$.

Let us explore the extended constructs through a few examples. As a simple instance of the universal abstraction, we take the standard universal identity.

$$\vdash \Lambda X.\lambda x^X.x : \forall X.X \Rightarrow X. \tag{8.16}$$

To use this identity, we instantiate it with a type and feed a value of that type:

$$((\Lambda X.\lambda x^X.x)\mathbb{N})2 \;\longrightarrow\; (\lambda x^{\mathbb{N}}.x)2 \;\longrightarrow\; 2. \tag{8.17}$$

The next example is a type for a list of elements of type α, using recursive types.

$$\mathtt{List}(\alpha) \stackrel{\text{def}}{=} \mu Y.(\mathtt{Unit} + (\alpha \times Y)). \tag{8.18}$$

The empty list is represented by $\mathtt{inl}(())$, while the list which consists of a head M (of type α) and a tail L (of type $\mathtt{List}(\alpha)$) becomes $\mathtt{inr}(\langle M, L\rangle)$, which is of type $\mathtt{List}(\alpha)$ by $\mathtt{Unit} + \alpha \times \mathtt{List}(\alpha) \approx \mathtt{List}(\alpha)$. A list can be infinite, e.g. $\mu x.\mathtt{inr}(\langle 1, x\rangle)$ is an infinite list of 1 and has type $\mathtt{List}(\mathbb{N})$. In our subsequent discussions we often treat lists. For legibility we use the following notations.

$$[\varepsilon] \stackrel{\text{def}}{=} \mathtt{inl}(()), \qquad\qquad [M :: L] \stackrel{\text{def}}{=} \mathtt{inr}(\langle M, L\rangle).$$

As is well-known, it is often convenient to treat lists generically, i.e. without depending on its types. In such situations, we can use the polymorphic list type $\forall X.\mathtt{List}(X)$. A simple example is the generic cons function:

$$\Lambda X.\, \lambda x^X.\lambda l^{\mathtt{List}(X)}.[x :: l]. \tag{8.19}$$

A little more complex example is a program which calculates the length of a list of an arbitrary type.

$$\Lambda X.\, \mu f^{\mathtt{List}(X)\Rightarrow\mathbb{N}}.\, \lambda l^{\mathtt{List}(X)}.\, \mathtt{case}\ l\ \mathtt{of}\ \mathsf{isNil} \Rightarrow 0 \mid \mathsf{isCons}(x, l') \Rightarrow f(l') + 1, \tag{8.20}$$

Above we use the $\mathtt{case\text{-}list}$ construct "$\mathtt{case}\ l\ \mathtt{of}\ \mathsf{isNil} \Rightarrow M \mid \mathsf{isCons}(x, l') \Rightarrow N$" which stands for "$\mathtt{case}\ l\ \mathtt{of}\ \{\mathtt{inl}(z).M \mid \mathtt{inr}(y).((\lambda x l'.N)\pi_l(y))\pi_r(y)\}$" ($z$ fresh).

Yet another example which uses a polymorphic list is a program which filters out elements of a list that satisfy a given predicate.

$$\vdash \Lambda X.\, \lambda p^{X\Rightarrow\mathbb{B}}.\, \mathsf{filter}(p) \;:\; \forall X.\, (X \Rightarrow \mathbb{B}) \Rightarrow \mathtt{List}(X) \Rightarrow \mathtt{List}(X) \tag{8.21}$$

where $\mathsf{filter}(p)$ is the following expressions:

$$\mu f.\lambda l.\mathtt{case}\ l\ \mathtt{of}\ \mathsf{isNil} \Rightarrow [\varepsilon] \mid \mathsf{isCons}(x, y) \Rightarrow \mathtt{if}\ p(x)\ \mathtt{then}\ [x :: f(y)]\ \mathtt{else}\ f(y).$$

We end our brief exploration with an example of existential abstraction.

$$\begin{aligned} \mathsf{cntr} \;\stackrel{\text{def}}{=}\; & \mathtt{pack}\ \ \langle \mathbb{N}, \{\mathtt{new} : \lambda().0,\ \ \mathtt{inc} : \lambda x.x + 1,\ \ \mathtt{get} : \lambda x.x\}\rangle \\ & \quad \mathtt{as}\ \ \exists X.\{\mathtt{new} : \mathtt{Unit} \Rightarrow X,\ \ \mathtt{inc} : X \Rightarrow X,\ \ \mathtt{get} : X \Rightarrow \mathbb{N}\}. \end{aligned} \tag{8.22}$$

The type represents an abstract data type for a counter, which has three methods: $\mathtt{new}$ exports an opaque integer, $\mathtt{inc}$ increments it, and $\mathtt{get}$ turns an opaque value to the corresponding non-opaque value.

$$[\textit{T-Abs}]\ \frac{A^{\text{-X}} \vdash V^{\Gamma;\alpha} :_m B}{A \vdash \Lambda \mathrm{X}.V^{\Gamma;\forall \mathrm{X}.\alpha} :_u \forall \mathrm{X}.B[u \bullet [\mathrm{X}]/m]}$$

$$[\textit{T-App}]\ \frac{A \vdash M^{\Gamma;\forall \mathrm{X}.\alpha} :_m \forall \mathrm{X}.B[m \bullet [\mathrm{X}]/u]}{A \vdash M\beta^{\ \Gamma;\alpha[\beta/\mathrm{X}]} :_u B[\beta/\mathrm{X}]}$$

$$[\textit{T-Pack}]\ \frac{A \vdash V^{\Gamma;\alpha[\beta/\mathrm{X}]} :_m B[\langle \mathrm{X}, m\rangle/u][\beta/\mathrm{X}]}{A \vdash \mathtt{pack}\ \langle \beta, V\rangle\ \mathtt{as}\ \exists \mathrm{X}.\alpha^{\Gamma;\exists \mathrm{X}.\alpha} :_u \exists \mathrm{X}.B}$$

$$[\textit{T-Unpack}]\ \frac{A \vdash M^{\Gamma;\exists \mathrm{X}.\alpha} :_m \exists \mathrm{X}.E \quad A^{\text{-}x\mathrm{X}} \wedge E[\langle \mathrm{X}, x\rangle/m] \vdash N^{\Gamma \cdot x:\alpha;\beta} :_u B}{A \vdash \mathtt{unpack}\ M\ \mathtt{as}\ \langle \mathrm{X}, x\rangle\ \mathtt{in}\ N^{\Gamma;\beta} :_u B}$$

$$[\textit{T-Rec}]\ \frac{A' \vdash M^{\Gamma;\alpha'} :_m B' \quad \alpha' \approx \alpha \quad A' \approx A \quad B' \approx B}{A \vdash M^{\Gamma;\alpha} :_m B}$$

Fig. 25. Added Proof Rules for $\mathrm{PCF}_v^{\forall\exists\mu}$ (explicitly typed presentation)

8.6 A Logic for $\mathbf{PCF}_v^{\forall\exists\mu}$ (2): Assertions and Proof Rules

The total logic for $\mathrm{PCF}_v^{\forall\exists\mu}$ extends that of PCF_v based on the corresponding process logic, and offers a natural specification/reasoning tool for advanced data structures. The terms of the logical language add:

$$e \quad ::= \quad \ldots \quad | \quad e \bullet [\beta] \quad | \quad \langle \beta, e\rangle^{\exists \mathrm{X}.\alpha}$$

For these terms, the following typing constraints should be obeyed:

– In $e \bullet [\beta]$, e should have a type of shape $\forall \mathrm{X}.\alpha$, in which case the whole term has the type $\alpha[\beta/\mathrm{X}]$.
– In $\langle \beta, e\rangle^{\exists \mathrm{X}.\alpha}$, e should have a type of shape $\alpha[\beta/\mathrm{X}]$, and the whole term has the type $\exists \mathrm{X}.\alpha$.

Operationally $e^{\forall \mathrm{X}.\alpha} \bullet [\beta]$ corresponds to $M\beta$, while $\langle \beta, e^{\alpha[\beta/\mathrm{X}]}\rangle$ corresponds to a packed pair: further elucidation will be given later using their process encodings. To the formulae of the PCF_v-logic, we add:

$$A \quad ::= \quad \ldots \quad | \quad \forall \mathrm{X}.\, A \quad | \quad \exists \mathrm{X}.\, A.$$

Both act as binders in the obvious way. We thus reach the assertion of the same form as before, $A \models M^{\Gamma;\alpha} :_u B$ for validity and $A \vdash M^{\Gamma;\alpha} :_u B$ for provability, where A and B should be typed under Γ and $u : \alpha$ respectively except for auxiliary names, saying, as before: M guarantees the specification B relying on the behaviour of the environment as stipulated in A.

The semantics of formulae and assertions also extends that of the total PCF_v-logic (cf. §6.4, Page 68). *Behaviours* $(\lambda, \ldots)$ are closed $\mathrm{PCF}_v^{\forall\exists\mu}$-terms of closed types up to the contextual congruence, with well-typed operations such as $\lambda \bullet \lambda'$, $\lambda \bullet \tau$, $\lambda \bullet \tau$ and $\langle \beta, \lambda \rangle^{\exists \mathrm{X}.\alpha}$ defined from the corresponding operations on terms in the obvious way. As before, for each non-empty (closed) type τ, we have a unique divergent behaviour, which we write $\bot^{\alpha}$.

A *model* $(\xi, \ldots)$ is a finite map from names to non-divergent behaviours. The satisfaction $\xi \models^{\mathcal{I}} A$ is given precisely as before (with $\mathcal{I}$ being a finite map from names to possibly divergent behaviours), interpreting the added constructs using the operations on behaviours noted above. This leads to the relation $\vec{x}\!:\!\vec{V} \models^{\mathcal{I}} A$, from which we define:

$$M \models^{\mathcal{I}}_{u} A \quad \equiv \quad M \Downarrow V \ \wedge \ u\!:\!V \models^{\mathcal{I}} A \tag{8.23}$$

Finally the semantics of assertions is given by, with $\mathsf{dom}(\Gamma) = \{\vec{x}\}$:

$$A \models M^{\Gamma;\alpha} :_{u} B \quad \equiv \quad \vec{x}\!:\!\vec{V}^{\Gamma} \models^{\mathcal{I}} A \supset M[\vec{V}/\vec{x}] \models^{\mathcal{I}}_{u} B. \tag{8.24}$$

which is precisely as in the total PCF_v-logic.

One of the drawbacks of taking such a syntactic approach is that it is not always straightforward to derive basic properties of the model. We can however appeal to the fully abstract embedding of $\mathrm{PCF}_v^{\forall\exists\mu}$ in the π-calculus,[5] where properties are easier to derive, and reflect the properties of operators of processes onto the program logic. Among others the embedding gives us the extensionality of such operators as $e \bullet e'$, $e \bullet [\alpha]$, and $\langle e, e' \rangle$ (via the context lemma in the π-calculus), and the entailments $\forall \mathrm{X}.A \supset A[\alpha/\mathrm{X}]$ and $A[\alpha/\mathrm{X}] \supset \exists \mathrm{X}.A$ (in both entailments we assume the action type of the formula does not change by quantification).

The proof rules for this logic are given in Figure 25. In each rule, terms are formally type-annotated so that the correspondence between the change of types and that of formulae can be seen clearly. As always, the rules assume the well-typedness of occurring assertions. Each rule for polymorphism naturally follows the construction/destruction of the concerned type. In the rule for recursive types, $A \approx A'$ denotes, as before, two formulae are identical except for substituting $\approx$-related types. The usage of these rules in inferences will be shown in Section 8.8 later.

[5] Indeed, one may as well give the model of the total $\mathrm{PCF}_v^{\forall\exists\mu}$-logic in the universe of polymorphic processes we discussed in the previous subsections. The validity of assertions is precisely the same as the direct model we discussed above, while the derivation of properties is more straightforward.

8.7 Process Encoding of $\mathrm{PCF}_v^{\forall\exists\mu}$-Logic

This subsection explores the process encoding of the $\mathrm{PCF}_v^{\forall\exists\mu}$-logic. First we present the encoding of types and terms of $\mathrm{PCF}_v^{\forall\exists\mu}$.

$$\begin{aligned}
\mathrm{X}^\circ &\stackrel{\mathrm{def}}{=} \mathrm{X} \\
(\forall \mathrm{X}.\alpha)^\circ &\stackrel{\mathrm{def}}{=} \forall \mathrm{X}.((\alpha^\circ)^\uparrow) \\
(\exists \mathrm{X}.\alpha)^\circ &\stackrel{\mathrm{def}}{=} (\exists \mathrm{X}.(\alpha^\circ)^\uparrow)^! \\
(\mu \mathrm{X}.\alpha)^\circ &\stackrel{\mathrm{def}}{=} \mu \mathrm{X}.\alpha^\circ \\
\langle\!\langle \Lambda \mathrm{X}.V \rangle\!\rangle_u^{\mathrm{val}} &\stackrel{\mathrm{def}}{=} !u(\mathrm{X})(c).\overline{c}(m)\langle\!\langle V \rangle\!\rangle_m \\
\langle\!\langle M\beta \rangle\!\rangle_u &\stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, m)(\langle\!\langle M \rangle\!\rangle_m \mid m(c).\overline{c}[\beta^\circ](c)[c \triangleright u]) \\
\langle\!\langle \mathtt{pack}\, \langle \beta, V\rangle\, \mathtt{as}\, \exists \mathrm{X}.\alpha \rangle\!\rangle_u^{\mathrm{val}} &\stackrel{\mathrm{def}}{=} !u(c).\overline{c}[\beta^\circ](m)\langle\!\langle V \rangle\!\rangle_m^{\mathrm{val}} \\
\langle\!\langle \mathtt{unpack}\, M\, \mathtt{as}\, \langle \mathrm{X}, x\rangle\, \mathtt{in}\, N \rangle\!\rangle_u &\stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\, m)(\langle\!\langle M \rangle\!\rangle_m \mid m(e).\overline{e}(c)c(\mathrm{X})(x).\langle\!\langle N \rangle\!\rangle_u)
\end{aligned}$$

Thus a universal abstraction adds one indirection for receiving a type, while a concretion dually sends a type to a type abstraction. The packing with an existential abstraction adds one indirection for outputting a type, while the unpacking dually lets the (encoding of the) program N receive that type. As in the case of atomic values, the linear actions (for indirections) in the above encoding should in fact be truly linear, in the sense that no divergent actions should be among their inhabitants.

To understand the above encodings of type abstractions/concretions, it is instructive to see how the encoded terms behave in dynamics. Below we use two basic syntactic equalities [80] (both staying within strong bisimilarity), $\equiv_g$ and $\equiv_r$. These are the congruent closure of the garbage collection law $(\boldsymbol{\nu}\, x)!x(\vec{y}z).P \equiv_g \mathbf{0}$ and the replication law $!x(\vec{y}z).P|R \equiv_g !x(\vec{y}z).P|(\boldsymbol{\nu}\, w)(!w(\vec{y}z).P|R[w/x])$, respectively. We start from the reduction associated with type application.

$$\begin{aligned}
&\langle\!\langle (\Lambda \mathrm{X}.V)\beta \rangle\!\rangle_u \\
&\quad \stackrel{\mathrm{def}}{=} \quad (\boldsymbol{\nu}\, m)(\ \overline{m}(c)!c(\mathrm{X})(v).\overline{v}(n)\langle\!\langle V \rangle\!\rangle_n \mid m(c).\overline{c}[\beta^\circ]\langle u\rangle\ ) \\
&\quad \longrightarrow \quad (\boldsymbol{\nu}\, c)(\ !c(\mathrm{X})(v).\overline{v}(n)\langle\!\langle V \rangle\!\rangle_n^{\mathrm{val}} \mid \overline{c}[\beta^\circ]\langle u\rangle\ ) \\
&\quad \longrightarrow \quad (\boldsymbol{\nu}\, v)(\ \overline{v}(n)\langle\!\langle V \rangle\!\rangle_n \mid [v \triangleright u]\ ) \mid (\boldsymbol{\nu}\, c)!c(\mathrm{X})(v).\overline{v}(n)\langle\!\langle V \rangle\!\rangle_n \\
&\quad \longrightarrow\equiv_g \overline{u}(n)\langle\!\langle V \rangle\!\rangle_n[\beta/\mathrm{X}]
\end{aligned}$$

On the other hand, the unpacking becomes:

$$\begin{aligned}
&\langle\!\langle \mathtt{unpack}\, (\mathtt{pack}\, \langle \beta, V\rangle\, \mathtt{as}\, \exists \mathrm{X}.\alpha)\, \mathtt{as}\, \langle \mathrm{X}, x\rangle\, \mathtt{in}\, N \rangle\!\rangle_u \\
&\quad \stackrel{\mathrm{def}}{=} \quad (\boldsymbol{\nu}\, m)(\ \overline{m}(e)!e(c).\overline{c}[\beta^\circ](v)\langle\!\langle V \rangle\!\rangle_v^{\mathrm{val}} \mid m(e).\overline{e}(c)c(\mathrm{X})(x).\langle\!\langle N \rangle\!\rangle_u\ ) \\
&\quad \longrightarrow \quad (\boldsymbol{\nu}\, e)(\ !e(c).\overline{c}[\beta^\circ](v)\langle\!\langle V \rangle\!\rangle_v^{\mathrm{val}} \mid \overline{e}(c).c(\mathrm{X})(x).\langle\!\langle N \rangle\!\rangle_u\ ) \\
&\quad \longrightarrow \quad (\boldsymbol{\nu}\, c)(\ c(\mathrm{X})(x).\langle\!\langle N \rangle\!\rangle_u \mid \overline{c}[\beta^\circ](v)\langle\!\langle V \rangle\!\rangle_v^{\mathrm{val}}\ ) \mid (\boldsymbol{\nu}\, e)!e(c).\overline{c}[\beta^\circ](v)\langle\!\langle V \rangle\!\rangle_m^{\mathrm{val}} \\
&\quad \equiv_g \quad (\boldsymbol{\nu}\, v)(\ \langle\!\langle N[\beta^\circ/\mathrm{X}] \rangle\!\rangle_u \mid \langle\!\langle V \rangle\!\rangle_v^{\mathrm{val}}\ ) \\
&\quad \equiv_r \quad \langle\!\langle N[\beta^\circ/\mathrm{X}][V/x] \rangle\!\rangle_u
\end{aligned}$$

The reduction of encodings clearly demonstrates the dual nature of universals and existentials.

The mapping of the terms and formulae of the $\mathrm{PCF}_v^{\forall\exists\mu}$-Logic to those of the process logic closely follows that of types, adding the following clauses to those of the PCF_v-logic.

$$\begin{aligned}
(e \bullet [\alpha])^\circ_u &\stackrel{\text{def}}{=} \exists x.\, (e^\circ_x \;\wedge\; (u)^\uparrow = x \bullet [\alpha^\circ]\varepsilon) \\
(\langle \beta, e\rangle^{\exists \mathrm{X}.\alpha})^\circ_u &\stackrel{\text{def}}{=} \exists x.(e^\circ_x \;\wedge\; u \bullet \varepsilon = [\beta^\circ]^{\exists \mathrm{X}.(\alpha^\circ)}(x)^\uparrow) \\
(\forall \mathrm{X}.A)^\circ &\stackrel{\text{def}}{=} \forall \mathrm{X}.A^\circ \\
(\exists \mathrm{X}.A)^\circ &\stackrel{\text{def}}{=} \exists \mathrm{X}.A^\circ.
\end{aligned}$$

Note the encodings in the first two lines do not assume the possibility of divergence at the linear output, which is because, as we noted above, these linear outputs are indeed truly linear (hence are never inhabited by divergent behaviours), even though we omit type annotations.

The encoding gives us the decomposition and justification of the proof rules of the $\mathrm{PCF}_v^{\forall\exists\mu}$-logic by those of the process logic. For example, $[\mathit{T\text{-}Abs}]$ is decomposed into several steps of inferences in the process logic, as given below.

1.	$A \vdash V :_m B$	(Ass.)
2.	$\langle\!\langle V \rangle\!\rangle^{\mathrm{val}}_m \vdash \mathbf{rely}\ A^\circ\ \mathbf{guar}\ B^\circ$	(Ind. Hyp.)
3.	$\overline{c}(m)\langle\!\langle V \rangle\!\rangle^{\mathrm{val}}_m \vdash \mathbf{rely}\ A^\circ\ \mathbf{guar}\ \exists m.\,(c = (m)^\uparrow \;\wedge\; B^\circ)$	(Bra-fo)
4.	$!u(\mathrm{X})(c).\overline{c}(m)\langle\!\langle V \rangle\!\rangle^{\mathrm{val}}_m \vdash \mathbf{rely}\ A^\circ\ \mathbf{guar}\ \forall \mathrm{X}.\,\exists m.\,(u \bullet [\mathrm{X}]\varepsilon = (m)^\uparrow \;\wedge\; B^\circ)$	(In-$\forall$)
5.	$!u(\mathrm{X})(c).\overline{c}(m)\langle\!\langle V \rangle\!\rangle^{\mathrm{val}}_m \vdash \mathbf{rely}\ A^\circ\ \mathbf{guar}\ \forall \mathrm{X}.\,(B[u \bullet [\mathrm{X}]/m])^\circ$	(Cons)

In Line 5, the following logical equivalence is used.

$$(m = u \bullet [\mathrm{X}]\varepsilon)^\circ \;\stackrel{\text{def}}{=}\; \exists xy.(m = x \;\wedge\; (y)^\uparrow = u \bullet [\mathrm{X}]\varepsilon \;\wedge\; x = y) \;\;\equiv\;\; (m)^\uparrow = u \bullet [\mathrm{X}]\varepsilon.$$

We also treat $[\mathit{T\text{-}App}]$. We use the following notations for brevity.

$$G(c) \stackrel{\text{def}}{=} \forall \mathrm{X}.\exists g.(B^\circ[g/u] \wedge c \bullet [\mathrm{X}]\varepsilon = (g)^\uparrow) \tag{8.25}$$

$$(\forall \mathrm{X}.B[m \bullet [\mathrm{X}]/u])^\bullet_e \stackrel{\text{def}}{=} \exists c.\,(e = (c)^\uparrow \wedge G(c)) \tag{8.26}$$

$$E(u) \stackrel{\text{def}}{=} \exists g.(B^\circ[g/u][\beta^\circ/\mathrm{X}] \;\wedge\; u = (g)^\uparrow) \tag{8.27}$$

The inference follows.

1.	$A \vdash M :_m \forall \mathrm{X}.B[m \bullet [\mathrm{X}]/u]$	(Ass.)
2.	$\langle\!\langle M \rangle\!\rangle_e \vdash \mathbf{rely}\ A^\circ\ \mathbf{guar}\ \exists c.\,(e = (c)^\uparrow \wedge G(c))$	(Ind. Hyp., (8.26) above)
3.	$[u' \triangleright u] \vdash \mathbf{rely}\ G(c) \wedge E(u')\ \mathbf{guar}\ E(u)$	(Wire, Conseq)
4.	$\overline{c}[\beta]\langle u \rangle \vdash \mathbf{rely}\ G(c)\ \mathbf{guar}\ E(u)$	(3, Out-$\exists$)
5.	$e(c).\overline{c}[\beta]\langle u \rangle \vdash \mathbf{rely}\ \exists c.(e = (c)^\uparrow \wedge G(c))\ \mathbf{guar}\ E(u)$	(5, Bra-fo)
6.	$(\boldsymbol{\nu}\, e)(\langle\!\langle M \rangle\!\rangle_e \mid e(c).\overline{c}[\beta]\langle u \rangle) \vdash \mathbf{rely}\ A^\circ\ \mathbf{guar}\ E^\circ[\beta^\circ/\mathrm{X}]$	(2, 5, Par, Res)

where in Line 4 the following implications are used (via (8.25) and (8.27)).

$$G(c) \;\supset\; (\exists g.(B^{\circ}[g/u] \wedge c \bullet [\mathrm{X}]\varepsilon = (g)^{\uparrow}))[\beta^{\circ}/\mathrm{X}] \;\stackrel{\mathrm{def}}{=}\; E(u)[c \bullet [\beta^{\circ}]\varepsilon/u]$$

The remaining rules are similarly verified.

8.8 A Logic for $\mathbf{PCF}_v^{\forall\exists\mu}$ (3): Reasoning Examples

We conclude this section with three simple examples of derivations of assertions for $\mathrm{PCF}_v^{\forall\exists\mu}$, using the proof rules in Figure 25. After presenting three examples, we also show how a terser but still lucid specification in reasoning is possible by sound simplifications. All $\mathrm{PCF}_v^{\forall\exists\mu}$-programs which are reasoned below come from Section 8.5. For legibility we use the following convention in the rest of the section.

Convention 4. In the following derivations we allow free variables of the terms to appear in the guarantee formula.

As we discussed in Remark 6, §subsec:pcfv:proofrules, Page 76, this is more convenient when we have many variables and whose semantics simply extends that of the original one while not necessitating any change in proof rules (we have been sticking to the strict distinction between negative names (free variables in the term) and positive names (the anchor) so far because, for our meta-logical inquiries, the distinction gives a clearer conceptual footing while not jeopardising generality; however, for writing down non-trivial proofs, this convention is quite convenient). Note that, unlike in VDM [66] or an extended version of Hoare Logic which incorporates this idea [49], we do not need to distinguish hooked and unhooked variables, since the negative names in the present logic are stateless — they do not change what they denote how many times they are used/invoked.

Inference Examples (1): Universal Identity. We infer the property of the universal identity, presented in (8.16).

1.	$\mathsf{T} \vdash x :_m m^{\mathrm{X}} = x^{\mathrm{X}}$	(Var)
2.	$\mathsf{T} \vdash \lambda x^{\mathrm{X}}.x :_u \forall x^{\mathrm{X}}.u^{\mathrm{X}\Rightarrow\mathrm{X}} \bullet x = x$	(Abs)
3.	$\mathsf{T} \vdash \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x :_u \forall \mathrm{X}.\forall x^{\mathrm{X}}.(u^{\forall\mathrm{X}.\mathrm{X}\Rightarrow\mathrm{X}} \bullet [\mathrm{X}]) \bullet x = x$	(T-Abs)
4.	$\mathsf{T} \vdash \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x :_u \forall x^{\mathbb{B}}.(u^{\forall\mathrm{X}.\mathrm{X}\Rightarrow\mathrm{X}} \bullet [\mathbb{B}]) \bullet x = x$	(3. Conseq)
5.	$\mathsf{T} \vdash \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x :_u \forall x^{\mathbb{N}}.(u^{\forall\mathrm{X}.\mathrm{X}\Rightarrow\mathrm{X}} \bullet [\mathbb{N}]) \bullet x = x$	(3, Conseq)

In Line 1, we apply [*Var*] under Convention 4. In the entailments in Lines 4 and 5, observe the guarantee formulae in Lines 3, 4 and 5 own the identical action type, $u : \forall \mathrm{X}.\mathrm{X} \Rightarrow \mathrm{X}$, by which we can freely instantiate X to a concrete type. From the above inference we can then easily infer, e.g. $\mathsf{T} \vdash ((\mathtt{ID}\,\mathbb{N})3) :_u = 3$, where ID denotes the universal identity

Inference Examples (2). Polymorphic List. Recall the program $\mathsf{filter}(f)$ which filters a list using a function $f : \mathrm{X} \Rightarrow \mathbb{B}$, given in (8.21). The following program takes off all occurrences of 0 from a list of natural numbers using $\mathsf{filter}(f)$.

$$\mathsf{elimZero} \stackrel{\text{def}}{=} ((\Lambda \mathrm{X}.\, \lambda f^{\mathrm{X}\Rightarrow\mathbb{B}}.\, \mathsf{filter}(f))\mathbb{N})(\lambda x^{\mathbb{N}}.x \neq 0).$$

We can easily see this program does not terminate if a list fed to the program is infinite. Thus, to prove total correctness, we should specify that a given list is well-founded (or is finite in length) and each element terminates. We define this notion together with a related predicate which gives the length of a finite list, as derived predicates.

$$\begin{array}{rcl} \mathsf{Wf}(l) & \equiv & \exists n^{\mathbb{N}}.\, \mathsf{Len}(l,n). \\ \mathsf{Len}(l,0) & \equiv & l = [\varepsilon]. \\ \mathsf{Len}(l,n+1) & \equiv & \exists x, l'.\, l = [x :: l'] \wedge x \neq \bot \wedge \mathsf{Len}(l', n). \end{array}$$

One of the natural specifications for $\mathsf{elimZero}$ is given as:

$$\mathit{ElimZero}(u) \stackrel{\text{def}}{=} \forall l^{\mathtt{List}(\mathbb{N})}.\, (\mathsf{Wf}(l) \supset A(u,l) \wedge B(u,l)) \tag{8.28}$$

where we set:

$$\begin{array}{l} A(u,l) \stackrel{\text{def}}{=} l = [\varepsilon] \supset u \bullet l = [\varepsilon]. \\ B(u,l) \stackrel{\text{def}}{=} l = [x :: y] \ \wedge \ x \neq \bot \supset (x \neq 0 \supset u \bullet l = [x :: u \bullet y]) \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad (x = 0 \supset u \bullet l = u \bullet y). \end{array}$$

The predicate $\mathit{ElimZero}(u)$ (on u of type $\mathtt{List}(\mathbb{N}) \Rightarrow \mathtt{List}(\mathbb{N})$) says that, assuming an integer list is well-founded, applying the list to u would result in another integer list which is the same as the original one except that each 0 in the list is eliminated one by one. Our aim is to prove:

$$\mathsf{T} \vdash \mathsf{elimZero} :_u \mathit{ElimZero}(u). \tag{8.29}$$

To derive (8.29), the key step is to infer the following generic assertion for the program $\mathsf{filter}(f)$.

$$\mathsf{T} \vdash \mathsf{filter}(f) :_u \forall l.\ \mathit{Filter}(u,l,f) \tag{8.30}$$

where $\mathit{Filter}(u,l,f)$ says $u \bullet l$ is the result of filtering l by f:

$$\mathit{Filter}(u,l,f) \stackrel{\text{def}}{=} \mathsf{Wf}(l) \supset A(u,l) \wedge C(u,l,f) \tag{8.31}$$

where $C(u,l,f)$ is defined as follows:

$$\begin{array}{l} l = [x :: y] \ \wedge \ x \neq \bot \supset (f \bullet x = \mathsf{T} \supset u \bullet l = [x :: u \bullet y]) \wedge \\ \qquad\qquad\qquad\qquad (f \bullet x = \mathsf{F} \supset u \bullet l = u \bullet y). \end{array}$$

Once we have (8.30), we can reach (8.29) as follows.

1.	$\mathsf{T} \vdash \mathsf{filter}(f) :_u \forall l.\ \mathit{Filter}(u,l,f)$	(Assumption)
2.	$\mathsf{T} \vdash \lambda f.\mathsf{filter}(f) :_u \forall f,l.\ \mathit{Filter}(u \bullet f,l,f)$	(Abs)
3.	$\mathsf{T} \vdash \Lambda \mathrm{X}.\lambda f.\mathsf{filter}(f) :_u \forall \mathrm{X}.\forall f,l.\ \mathit{Filter}(u \bullet [\mathrm{X}] \bullet f,l,f)$	(T-Abs)
4.	$\mathsf{T} \vdash (\Lambda \mathrm{X}.\lambda f.\mathsf{filter}(f))\mathbb{N} :_u \forall f,l.\ \mathit{Filter}(u \bullet f,l,f)$	(T-App)
5.	$\mathsf{T} \vdash \lambda x.(x \neq 0) :_f \mathit{NonZero}(f)$	(Var, Num, Eq, Abs)
6.	$\mathsf{T} \vdash ((\Lambda \mathrm{X}.\lambda f.\mathsf{filter}(f))\mathbb{N})(\lambda x.x \neq 0) :_u \mathit{ElimZero}(u)$	(4, 5, App)

In Line 5, $\mathit{NonZero}(f)$ stands for

$$\forall x.((x = 0 \supset f \bullet x = \mathsf{F}) \wedge (x = n \geq 1 \supset f \bullet x = \mathsf{T})$$

Then in Line 6, we calculate the assertions as:

$$\forall l.\ \mathit{Filter}(u \bullet f,l,f) \wedge \mathit{NonZero}(f) \supset \mathit{ElimZero}(u \bullet f)$$

We now derive (8.30). The derivation is, as usual, divided into the base case and the induction. For reasoning, we first note the following inductive definition of the filter predicate.

$$\begin{aligned}
\mathit{Filter}(u,l,f) &\equiv \forall n^{\mathbb{N}}.\,(\mathsf{Len}(l,n) \supset \mathit{Fil}(l,f,u \bullet l,n)) \\
\mathit{Fil}(l,f,l',0) &\stackrel{\text{def}}{=} l = l' \\
\mathit{Fil}(l,f,l',n{+}1) &\stackrel{\text{def}}{=} (l{=}[\varepsilon] \supset l'{=}[\varepsilon]) \wedge \\
&\qquad (l = [x :: y] \wedge x \neq \bot \supset \\
&\qquad\quad (f \bullet x = \mathsf{T} \supset \\
&\qquad\qquad \exists y'.\,(l'{=}[x :: y'] \ \wedge\ \mathit{Fil}(y,f,y',n))) \wedge \\
&\qquad\quad (f \bullet x = \mathsf{F} \ \supset\ \mathit{Fil}(y,f,l',n))
\end{aligned}$$

The equivalence with (8.31) is immediate by induction on the length of a list. We also let:

$$\begin{aligned}
M &\stackrel{\text{def}}{=} \texttt{case}\ l\ \texttt{of}\ \mathsf{isNil} \Rightarrow [\varepsilon] \mid \mathsf{isCons}(x,y) \Rightarrow N \\
N &\stackrel{\text{def}}{=} \texttt{if}\ f(x)\ \texttt{then}\ [x :: g(y)]\ \texttt{else}\ g(y)
\end{aligned}$$

Note $\mathsf{filter}(f) \stackrel{\text{def}}{=} \mu g.\lambda l.M$. We also use the following abbreviations for brevity.

$$E(n)(u) \ \equiv\ \forall l.(\mathsf{Len}(l,n) \supset \mathit{Fil}(l,f,u \bullet l,n))$$

The base case follows.

1.	$l = [\varepsilon] \vdash l :_{l'} l' = [\varepsilon]$	(Var)
2.	$[\varepsilon] = [\varepsilon] \vdash [\varepsilon] :_m m = [\varepsilon]$	(Nil)
3.	$\mathsf{F} \vdash \texttt{if}\ f(x)\ \texttt{then}\ [x :: g(y)]\ \texttt{else}\ g(y) :_m m = [\varepsilon]$	(falsity)
4.	$l{=}[\varepsilon] \vdash M :_m \mathit{Fil}(l,f,m,0)$	(1,2,3,List)
5.	$\mathsf{T} \vdash \lambda l.M :_m E(0)(m)$	(Abs)

$$[\mathit{Nil}]\ \frac{-}{A[[\varepsilon]/u] \vdash [\varepsilon] :_u A}$$

$$[\mathit{Cons}]\ \frac{A \vdash M :_m B_1 \quad A \vdash L :_l B_2 \quad B_1 \wedge B_2 \supset B[[m :: l]/u]}{A \vdash [M :: L] :_u B}$$

$$[\mathit{List}]\ \frac{A \vdash M :_m E \qquad \begin{array}{l} A \wedge E[[\varepsilon]/m] \vdash N_1 :_u B \\ A \wedge E[[x :: l]/m] \vdash N_2 :_u B \end{array}}{A \vdash \mathtt{case}\ M\ \mathtt{of}\ \mathsf{isNil} \Rightarrow N_1 \mid \mathsf{isCons}(x, l) \Rightarrow N_2 :_u B}$$

Fig. 26. Proof Rules for Lists and Case

In Lines 2 and 4, the derived proof rules for lists in Figure 26 are used where we assume $[\varepsilon]$ and $[e :: e']$ are included among the terms in the logic (which formally stand for $\mathtt{in}_1(())$ and $\mathtt{in}_2(\langle e, e' \rangle)$, respectively).

We move to the induction step. Below we use the following abbreviations:

$$\begin{aligned}
C(lxyn) &\stackrel{\text{def}}{=} l = [x :: y] \wedge \mathsf{Len}(y, n).\\
G(fxl'yn) &\stackrel{\text{def}}{=} (f \bullet x = \mathsf{T} \ \supset\ \exists y'.\, (l' = [x :: y'] \ \wedge\ \mathit{Fil}(y, f, y', n))) \ \wedge\\
&\phantom{\stackrel{\text{def}}{=}}\ (f \bullet x = \mathsf{F} \ \supset \ \ \mathit{Fil}(y, f, l', n))\\
A(lxyng\mathsf{T}) &\stackrel{\text{def}}{=} E(n)(g) \wedge C(lxyn) \wedge f \bullet x = \mathsf{T}\\
A(lxyng\mathsf{F}) &\stackrel{\text{def}}{=} E(n)(g) \wedge C(lxyn) \wedge f \bullet x = \mathsf{F}
\end{aligned}$$

Note $G(fxl'ny)$ is the part of $Fil(l, f, l', n+1)$ when the list is non-empty.

1. $C(lxyn) \vdash l :_{l'} l' = [x :: y] \wedge \mathsf{Len}(y, n)$	(Var)
2. $\mathsf{F} \vdash [\varepsilon] :_m E(n+1)(m)$	(falsity)
3. $\mathsf{T} \vdash f(x) :_b f \bullet x = b$	(Var, Var, App)
4. $A(lxyng\mathsf{T}) \vdash g(y) :_m f \bullet x = \mathsf{T} \wedge Fil(y, f, m, n)$	(Var, Var, App)
5. $A(lxyng\mathsf{T}) \vdash [x :: g(y)] :_m$ $f \bullet x \wedge \exists y'.(m = [x :: y'] \wedge Fil(y, f, y', n))$	(Cons)
6. $A(lxyng\mathsf{T}) \vdash [x :: g(y)] :_m G(fxmyn)$	(Conseq)
7. $A(lxyng\mathsf{F}) \vdash g(y) :_m f \bullet x = \mathsf{F} \wedge Fil(l, f, m, n)$	(Var, Var, App)
8. $A(lxyng\mathsf{F}) \vdash g(y) :_m G(fxmyn)$	(Conseq)
9. $E(n)(g) \wedge C(lxyn) \vdash N :_m G(fxmyn)$	(3,6,8,If)
10. $E(n)(g) \wedge C(lxyn) \vdash M :_m G(fxmyn)$	(1,2,9,List)
11. $E(n)(g) \vdash \lambda l.M :_m \forall lxy.\ (C(lxyn) \supset G(fxmyn))$	(10, Abs)
12. $E(n)(g) \vdash \lambda l.M :_m E(n+1)(m)$	(Conseq)

In the final line, we observe: $\forall lxy.\ (C(lxyn) \supset G(fxmyn)) \supset$
$\forall l.\ (\mathsf{Len}(l, n+1) \supset Fil(l, f, m \bullet l, n+1)) \stackrel{\text{def}}{=} E(n+1)(m)$. We can now combine the two conclusions, reaching (8.30).

1.	$\mathsf{T} \vdash \lambda l.M :_m E(0)(m)$	
2.	$E(n)(g) \vdash \lambda l.M :_m E(n+1)(m)$	
3.	$\mathsf{T} \vdash \mu g.\lambda l.M :_m \forall n.E(n)(m)$	(Rec)

We have now arrived at the assertion (8.30).

Inference Examples (3). Abstract Data Type. The final example is a short inference for packing and unpacking, using the counter ADT in (8.22). To recall, cntr is given by $\mathtt{pack}\ \langle \mathbb{N}, M \rangle\ \mathtt{as}\ \exists \mathrm{X}.\alpha$ where:

$$M \stackrel{\text{def}}{=} \langle \mathbb{N}, \{\mathtt{new} : \lambda().0,\ \ \mathtt{inc} : \lambda x.x+1,\ \ \mathtt{get} : \lambda x.x\} \rangle$$
$$\alpha \stackrel{\text{def}}{=} \{\mathtt{new} : \mathtt{Unit} \Rightarrow \mathrm{X},\ \ \mathtt{inc} : \mathrm{X} \Rightarrow \mathrm{X},\ \ \mathtt{get} : \mathrm{X} \Rightarrow \mathbb{N}\}.$$

Our aim is to prove, writing $N.\mathtt{op}(M)$ for $(N.\mathtt{op})(M)$:

$$N \stackrel{\text{def}}{=} \mathtt{unpack\ cntr\ as}\ \langle \mathrm{X}, x \rangle\ \mathtt{in}\ \lambda().x.\mathtt{get}(x.\mathtt{inc}(x.\mathtt{new}()))$$

Observe $\vdash N : \mathtt{Unit} \Rightarrow \mathbb{N}$. The assertion we wish to prove is very simple.

$$\mathsf{T} \vdash N :_u u \bullet () \geq 1.$$

We use the following specification of M, writing $e.\mathsf{op}(e')$ for $(e.\mathsf{op}) \bullet (e')$.

$$C(u, \mathrm{X}) \stackrel{\text{def}}{=} \exists w^{\alpha}, y^{\mathrm{X}}, z^{\mathrm{X}}.(u = \langle \mathrm{X}, w\rangle \wedge w.\mathtt{new}(\,) = y \wedge w.\mathtt{inc}(y) = z \wedge w.\mathtt{get}(z) \geq 1)$$

Note $C(\langle \mathrm{X}, x\rangle, \mathrm{X})$ entails these invocations indeed terminate. Also observe, by extensionality, we have e.g. $\forall y.(x.\mathtt{new}(\,) = y \wedge x.\mathtt{inc}(y) = z) \supset \exists z.\, x.\mathtt{get}(z) \geq 1)$.

The derivation follows. Below [*Record*] and [*Sel*] are the proof rules for record and field selection given at the end of Section 6.5.

1. $\mathsf{T} \vdash \lambda().0 :_{m_1} m_1 \bullet () = 0$	(Num, Abs)
2. $\mathsf{T} \vdash \lambda x.x + 1 :_{m_2} m_2 \bullet 0 = 1$	(Var, Add, Abs)
3. $\mathsf{T} \vdash \lambda x.x :_{m_3} m_3 \bullet 1 = 1$	(Var, Abs)
4. $\mathsf{T} \vdash M :_m m.\mathtt{new}(()) = 0 \wedge m.\mathtt{inc}(0) = 1 \wedge m.\mathtt{get}(1) = 1$	(1,2,3,Record)
5. $\mathsf{T} \vdash M :_m \exists y^{\mathbb{N}}, z^{\mathbb{N}}.\ (m.\mathtt{new}(()) = y \wedge m.\mathtt{inc}(0) = z \wedge m.\mathtt{get}(1) = 1)$	(Conseq)
6. $\mathsf{T} \vdash \mathtt{pack}\,\langle \mathbb{N}, M\rangle\ \mathtt{as}\ \exists \mathrm{X}.\alpha :_m \exists \mathrm{X}.\, C(m, \mathrm{X})$	(T-Pack)
7. $C(\langle \mathrm{X}, x\rangle, \mathrm{X}) \vdash x.\mathtt{get} :_m (x.\mathtt{new}(()) = y \wedge y.\mathtt{inc}(y) = z) \supset m \bullet z \geq 1$	(Var, Sel)
8. $C(\langle \mathrm{X}, x\rangle, \mathrm{X}) \vdash x.\mathtt{inc} :_m x.\mathtt{new}(()) = y \supset \exists z.m \bullet y = z$	(Var, Sel)
9. $C(\langle \mathrm{X}, x\rangle, \mathrm{X}) \vdash x.\mathtt{new} :_m \exists y.m \bullet () = y$	(Var, Sel)
10. $C(\langle \mathrm{X}, x\rangle, \mathrm{X}) \vdash x.\mathtt{new}() :_m \exists y.m.\mathtt{new}(()) = y$	(9, App)
11. $C(\langle \mathrm{X}, x\rangle, \mathrm{X}) \vdash x.\mathtt{inc}(x.\mathtt{new}()) :_m \exists y, z.(n.\mathtt{new}(()) = y \wedge m.\mathtt{inc}(()) = z)$	(8,10,App)
12. $C(\langle \mathrm{X}, x\rangle, \mathrm{X}) \vdash x.\mathtt{get}(x.\mathtt{inc}(x.\mathtt{new}())) :_m m \geq 1$	(7,11,App)
13. $C(\langle \mathrm{X}, x\rangle, \mathrm{X}) \vdash \lambda().x.\mathtt{get}(x.\mathtt{inc}(x.\mathtt{new}())) :_u u \bullet () \geq 1$	(Abs)
14. $\mathsf{T} \vdash N :_u u \bullet () \geq 1$	(T-Unpack)

In Line 5, turning concrete values to variables is essential to reach Line 6.

The reasoning above uses the standard decomposition of method invocations into label selection and application. The reasoning becomes much simpler if we use the proof rules which are directly about method invocation. The program, denoted cntr', is given as $\mathtt{pack}\,\langle \mathbb{N}, M'\rangle\ \mathtt{as}\ \exists \mathrm{X}.\alpha'$, where we set:

$$\begin{aligned} M' &\stackrel{\text{def}}{=} \langle \mathbb{N}, \{\mathtt{new}() : 0,\ \ \mathtt{inc}(x) : x + 1,\ \ \mathtt{get}(x) : x\}\rangle \\ \alpha' &\stackrel{\text{def}}{=} \{\mathtt{new} : \varepsilon \Rightarrow \mathrm{X},\ \ \mathtt{inc} : \mathrm{X} \Rightarrow \mathrm{X},\ \ \mathtt{get} : \mathrm{X} \Rightarrow \mathbb{N}\}. \end{aligned}$$

The client is now denoted N', which is the same as before except it uses cntr' instead of cntr, while its specification remains the same, even though the logical

$$[T\text{-}Abs]\ \frac{A^{\text{-X}} \vdash V :_m B[u^{\forall \mathrm{X}.\alpha}/m^{\alpha}]}{A \vdash \Lambda \mathrm{X}.V :_u \forall \mathrm{X}.B}$$

$$[T\text{-}App]\ \frac{A \vdash M :_m \forall \mathrm{X}.\, B[m^{\forall \mathrm{X}.\alpha}/u^{\alpha[\beta/\mathrm{X}]}]}{A \vdash M\beta :_u B[\beta/\mathrm{X}]}$$

$$[Pack]\ \frac{A \vdash V :_m B[\beta/\mathrm{X}][m^{\alpha[\beta/\mathrm{X}]}/u^{\exists \mathrm{X}.\alpha}]}{A \vdash \mathtt{pack}\ \langle \beta, V\rangle\ \mathtt{as}\ \exists \mathrm{X}.\alpha :_u \exists \mathrm{X}.B}$$

$$[Unpack]\ \frac{A^{\text{-}x\mathrm{X}} \vdash M :_m \exists \mathrm{X}.E \quad A \wedge E[x^{\alpha}/m^{\exists \mathrm{X}.\alpha}] \vdash N :_u B}{A \vdash \mathtt{unpack}\ M\ \mathtt{as}\ \langle \mathrm{X}, x\rangle\ \mathtt{in}\ N :_u B}$$

Fig. 27. Proof Rules for $\mathrm{PCF}_v^{\forall\exists\mu}$(with implicit type conversion in assertions)

language now has the intrinsic construct $e'.\mathtt{op}(\vec{e})$. The proof rules follow.

$$[Object]\frac{A \wedge E_i^{\vec{y}_i} \vdash M_i :_{m_i} B_i \quad \wedge_i \forall \vec{y}_i.(E_i \supset \wedge B_i[u.l_i(\vec{y}_i)/m_i]) \supset B}{A \vdash \{l_i(\vec{y}_i) : M_i\} :_u B}$$

$$[Inv_l]\ \frac{A \vdash M :_m B \quad A \vdash N_i :_{y_i} B_i \quad A \supset \wedge_i B_i \supset B[u.l(\vec{y}_i)/m]}{A \vdash M.l(\vec{N}) :_u B}$$

For further simplification, we take off the construct $e\bullet[\alpha]$ and $\langle \alpha, e\rangle^{\beta}$, and instead allow *implicit type conversion*, treating e of type $\exists \mathrm{X}.\alpha$ as an expression of type α, similarly for universal abstraction (this in fact forces existentially abstracted types to be treated opaquely, but practically this is all we need). We can now write the specification for the counter as:

$$C'(u^{\exists \mathrm{X}.\alpha}, \mathrm{X}) \ \stackrel{\text{def}}{=}\ \exists y^{\mathrm{X}}, z^{\mathrm{X}}.(\, u.\mathtt{new}(\,) = y \ \wedge\ u.\mathtt{inc}(y) = z \ \wedge\ u.\mathtt{get}(z) \geq 1\,)$$

The corresponding proof rules are given in Figure 25. To make clear the change of types (which is essentially all what these rules do), we explicitly annotate the anchor of each assertion with a type. Let us experiment with these rules using a simple example, taking the inference for the universal identity given before.

1.	$\mathsf{T} \vdash x :_m m^{\mathrm{X}} = x^{\mathrm{X}}$	(Var)
2.	$\mathsf{T} \vdash \lambda x^{\mathrm{X}}.x :_u \forall x^{\mathrm{X}}.u^{\mathrm{X}\Rightarrow\mathrm{X}} \bullet x = x$	(Abs)
3.	$\mathsf{T} \vdash \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x :_u \forall \mathrm{X}.\forall x^{\mathrm{X}}.u^{\forall \mathrm{X}.\mathrm{X}\Rightarrow\mathrm{X}} \bullet x = x$	(T-Abs)
4.	$\mathsf{T} \vdash \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x :_u \forall x^{\mathbb{B}}.u^{\forall \mathrm{X}.\mathrm{X}\Rightarrow\mathrm{X}} \bullet x = x$	(3. Conseq)
5.	$\mathsf{T} \vdash \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x :_u \forall x^{\mathbb{N}}.u^{\forall \mathrm{X}.\mathrm{X}\Rightarrow\mathrm{X}} \bullet x = x$	(3, Conseq)

The assertions reflect the type operations less rigorously, while giving (arguably) more natural specifications.

We now turn to the new derivation for the existential, which use both simplified rules for polymorphism and the rules for objects.

1.	$\mathsf{T} \vdash M' :_m \; m.\mathtt{new}(\varepsilon) = 0 \;\wedge\; m.\mathtt{inc}(0) = 1 \;\wedge\; m.\mathtt{get}(1) = 1$	(Num, Var, Add, Var, Object)
2.	$\mathsf{T} \vdash M' :_m \exists y^{\mathbb{N}}, z^{\mathbb{N}}.\, (m.\mathtt{new}(\varepsilon) = y \wedge m.\mathtt{inc}(0) = z \wedge m.\mathtt{get}(z) = 1)$	(Conseq)
3.	$\mathsf{T} \vdash \mathtt{pack}\,\langle \mathbb{N}, M\rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha :_m \; \exists \mathrm{X}.\, C'(m, \mathrm{X})$	(T-Pack)
4.	$C(x, \mathrm{X}) \vdash x.\mathtt{new}(\varepsilon) :_m \; x.\mathtt{new}(()) = m$	(Inv)
5.	$C(x, \mathrm{X}) \vdash x.\mathtt{inc}(x.\mathtt{new}(\varepsilon)) :_m$ $\exists y.\, (x.\mathtt{new}(()) = y \;\wedge\; x.\mathtt{inc}(y) = m)$	(Var, 4, Inv)
6.	$C'(x, \mathrm{X}) \vdash x.\mathtt{get}(x.\mathtt{inc}(x.\mathtt{new}(\varepsilon))) :_m \; m \geq 1$	(Var, 5, Inv)
7.	$C(x, \mathrm{X}) \vdash \lambda().x.\mathtt{get}(x.\mathtt{inc}(x.\mathtt{new}(\varepsilon))) :_u \; u \bullet () \geq 1$	(Abs)
8.	$\mathsf{T} \vdash N' :_u \; u \bullet () \geq 1$	(T-Unpack)

Note how the reasoning now only concerns essential dynamics programs would be involved in. While we have some reservations on the unbridled use of implicit type conversion, we do believe it is essential to have the proof rules which precisely correspond to the granularity of operations which a given (decent) programming language offers. The significance of the π-calculus and its typed logic in their use for developing, and giving the basis for, logics for programming languages lies in its power to faithfully and cleanly model diverse forms of the dynamics and algebras which (decent) programming languages offer, rather than imposing a particular kind of granularity, which does not sacrifice but in fact promote precision in their mathematical analysis. In the subsequent sections we shall examine how this feature of the π-calculus logic would be used in the domain of extremely expressive sequential behaviour which involves, in addition to higher-order computation we have studied so far, general state and locality.

9 Sequential Processes with Open State

9.1 State and Processes

The remaining sections of Part I is devoted to the study of logical articulation of sequential, stateful computation. The combination of sequentiality and state has a special meaning, both historically and practically. It is in this setting where Algol 60 was originally developed; it is also the setting where the first compositional program logic was originally developed. Practically, sequential stateful programs form the fundamental part of general computing systems, including concurrent, distributed ones. In the context of semantic studies, the richness and subtleties of stateful sequential computation have been the target of research over long years, especially because behaviours of programs in this class cannot be easily captured in the framework of functions. But in theory of processes, we can position this class of behaviour as a sub-class of general process behaviour. Can this positioning help us to give a clear logical articulation of behaviours belonging to this class? For that purpose we first need to clarify what we mean by state, and what interplay it would have with sequentiality in the context of processes.

Processes interact at channels. If a process can be invoked many times though a certain channel, then either it has the same interactive behaviour each time; or it can change behaviour from one invocation to another. As a familiar example, just consider procedures or objects regarded as interacting processes. Over a series of invocations, some would never change their behaviour while some may do so. In the former case, we say the process is **stateless**: in the latter, it is **stateful**. This distinction is meaningful both in concurrent computation and sequential one, since it only pertains to whether a channel which can be used many times exhibits the constant behaviour or not. Clearly stateless computation is a special case of stateful computation. Of the two, stateless computation is more tractable than stateful ones, since we can assume their properties are constant over time, the assumption not possible in the stateful setting. Indeed we have fully exploited this constancy in our study in the preceding sections: even under divergence and advanced type structure, the specification of process behaviour can always rely on the assumption that the behaviour of an environment is constant, how many times the process interacts with it. For example, if we once assume the environment at x returns an even number, then we can safely assume it will continue to do so how many times it is invoked at x, the assumption which is true in stateless computation but not so in the presence of state.

Stateful computation is represented in various ways in the π-calculus. Suggested by the preceding studies [5, 107], the present inquiry considers representation of a stateful behaviour as a combination of replications and a *reference agent*. The latter represents a generic behaviour of an imperative variable, defined recursively as follows:

$$\mathsf{Ref}\langle xv\rangle \;=\; x[\mathsf{read}(c).\overline{c}\langle v\rangle \,\&\, \mathsf{write}(wc).(\overline{c}|\mathsf{Ref}\langle xw\rangle)].$$

A reference waits with two branches at its subject: when the "read" branch is selected, it receives a name to which it sends the value it owns. If its "write" branch is selected receiving two names, a new value and a continuation; and, while acknowledging the receipt at the latter, will become a new cell with the received value. Note a reference itself is characterised by simple name passing interaction: but when combined with replicated processes, this behaviour can generate a large class of stateful behaviours.

For our inquiries, the combination of references and replication gives a particularly convenient setting, since in this framework we can focus our attention on references for a logical analysis of state: "state change" now simply means "change of a value (name) stored in a reference". In the sequential setting (in the sense that there are never more than one thread in a configuration), we have further simplification: if a process changes the content of a reference, then it can only be observed *after* that process finishes its own activity — since, if not, we are to have more than one threads, contrary to our assumption of sequentiality. As a simple example, in the following configuration:

$$(\boldsymbol{\nu}\, c)(\overline{x}\mathsf{write}\langle wc\rangle | c.\overline{c'}) \mid \mathsf{Ref}\langle xv\rangle \mid c'.\overline{x}\mathsf{read}(f)f(y).P$$

the first thread writes to a reference, and ends its execution by signalling at c': then the next thread starts execution, reads from the same variable and continues its execution at P. As far as we assume sequentiality, the first write never overlaps with another write/read at the same variable. Thus the effect a sequential process has on, or what it guarantees to, the environment, can only be perceived by another sequential process *after* its execution: in turn, a process would observe (get affected by, rely on) the effect of another process which took place *before* the former's execution. Thus we arrive at the fundamental insight of Floyed-Hoare's method for the logical analysis of sequential stateful computation:

> *For capturing a stateful sequential behaviour, it is enough to specify how it changes the values of references, which can be written down as a predicate on the initial state (pre-condition) and a predicate on the final state (post-condition); two such specifications can then be composed by checking the first behaviour's post-condition entails that of the second behaviour, leading to the specification of the composite behaviour.*

In the context of the typed process logic, this means we should augment the original sequent $P^{\Gamma} \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B$ with two predicates on references, arriving at the sequent of the form:

$$P^{\Gamma} \vdash \{C\}\, \mathbf{rely}\, A\, \mathbf{guar}\, B\, \{C'\}. \tag{9.1}$$

Following the standard terminology, we call C and C' *pre-condition* and *post-condition.* The sequent is read: *Under the stateless behaviour of the environment specified by A and the state of references specified by C, P will behave as B, as well as changing, when terminated, the current state of references to C.* Note we retain the rely and guarantee formulae in the above sequent, which are used for specifications on stateless behaviour (which includes replicated and linear behaviours). Further we shall still use the assertions of the same shape as before:

$$P^{\Gamma} \vdash \textbf{rely}\, A\, \textbf{guar}\, B. \tag{9.2}$$

This distinct use of two forms of judgements is strictly guided by types: the sequent in (9.1) is used when P (hence Γ) contains a linear output, while the original form (9.2) is used when a process does not. As we shall see later, this distinction is essential for a consistent logical articulation of stateful behaviours, both in the process logic and its applications.

When state is exposed to the outside, the behaviour of a replicated process which uses it can be analysed away as a function of some sort, since its behaviour is completely determined by a given (global) state of references as well as other replicated processes. If however a state is hidden, the behaviour of a replicated process cannot be analysed in this way. As an example, consider the following process with a replicated channel u as its interface:

$$(\boldsymbol{\nu}\, x)(!u(v'z).(\boldsymbol{\nu}\, cc')(\overline{x}\mathsf{read}(c)c(w).\overline{x}\mathsf{write}\langle v'c'\rangle c'.\overline{z}\langle w\rangle) \mid \mathsf{Ref}\langle xv\rangle) \tag{9.3}$$

Here the state is hidden, or local: each time this process is invoked at u, its external behaviour — what it outputs at z — changes. On the other hand, if we take off the hiding, state open:

$$!u(v'z).(\boldsymbol{\nu}\, cc')(\overline{x}\mathsf{read}(c)c(w).\overline{x}\mathsf{write}\langle v'c'\rangle c'.\overline{z}\langle w\rangle) \mid \mathsf{Ref}\langle xv\rangle \tag{9.4}$$

the behaviour is completely transparent, and can be analysed as a function which is determined by the content of a reference agent together with its input argument. This suggests there is a basic distinction between stateful behaviours with open state and those with local state, which in fact follows the insight gained by a long semantic study of expressiveness and subtleties of stateful computation with local state. This insight we shall confirm in the following technical development, presenting a clean stratification of logics for sequential stateful behaviours based on the distinction between local state and global state.

In the remainder, Section 9 (this section) discusses the affine total logic for open stateful processes. Section 10 shows applications of the stateful process logic to program logics. We show two derived program logics:

- A total logic for PCF_v extended with global imperative variables.
- An straightforward extension of Hoare's logic with (named) higher-order procedures.

In both cases, we treat variables which can store arbitrary higher-order programs. An example inference is shown which demonstrates a non-trivial usage of a higher-order store. Section 11 adds local state (hiding of references) to the stateful process logic. This small addition results in the fundamental change in structure of behaviour, since now the behaviour is not functional but depends on hidden, local state of processes. We show how this complexity is cleanly and generally dealt with in the typed process logic. The resulting logic can be used, possibly in combination with advanced type structure studied in Section 8, for deriving program logics for a wide range of programming languages and data structures, out of which Section 12 presents a few basic instances.

$$\frac{\begin{array}{l}(\text{Weak-}?_{rw})\\ \vdash P \triangleright \Gamma, z:\rho^{\uparrow}\end{array}}{\vdash P \triangleright \Gamma, z:\rho^{\uparrow}, x:\tau^{?_{rw}}} \qquad \frac{\begin{array}{l}(\mathsf{In}^{!})\\ \vdash P \triangleright ?\Gamma^{\text{-}x}, \Upsilon^{\text{-}x}, \vec{y}z:\vec{\tau}\rho\end{array}}{\vdash !x(\vec{y}z).P \triangleright \Gamma, x:(\vec{\tau}\rho)^{!}_{\Upsilon}} \qquad \frac{\begin{array}{l}(\mathsf{Out}^{?})\ \ (\Gamma(x)=(\vec{\tau}\rho)^{?}_{\Upsilon'},\ \Upsilon\subset\Upsilon')\\ \vdash P \triangleright ?\Gamma, \Upsilon, \vec{y}:\vec{\tau}, z:\rho^{\downarrow}, v:\sigma^{\uparrow}\end{array}}{\vdash \overline{x}(\vec{y}z)P \triangleright \Gamma, \Upsilon, v:\sigma}$$

$$\frac{\begin{array}{l}(\mathsf{Ref})\quad (x\notin \mathsf{fn}(\tau))\\ -\end{array}}{\vdash \mathsf{Ref}\langle xy\rangle \triangleright x:\mathit{ref}(\tau), y:\overline{\tau}} \qquad \frac{\begin{array}{l}(\mathsf{Read})\quad (\Gamma(x)=\overline{\mathit{ref}}(\tau))\\ \vdash P \triangleright \uparrow ? ?_{rw}\Gamma, c:(\overline{\tau})^{\downarrow}\end{array}}{\vdash \overline{x}\mathsf{read}(c)P \triangleright \Gamma} \qquad \frac{\begin{array}{l}(\mathsf{Write})\quad (\Gamma(x)=\overline{\mathit{ref}}(\tau))\\ \vdash P \triangleright \uparrow ? ?_{rw}\Gamma, v:\tau, c:()^{\downarrow}\end{array}}{\vdash \overline{x}\mathsf{write}(vc)P \triangleright \Gamma}$$

Fig. 28. Typing Rules for Open State (revised/added rules)

9.2 Typing Open Stateful Behaviour

As we already discussed, the only change in syntax of processes is to add a reference agent. For processes to interact with this agent, we also need reading and writing actions.

$$P \quad ::= \quad ... \mid \mathsf{Ref}\langle xy\rangle \mid \overline{x}\mathsf{read}(c)P \mid \overline{x}\mathsf{write}(yc)P$$

Read and write actions are in fact best seen as instances of $?$-mode selection actions, though we use the above more readable symbols. The reduction rules are naturally given as follows:

$$\begin{array}{rcl}\mathsf{Ref}\langle xy\rangle|\overline{x}\mathsf{read}(c)P & \longrightarrow & \mathsf{Ref}\langle xy\rangle|(\boldsymbol{\nu}\, c)(\overline{c}\langle y\rangle|P)\\ \mathsf{Ref}\langle xy\rangle|\overline{x}\mathsf{write}(y'c)P & \longrightarrow & (\boldsymbol{\nu}\, y')(\mathsf{Ref}\langle xy'\rangle|\overline{c}|P)\end{array}$$

In the write rule, $\overline{c}$ is the acknowledgement P would receive.

We next stipulate the types and typing for open state, which, as before, give a basis and guiding principle for the corresponding process logic. The following channel types are common to both open and local state.

$$\begin{array}{rcl}\tau & ::= & ... \mid (\vec{\tau}^{?}\rho^{\uparrow})^{!}_{\Upsilon} \mid (\vec{\tau}^{!}\rho^{\downarrow})^{?}_{\Upsilon}\\ \delta & ::= & \mathit{ref}(\tau^{!})^{!_{rw}} \mid \overline{\mathit{ref}}(\tau^{?})^{?_{rw}}\end{array}$$

where Υ in the first line is a finite map from names to types of the form $\overline{\mathit{ref}}(\tau^{!})$. Υ in $(\vec{\tau}^{?}\rho^{\uparrow})^{!}_{\Upsilon}$ and $(\vec{\tau}^{!}\rho^{\downarrow})^{?}_{\Upsilon}$ is called *effect*. When $\Upsilon=\emptyset$, we identify $(\vec{\tau}^{?}\rho^{\uparrow})^{!}_{\Upsilon}$ with $(\vec{\tau}^{?}\rho^{\uparrow})^{!}$, similarly for $(\vec{\tau}^{!}\rho^{\downarrow})^{?}_{\Upsilon}$. $\mathit{ref}(\tau)$ is a type of a reference whose stored value has type τ, while $\overline{\mathit{ref}}(\overline{\tau})$ is its dual. A basic aspect of these channel types for open state is that a channel type never carries a (co-)reference type, except indirectly as an effect of a carried type. Communication of reference names leads to a different level of logical complexity, discussed in Section 11.

$$\frac{(\mathsf{Hide\text{-}global}) \quad (x \notin \mathsf{fn}(\Gamma)) \\ \vdash P \triangleright \Gamma, x : \mathit{ref}(\tau)}{\vdash (\boldsymbol{\nu}\, x)P \triangleright \Gamma}$$

Fig. 29. Typing Rule of Reference Restriction (for open state)

As before, an *action type* $(\Gamma, \Delta, \ldots)$ is a finite map from names to channel types. The polarisation $\Gamma; \Delta$ now involves reference types, for which we assume co-reference types should occur in Γ and reference types in Δ. Among action types, $\Upsilon, \Upsilon', \ldots$ exclusively range over those which have only co-reference types in their ranges. τ (resp. δ, resp. Γ) is *well-formed* if (it conforms to the grammar above and) all occurring effects in it are subsets of a effect, or, equivalently, the union of occurring effects is again an effect. *From now on we only consider well-formed types and action types.*

The typing rules are listed in Figure 28. We add the rules for references and two actions on them. (In) and (Out) now incorporate effects, subsuming the respective corresponding rules in Figure 1. We also add the weakening of co-reference (read/write) channels. By its side condition, the only case when $?_{rw}$-types occur directly in an action type is when a process contains a linear output, which does not lose expressiveness while leading to a cleaner logical articulation. We use the same (Res)-rule as before for non-reference channels. In addition, we have the rule for hiding a reference name in Figure 29, whose side condition ensures that the hidden reference name is never used as an effect name: indeed, if it does, then a replicated channel is going to have a local state, which we do not want in open stateful typing.

9.3 Examples of Stateful Processes

Let us present a couple of examples of typable processes. First, any typable processes in the purely functional affine π-calculus are also typable in the present system, with an identical action type. For example, the following term is typable in open stateful typing (recall $\mathbb{N}^\circ \stackrel{\text{def}}{=} ([\oplus_{\mathbb{N}}]^\uparrow)^!$):

$$\vdash \overline{y}(c)c[\&_n \overline{z}\mathtt{in}_{2\times n}] \triangleright y : \overline{\mathbb{N}^\circ},\, z : [\oplus_{i\in\mathbb{N}}]^\uparrow \tag{9.5}$$

In addition, we can also type a stateful process, such as follows. Below we remember $[\![n]\!]_u \stackrel{\text{def}}{=} !u(c).\overline{c}\mathtt{in}_n$ and let $\Upsilon \stackrel{\text{def}}{=} y : \overline{\mathit{ref}}(\overline{\mathbb{N}^\circ}),\, z : \overline{\mathit{ref}}(\overline{\mathbb{N}^\circ})$.

$$\vdash \overline{y}\mathsf{read}(c)c(c')\overline{c'}(e)e[\&_n \overline{z}\mathsf{write}(eg)([\![n \times 2]\!]_e | g.\overline{w})] \triangleright \Upsilon,\, w : ()^\uparrow \tag{9.6}$$

which may be regarded as a stateful version of (9.5).

Next, by prefixing (9.6) with replication, we obtain another typed term.

$$\vdash !u(w).\overline{y}\mathsf{read}(c)c(c')\overline{c'}(e)e[\&_n \overline{z}\mathsf{write}(eg)([\![n \times 2]\!]_e | g.\overline{w})] \triangleright u : (()^\uparrow)^!_\Upsilon \tag{9.7}$$

where all Υ is taken off which is now used to annotate the type of u. We can further abstract u:

$$\vdash \overline{x}(u)!u(w).\overline{y}\mathsf{read}(c)c(c')\overline{c'}(e)e[\&_n\overline{z}\mathsf{write}(fg)([\![n\times 2]\!]_f|g.\overline{w})] \triangleright x:((()^{\uparrow})^{!}_{\Upsilon})^{\uparrow} \quad (9.8)$$

where an effect-annotated type is carried inside another type.

As another example, the following is a process which may get composed with the process in (9.7) and invokes it:

$$\vdash \overline{u}(w)w.\overline{l} \triangleright u:\overline{(()^{\uparrow})^{!}_{\Upsilon}},\ \Upsilon \tag{9.9}$$

Note Υ not only annotates the co-replicated type of u, but also occurs in the action type, due to the side condition on the refined (Out) in Figure 1. Indeed, because the process in (9.9) invokes at u and because the type of u says this may cause an effect at Υ, we may as well record it as an effect of this invoker itself.

A reference in the present system cannot be hidden, but can store a complex behaviour. For example the following is a process which uses the behaviour as given in (9.5), assuming it is stored in a reference. Let $\tau \stackrel{\text{def}}{=} (\mathbb{N}^{\circ}[\&_{i\in\mathbb{N}}])^{?}$.

$$\vdash \overline{x}\mathsf{read}(c)c(e).\overline{e}(yw)([\![3]\!]_y|w[\&_n\overline{z}\mathtt{in}_n]) \triangleright x:\overline{\mathit{ref}}(\tau),\ z:[\oplus_{i\in\mathbb{N}}]^{\uparrow}. \tag{9.10}$$

A dual behaviour can be written as:

$$\vdash \mathsf{Ref}\langle xv\rangle \mid !v(yw).\overline{y}(c)c[\&_n\overline{w}\mathtt{in}_{2\times n}] \triangleright x:\mathit{ref}((\overline{\mathbb{N}^{\circ}}[\oplus_{i\in\mathbb{N}}])^{!}),\ v:(\overline{\mathbb{N}^{\circ}}[\oplus_{i\in\mathbb{N}}])^{!} \tag{9.11}$$

Note the stored value can also be a replicated process with effects.

Finally we show a replicated process can simulate a reference. We use the free output notation from §3.7 (e.g. $\overline{x}\langle v\rangle$ becomes $\overline{x}(v')[v'\to v]$). Let $\delta \stackrel{\text{def}}{=} \mathit{ref}(\tau)$.

$$\vdash !r(c).\overline{x}\mathsf{read}\langle c\rangle \mid !w(wc).\overline{x}\mathsf{write}\langle wc\rangle \triangleright r:((\tau)^{\uparrow})^{!}_{x:\overline{\delta}},\ w:(\overline{\tau}()^{\uparrow})^{!}_{x:\overline{\delta}} \tag{9.12}$$

Invoking the process in (9.12) at r is the same thing as reading from x, while invoking at w is the same thing as writing at x. Note also the types at r and w do record they access x. Thus, disallowing reference types to be carried in types does not lead to a loss of basic expressive power. The construction is closely related with the emulation of a reference with a product.

Finally we consider how a distinction between an open state and a local state arise via (un)typability in the present system, using processes which appeared in (9.3) and (9.4). Let $P \stackrel{\text{def}}{=} !u(v'z).(\boldsymbol{\nu}\, cc')(\overline{x}\mathsf{read}(c)c(w).\overline{x}\mathsf{write}\langle v'c'\rangle c'.\overline{z}\langle w\rangle)$. Then we have, for an appropriate $\tau^{!}$:

$$\vdash P \triangleright u : (\overline{\tau}(\tau)^{\uparrow})^{!}_{x:\mathit{ref}(\tau)} \tag{9.13}$$

Note the effect mentions x. Then the following is not typable:

$$(\boldsymbol{\nu}\, x)(P \mid \mathsf{Ref}\langle xv\rangle) \tag{9.14}$$

since the action type of P still mentions x as its effect. On the other hand, we can safely type:

$$(\boldsymbol{\nu}\, x)(\overline{x}\mathsf{read}(c)c(w).\overline{z}\langle w\rangle \mid \mathsf{Ref}\langle xv\rangle) \tag{9.15}$$

noting the type of the process $\overline{x}\mathsf{read}(c)c(w).\overline{z}\langle w\rangle$ is $x : \overline{ref}(\overline{\tau}), z : (\tau)^{\uparrow}$ (in fact the behaviour (9.15) is realisable without a reference, so this is also natural semantically).

9.4 Assertions for Sequential Processes with Open State

Basic Ideas. Since state of a reference is completely characterised what it stores, we only need a notation to dereference a reference channel for describing its properties. The dereference of x (of a reference type) is written $!x$. Using this operation, we can write a specification for the typed process (9.6). For reference we first reproduce the process:

$$P_1 \stackrel{\text{def}}{=} \overline{y}\mathsf{read}(c)c(c')\overline{c'}(e)e[\&_n \overline{z}\mathsf{write}(eg)([\![n\times 2]\!]_e | g.\overline{w})]$$

which has the type $\vdash P_1 \triangleright \varUpsilon,\, w : ()^{\uparrow}$ with $\varUpsilon \stackrel{\text{def}}{=} y : \overline{ref}(\overline{\mathbb{N}^{\circ}}),\, z : \overline{ref}(\overline{\mathbb{N}^{\circ}})$. The specification for P_1 follows, recalling $\underline{a \text{ is } n}$ stands for $a \bullet \varepsilon = n$.

$$P_1 \vdash \{\, \underline{!y \text{ is } 3} \,\}\, \textbf{rely}\; \mathsf{T}\; \textbf{guar}\; w = ()^{\uparrow}\; \{\, \underline{!z \text{ is } 6} \,\} \tag{9.16}$$

which says that, if a reference y has content 3 initially, then running P_1 would ensure a linear output at w and the value of z to be 6. Note we can add $\underline{!y \text{ is } 3}$ to the post-condition, saying it does not change the value of y.

Next we consider invocation of a stateful replication, i.e. a replication which indirectly reads/writes references. In the stateless process logic, we have the operation $a \bullet \vec{b}$, which we shall continue to use when the type of a mentions the empty effect. However if the type of an invoked channel does mention an effect, say $\varUpsilon$, its invocation may lead not only to a linear answer but also some writes at $\varUpsilon$; further these actions may depend on some reads from $\varUpsilon$. These observations lead to the following refinement of $a \bullet \vec{b}$, called *stateful invocation.*

$$\{C\}[a \bullet \vec{b}/z]A\{C'\} \tag{9.17}$$

The predicate[6] [7] says that, when a is invoked with arguments $\vec{b}$ at the initial state C, it terminates with the resulting state C' and the resulting linear output, named z, which satisfies A. For brevity we also write:

$$\{C\}a \bullet \vec{b} \rightsquigarrow a'\{C'\} \quad\equiv\quad \{C\}[a \bullet \vec{b}/z](z = a')\{C'\} \tag{9.18}$$

$$\{C\}a \bullet \vec{b}\{C'\} \quad\equiv\quad \{C\}[a \bullet \vec{b}/z]\mathsf{T}\{C'\} \tag{9.19}$$

[6] The stateful invocation, like a partial substitution (§5.2), is not a standard first-order formula. It is best seen as a typed reconstruction of formulae in the Hennessy-Milner logic for the π-calculus [85] such as $\langle a(\vec{b}c)\rangle(\langle\overline{c}(z)\rangle A \;\wedge\; B)$ where B describes the interactions with references.

[7] An expression $\{C\}a \bullet \vec{b}$ does not make sense as a term, since the resulting linear output may not be determined uniquely due to multiple states C may denote (if we had the set inclusion in our logical language we could have used it).

where, in both cases, z on the left-hand side is chosen fresh.

Using the abbreviation (9.18), we can concisely write down a specification for the process in (9.7), which we call P_2 ($\stackrel{\text{def}}{=} !u(g).P_1$). Below we let $\tau = (()^{\uparrow})^{!}_{\Upsilon}$ with $\Upsilon = y : \overline{ref}(\mathbb{N}^{\circ}), z : \overline{ref}(\mathbb{N}^{\circ})$.

$$P_2 \vdash \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ \forall i^{\mathsf{Nat}}. \{\, \underline{!y \text{ is } i} \,\}\ u^{\tau} \bullet \varepsilon \rightsquigarrow ()^{\uparrow}\ \{\, \underline{!z \text{ is } 2 \times i} \,\} \tag{9.20}$$

The specification says that P_2 will, when invoked in the state that y has value n, will ensure the value of a reference z to be its double, as well as giving a linear output as an answer to the invocation. Note the typing of the internalised pre/post conditions in (9.16) conforms to Υ. Note also the sequent itself does not have pre/post conditions: in fact, P_2 will be invoked many times, with distinct behaviours depending on the state at that moment.

As another example, let us see how these constructs can handle higher-order behaviours stored in a reference, taking the process in (9.10), which we reproduce below for reference.

$$P_3 \stackrel{\text{def}}{=} \overline{x}\mathsf{read}(c)c(e).\overline{e}(yw)([\![3]\!]_y | w[\&_n \overline{z}\mathtt{in}_n])$$

which is typed as $\vdash P_3 \triangleright x : \overline{ref}(\tau),\ z : [\oplus_{i \in \mathbb{N}}]^{\uparrow}$. The following is a possible specification for this agent.

$$P_3 \vdash \{\forall i^{\mathbb{N}}, w. (\, \underline{w \text{ is } i}\ \supset !y \bullet w = 2 \times i)\}\ \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ z = \mathtt{in}_6()^{\uparrow}\ \{\mathsf{T}\} \tag{9.21}$$

The assertion says that, if y stores a behaviour which returns the double of a given argument, then the process ensures a linear output of 6 at z.

Formal Syntax. We summarise the syntax of terms, formulae and assertions below. For terms and formulae, we augment their grammars as follows (on the basis of those given in §3.2).

$$a\ ::=\ ...\ |\ !x^{ref(\tau)} \qquad A\ ::=\ ...\ |\ \{C\}[a \bullet \vec{b}/z]A\{C'\}$$

$\mathsf{fn}(a)$ and $\mathsf{fn}(A)$ now include effect names occurring in types, starting from $\mathsf{fn}(x^{\tau}) = \{x\} \cup \mathsf{fn}(\tau)$ where $\mathsf{fn}(\tau)$ may include effect names when τ is replicated. The typing of these expressions is given as follows.

- $!x^{ref(\tau)}$ has type τ.
- In $\{C\}[a \bullet \vec{b}/z]A\{C'\}$, if a has type $(\vec{\tau}\rho)^{!}_{\Upsilon}$, then: (1) C and C' should be typed under $\Upsilon \cdot \Upsilon'$ as well as a typing for auxiliary names, where Υ and Υ' are mutually disjoint; (2) $\vec{b}$ should be pointwise typed by $\vec{\rho}$; and (3) A should be typed under the global typing together with a typing of z to ρ.

For judgements, we first need the classification of typed processes. A process with a free linear output is called **thread**. Typewise such a process can be written $\vdash P \triangleright \Gamma \cdot x : \tau^{\uparrow}$. A process without a free linear output is called **non-thread**. Typewise P is a non-thread iff $\vdash P \triangleright \Gamma$ such that $\mathsf{md}(\Gamma) \subset \{!, ?, !_{rw}, ?_{rw}\}$. The key difference between threads and non-threads is that, due to the sequential type discipline, either a thread always has reduction until it terminates (in which case it has a unique active free linear output) or, alternatively, such computation is suppressed by a linear input; while a non-thread never has reduction, and only starts computation when it is invoked. Thus a thread means there can be a passage of time, while a non-thread means there cannot be. based on these observations, we use the following sequent for threads.

$$P^{\Gamma;\Delta} \models \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}$$

for which we require, assuming $\Gamma = \Upsilon \cdot \mathbf{?}\!\downarrow\!\Gamma_1$ and Θ for auxiliary names:

1. C and C' are typed by $\Gamma_0 \cdot \Theta$,
2. A is typed by $\Gamma_1 \cdot \Theta$, and
3. B is typed by $\Delta \cdot \Theta$.

As before, we always assume all occurring judgements (including these sequents) are well-typed. For non-threads, we use the same sequent as used in the linear/affine process logics, that is:

$$P^{\Gamma;\Delta} \models \mathbf{rely}\,A\,\mathbf{guar}\,B$$

with the same type constraint as before.

9.5 Semantics of Assertions

Models and Interpretation. Let $\cong$ be the maximum contextual congruence with observables at $x : ()^{\uparrow}, \mathbf{!}_{rw}\Gamma$ (we need Γ since computation may use interaction with references). *Abstract processes* $(p, q, \ldots)$ are then defined just as before.[8] As before, a *model* ξ is a well-typed finite map from typed names to abstract processes. Such notations as ξ^{Γ} are also defined as before. For interpretation of terms and formulae, we need the following notions.

- We say r *stores* p iff $(\boldsymbol{\nu}\, v)(\mathsf{Ref}\langle xv\rangle | P) \in r(x)$ s.t. $P \in p(v)$ with x, v fresh.
- Let $p^{x:(\vec{\rho}\sigma)^!_{\Upsilon}}$ with $\Upsilon = \vec{w} : \vec{\overline{\delta}}$. Then $\langle p, \vec{q}, \vec{r}^{\vec{w}}\rangle \Downarrow \langle s, \vec{r'}^{\vec{w}}\rangle$ (with $\vec{q}$ of type $\vec{\rho}$ and $\vec{r}$ of type $\vec{\delta}$) iff, for each $P \in p(x)$, $Q_i \in q_i(y_i)$ and $R_j \in r_j(w_j)$, we have:

  $$P \,|\, (\boldsymbol{\nu}\, \vec{y})(\overline{x}\langle \vec{y}z'\rangle \,|\, \Pi_i Q_i) \,|\, \Pi_j R_j \;\Downarrow\; P \,|\, \overline{z}(c)S \,|\, \Pi_j R'_j$$

  where $S \in s(c)$ and $R'_j \in r'_j(w_i)$.

In the first clause, it is easy to check that each r stores a unique p, while p (of a replicated type) determines a unique r that stores p. In the second clause, $\langle p, \vec{q}, \vec{r}^{\vec{w}}\rangle \Downarrow \langle s, \vec{r'}\rangle$ intuitively says that a replication p will, when invoked with arguments $\vec{q}$ in the presence of references $\vec{r}^{\vec{w}}$, converges with an answer of value s and new state $\vec{r'}$. As before, the choice of concrete processes and names is insignificant in determining the relation.

[8] To be precise, when defining an abstract process of type τ, the injective renaming should avoid effect names in τ (or alternatively we let renaming act on τ itself).

Satisfaction relation. For interpretation of terms, we observe two references in distinct locations cannot be identified even if their current stored values coincide. In fact, even if x and y both initially store 1, if one writes 3 on x, they start to behave quite differently. This observation suggests we cannot treat names standing for stateful processes as standard logical variables (see [52, 86]: this point will be discussed in detail in Section 11 where we introduce the logic with local state), leading to the following interpretation of the added terms.

$$
\begin{array}{lcll}
[\![x]\!]_{\mathcal{I}\cdot\xi} & \stackrel{\text{def}}{=} & x & \\
[\![!x]\!]_{\mathcal{I}\cdot\xi} & \stackrel{\text{def}}{=} & p & \quad ([\![x]\!]_{\mathcal{I}\cdot\xi} \text{ stores } p)
\end{array}
$$

Note the first clause means an equation $x^{ref(\tau)} = y^{ref(\tau)}$ is valid only when x and y are literally identical. This makes sense since if their *content* is identical we can write $!x = !y$: so $x = y$ should mean they are equal as names. In the terminology of [84], we are assuming the identity distinction (i.e. two names are equal iff they are literally identical). Assuming the identity distinction also means that having auxiliary names typed with reference types may not be meaningful (since it does not add any expressiveness). However it is sometimes convenient to have reference names among auxiliary names is meaningful. One of them is discussed in Section 10.5 later.

For the satisfaction relation, we augment the original definition for refined invocation. In the following we assume a has type $(\vec{\rho}\sigma)^{!}_{\Upsilon}$ with $\mathsf{dom}(\Upsilon) = \{\vec{w}\}$ and $\mathsf{dom}(\Upsilon') = \{\vec{v}\}$ such that $\vec{w}$ and $\vec{v}$ are mutually disjoint, as well as well-typedness including the typability of C and C' under, as well as auxiliary names, $\Upsilon \cdot \Upsilon'$.

$$
\begin{array}{c}
\xi \models^{\mathcal{I}} \{C\}[a \bullet \vec{b}/z]A\{C'\} \\
\equiv \\
\vec{w}\!:\!\vec{r}, \vec{v}\!:\!\vec{s} \models^{\mathcal{I}} C \;\supset\; \exists q\vec{r'}.\,(\,\langle [\![a]\!]_{\mathcal{I}\cdot\xi}, [\![\vec{b}]\!]_{\mathcal{I}\cdot\xi}, \vec{r}^{\vec{w}}\rangle \Downarrow \langle q, \vec{r'}\rangle \;\wedge\; \xi, \vec{w}\!:\!\vec{r'}, \vec{v}\!:\!\vec{s}, z\!:\!q \models^{\mathcal{I}} A \wedge C\,)
\end{array}
$$

In the second line, A is interpreted by ξ together with the new value for z, not using $\vec{w} : \vec{r'}$: the scope of a hypothetical state (C and C') is restricted to the triple $\{C\}a \bullet \vec{b}\{C'\}$ (which is consistent with the effect typing). Note also $q \neq \omega$ is assumed in the third line.

Remark 7. (totality in stateful invocation) We may allow stateful invocation to result in divergence, just as in the total PCF_v-logic. While this extension does have some advantage (in particular the extensionality axiom is better presented with the general invocation), this section considers only the total version, because of its compelling naturalness in specifications and because we can represent the generalised version of the invocation using the total one.

Finally $\forall x^{ref(\tau)}.A$ and $\xi \models^{\mathcal{I}} \exists x^{ref(\tau)}.A$ should be stipulated by letting the quantified name x to range over the set of names of the same type under a given effect typing (so $\forall x^{ref(\mathbb{N})}.!x = 1$ under Υ means all $ref(\mathbb{N})$-typed names in Υ store 1). This framework is natural and consistent with the open stateful typing (since all reference behaviours are exposed): when we introduce reference creation/passing we shall consider a more refined framework.

Validity of Judgement. We can now introduce the semantics of logical judgements. As before, $P \models^{\mathcal{I}} A$ stands for $P \models_{\mathsf{b}} \xi$ and $\xi \cdot I \models A$ for some ξ.

Definition 9. (semantics of assertions)

1. *Let $P^{\overline{\Gamma};\Delta}$ be a non-thread. Then we set:*

$$P^{\overline{\Gamma};\Delta} \models^{I} \mathbf{rely}\, A\, \mathbf{guar}\, B \quad \equiv \quad R^{\Gamma} \models^{\mathcal{I}} A \supset (\boldsymbol{\nu}\, \mathsf{fn}(\Gamma))(P|R) \models^{\mathcal{I}} B.$$

2. *Let $P^{\overline{\Gamma};\Delta}$ be a thread with $\overline{\Gamma} = \Upsilon \cdot \mathbf{?}\!\downarrow\!\Theta$. Then we set:*

$$P^{\overline{\Gamma};\Delta} \models^{\mathcal{I}} \{C\} \mathbf{rely} A\, \mathbf{guar} B\, \{C'\} \;\equiv\; R^{\Gamma} \models^{\mathcal{I}} C \wedge A \supset (\boldsymbol{\nu}\, \mathsf{fn}(\Theta))(P|R) \models^{\mathcal{I}} B \wedge C'.$$

In Clause 2, we do not hide names in Υ. By generalising or by refining (syntax of formulae and) models, the same clauses give semantics for generalised total correctness and partial correctness in the sense of Section 5.

9.6 Proof Rules

The proof rules are given in Figure 30 (for composition rules) and Figure 31 (for prefix rules), which are added to: (Zero), (Res), (Weak) from Figure 1 and the last two's obvious counterparts for threads; (Consequence), (Aux-$\forall$) and (Aux-$\exists$) from Figure 3 and their counterpart for threads; and (Rec) from Figure 5 (note recursion is never over threads). We always assume processes and sequents are well-typed, including in conclusions. In Figure 30, there are three (Par) rules, following the refined typing for parallel composition, discussed in Appendix C.

- (Par-$\Downarrow\!\Uparrow$) is for composing two threads to obtain another thread. Note the given typings mean these processes do not own free replicated channels (such threads are called *pure*).
- (Par-$\mathbf{!?}$) is for composing two non-threads to obtain another non-thread. The rule has the same form as the original (Par) rule.
- (Par-$\mathbf{!!}_{rw}\mathbf{??}_{rw}$) is for composing a non-thread and a pure thread, to obtain a non-pure thread.

For more discussions on the decomposed (Par) rules, see Appendix C. (Hide-Global) says that a name associated with a hidden reference does not play any rôle in specification. Note the hidden state has only an ephemeral usage because the hidden reference channel is never used as an effect name due to the underlying type discipline: indeed, the same observable behaviour is always realisable without using a hidden reference.

Among the rules in Figure 31, (Bra) simply adapts the shape of sequents to the original rules. In (FSel) we give the rule for free linear selection, which is more convenient when we extend the logic with local state.[9] The following

[9] The free output rule becomes essential if we are to incorporate literal equality comparison of names. As discussed in Remark 1, Page 35, these rules are best used using negative primary names in the guarantee formulae.

(Par-$\downarrow\uparrow$)

$$\frac{P_1^{\Gamma;x:\tau^{\uparrow}} \vdash \{C\}\,\mathbf{rely}\ A_1\ \mathbf{guar}\ E\,\{C''\} \qquad P_2^{\Gamma\cdot x:\overline{\tau};\Delta} \vdash \{C''\}\,\mathbf{rely}\ A_2 \wedge E\ \mathbf{guar}\ B\,\{C'\}}{P_1|P_2{}^{\Gamma;\Delta} \vdash \{C\}\,\mathbf{rely}\ A_1 \wedge A_2\ \mathbf{guar}\ B\,\{C'\}}$$

(Par-!?) $(\Delta = \Delta_1, \Delta_2)$

$$\frac{P_1^{\Gamma;\Delta_1,\Theta} \vdash \mathbf{rely}\ A_1\ \mathbf{guar}\ B_1 \wedge E \qquad P_2^{\Gamma,\overline{\Theta};\Delta_2} \vdash \mathbf{rely}\ A_2 \wedge E\ \mathbf{guar}\ B_2}{P_1|P_2^{\Gamma;\Delta} \vdash \mathbf{rely}\, A_1 \wedge A_2\, \mathbf{guar}\, B_1 \wedge B_2}$$

(Par-$!!_{rw}??_{rw}$) $(\Delta = \Delta_1, \Delta_2)$

$$\frac{P_1^{\Gamma;\Delta_1,\Theta} \vdash \mathbf{rely}\, A_1\ \mathbf{guar}\, B_1 \wedge D \wedge E \qquad P_2^{\Gamma,\Theta;\uparrow\Delta_2} \vdash \{C \wedge D\}\,\mathbf{rely}\, A_2 \wedge E\, \mathbf{guar}\, B_2\,\{C' \wedge D'\}}{P_1|P_2{}^{\Gamma;\Delta} \vdash \{C\}\,\mathbf{rely}\, A_1 \wedge A_2\, \mathbf{guar}\, B_1 \wedge B_2 \wedge D'\,\{C'\}}$$

(Hide-global)

$$\frac{P^{\Gamma,x:ref(\tau)} \vdash \mathbf{rely}\ A\,\mathbf{guar}\ B^{\text{-}x}}{(\boldsymbol{\nu}\,x)P^{\Gamma} \vdash \mathbf{rely}\ A\,\mathbf{guar}\ B}$$

(Weak-$?_{rw}$)

$$\frac{P^{\Gamma} \vdash \{C\}\,\mathbf{rely}\ A\,\mathbf{guar}\ B\,\{C'\}}{P^{\Gamma,x:\overline{ref}(\tau)} \vdash \{C\}\,\mathbf{rely}\ A\,\mathbf{guar}\ B\,\{C'\}}$$

(Consequence-thread)

$$\frac{(C \wedge A \supset C_0 \wedge A_0,\ B_0 \wedge C_0' \supset B \wedge C) \qquad P \vdash \{C_0\}\,\mathbf{rely}\ A_0\,\mathbf{guar}\ B_0\,\{C_0'\}}{P \vdash \{C\}\,\mathbf{rely}\ A\,\mathbf{guar}\ B\,\{C'\}}$$

Fig. 30. Refined/Added Proof Rules for Open State (composition)

simpler form is equivalent in provability via the consequence rule:

$$(\mathsf{FSel}^{\uparrow})\ \frac{-}{\overline{x}\mathtt{in}_i\langle\vec{y}\rangle \vdash \{C\}\,\mathbf{rely}\ \ A[\mathtt{in}_i(\vec{y})^{\uparrow}/x]\,a\,\mathbf{guar}\ \ A\ \{C\}}$$

Put in this form, this is simply a stateful version of the rule for encoded free outputs, presented in Section 3.7. We also list the rule for bound output, which is easily derived from the free output rule via parallel composition and restriction.

$$(\mathsf{Sel}^{\uparrow})\ \frac{P \vdash \mathbf{rely}\ \ A\ \ \mathbf{guar}\ \ B[\mathtt{in}_i(\vec{y})^{\uparrow}/x]}{\overline{x}\mathtt{in}_i(\vec{y})P \vdash \{C\}\,\mathbf{rely}\ A\,\mathbf{guar}\ B\,\{C\}}$$

The rule for bound output gives one well-articulated form especially in the light of its symmetry with the branching rule: however the free output rule is more flexible essentially because it is about a more fine-grained construct.

For the replication and its dual, let us first take a look at the (In) rule.

$$(\mathsf{In}^{!})\ \frac{P \vdash \{C\}\,\mathbf{rely}\ A^{\text{-}\vec{y}} \wedge B_1^{\vec{y}}\ \mathbf{guar}\ B_2\,\{C'\} \qquad \forall \vec{y}.(B_1 \supset \{C\}[x\bullet\vec{y}/z]B_2)\{C'\} \supset B}{!x(\vec{y}z).P \vdash \mathbf{rely}\ \ A\ \ \mathbf{guar}\ \ B}$$

The rule incorporates the stateful invocation into the original (In) rule. The rule can be read as the original rule except it "folds" the state change (from C to C') into the replication. Suppose P guarantees B_2 at z and state C', relying on

(Bra$^{\downarrow}$)
$$\frac{\forall i.\, P_i \vdash \{C\}\,\mathbf{rely}\ A[\mathtt{in}_i(\vec{y}_i)^{\uparrow}/x]\,\mathbf{guar}\,B\,\{C'\}}{x[\&_i(\vec{y}_i).P_i] \vdash \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}}$$

(FSel$^{\uparrow}$) $(A \supset B[\mathtt{in}_i(\vec{y})^{\uparrow}/x])$
$$\frac{-}{\overline{x}\mathtt{in}_i\langle\vec{y}\rangle \vdash \{C\}\,\mathbf{rely}\ A\ \mathbf{guar}\ B\ \{C\}}$$

(In$^{!}$) $(\forall\vec{y}.(\,B_1 \supset \{C\}[x\bullet\vec{y}/z]B_2\{C'\}\,) \supset B)$
$$\frac{P \vdash \{C\}\,\mathbf{rely}\ A^{\text{-}\vec{y}} \wedge B_1^{\vec{y}}\,\mathbf{guar}\ B_2\ \{C'\}}{!x(\vec{y}z).P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B}$$

(FOut$^{?}$) $(A \supset [\{C\}x\bullet\vec{y}\{C'\}/z]B)$
$$\frac{-}{\overline{x}\langle\vec{y}z\rangle \vdash \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}}$$

(Ref) $(A[!x/y] \supset B\,)$
$$\frac{-}{\mathsf{Ref}\langle xy\rangle \vdash \mathbf{rely}\,A\,\mathbf{guar}\,B}$$

(Read) $(C \supset B[(!x)^{\uparrow}/c\,])$
$$\frac{-}{\overline{x}\mathsf{read}\langle c\rangle \vdash \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C\}}$$

(Write)
$$\frac{A[!x/y] \wedge \exists x.C \supset C'}{\overline{x}\mathsf{write}\langle yc\rangle \vdash \{C\}\,\mathbf{rely}\ A \wedge B[()^{\uparrow}/c]\,\mathbf{guar}\,B\,\{C'\}}$$

Fig. 31. Proof Rules for Processes with Open State (prefix)

A and B_1 and starting from the state C. Then $!x(\vec{y}z).P$ can guarantee that, for all $\vec{y}$ satisfying B_1, the result of invoking x with $\vec{y}$ at C satisfies B_2 and C'.

The co-replication also incorporates state change: Figure 31 again presents the rule for free output, which is better suited when we introduce local state.

$$(\mathsf{FOut}^{?})\ \frac{A \supset \{C\}[x\bullet\vec{y}/z]B\{C'\}}{\overline{x}\langle\vec{y}z\rangle \vdash \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}}$$

The rule is naturally read, saying if A is such that, at x (which is negative), $\vec{y}$ (which are also negative) and z (which is so too), invoking x with arguments $\vec{y}z$ starting from C, then that output surely ensures B and C' if one starts from C. Again the following rule is easily derived, showing the versatility of the free output.

$$(\mathsf{Out}^{?})\ \frac{P \vdash \{C''\}\,\mathbf{rely}\ A^{\text{-}z} \wedge A_2^{z}\,\mathbf{guar}\ A_1^{\vec{y}} \wedge B^{\text{-}\vec{y}}\{C'\} \qquad A \supset A_1 \supset \{C\}[x\bullet\vec{y}/z]A_2\{C''\}}{\overline{x}(\vec{y}z)P \vdash \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}}$$

In this rule, we demand that A stipulates for x that, when x is fed $\vec{y}$ as given by A_1 under the state C, it converges with an answer z as given by A and the resulting state C'', and that A_1 is already guaranteed by (the non-thread part of) P while if P, starting from C'' under A, reaches C' and ensures B, then $\overline{x}(\vec{y}z)P$ as a whole ensures that, starting from C and relying on A, reaches C and ensures B.

Note we do not need to mention convergence guarantee since it is already implied in $\{C\}[x \bullet \vec{y}/z]B_2\{C'\}$.

Next we turn to (Ref), which introduces the reference agent.

$$(\mathsf{Ref})\ \frac{A[!x/y] \supset B}{\mathsf{Ref}\langle xy\rangle \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B}$$

It says that, for a reference to guarantee B as its property, what B says about the value of x, which is y, should be ensured in the rely condition. The rule is effectively equivalent to, assuming x occurs only in the form $!x$ (which is not restrictive under the assumption of the identity distinction (cf. §9.5):

$$(\mathsf{Ref'})\ \frac{-}{\mathsf{Ref}\langle xy\rangle \vdash \mathbf{rely}\, B[y/!x]\, \mathbf{guar}\, B}$$

which is in fact equivalent to the original rule.

The next rule is for reading a reference:

$$(\mathsf{Read})\ \frac{C \supset B[(!x)^{\dagger}/c\,]}{\overline{x}\mathsf{read}\langle c\rangle \vdash \{C\}\, \mathbf{rely}\, A\, \mathbf{guar}\, B\, \{C\}}$$

In this rule, we use the free output version of the read action, which gives a simpler rule (we soon present the bound output version). It is instructive to look at how the free name version behaves:

$$\mathsf{Ref}\langle xv\rangle|\overline{x}\mathsf{read}\langle c\rangle \quad\longrightarrow\quad \mathsf{Ref}\langle xv\rangle|\overline{c}\langle v\rangle$$

Based on this dynamics, (Read) rule says that, if, at c, the read action needs to ensure B, then the specification C for the reference at x should be stronger than B in terms of the value it carries. Further there is no state change which takes place. Replacing C with $B[(!x)^{\dagger}/c\,]$ is possible if we use the structural rule for conjunction, but we believe the present rule is more naturally read. Since $\overline{x}\mathsf{read}(c)P$ can be represented as $(\boldsymbol{\nu}\, c)(\overline{x}\mathsf{read}\langle c\rangle|P)$, we can easily derive the rule for the bound output version of the read rule:

$$(\mathsf{Read'})\ \frac{P \vdash \{C\}\, \mathbf{rely}\, A^{\text{-}c} \wedge E^{c}\, \mathbf{guar}\, B\, \{C'\} \quad C \supset E[(!x)^{\dagger}/c\,]}{\overline{x}\mathsf{read}(c)P \vdash \{C\}\, \mathbf{rely}\, A\, \mathbf{guar}\, B\, \{C'\}}$$

which simply replaces the guarantee condition on c in B of (Read) with the rely condition at E (which is needed for the process P in the antecedent to guarantee its own specification). We can easily check (Read) can also be derived from (Read') by translating free outputs into bound outputs via copycats. Thus (Read) and (Read') are equivalent: the former is more fine-grained, while the latter is sometimes useful when we encode program logics into process logics.

The next rule is for the writing action.[10] Again we present the rule in the free output format.

$$\frac{\begin{array}{l}(\mathsf{Write})\\ A[!x/y] \wedge \exists x.C \supset C'\end{array}}{\overline{x}\mathsf{write}\langle yc\rangle \vdash \{C\}\,\mathbf{rely}\,A\wedge B[()^{\uparrow}/c]\,\mathbf{guar}\,B\,\{C'\}}$$

For reference we list the dynamics of the writing action in its free output version:

$$\mathsf{Ref}\langle xv\rangle|\overline{x}\mathsf{write}\langle wc\rangle \quad\longrightarrow\quad \mathsf{Ref}\langle xw\rangle|\overline{c}$$

Given the dynamics above, (Write) can be read close to Hoare's original rule for assignment (reformulation by an existential quantifier is by Hoare and Wirth [50]). It says that, for the write action to guarantee B and C:

– if B says anything about c then it should be consistent with $c = ()^{\uparrow}$ (and if not the rely condition becomes absurd); and
– if E says anything about the content of x, then it should be implied by the specification for y, together with constraint on other references given by C.

Note B can only be about c by typing. Similarly, by typing, A can only be about (at most) y, since if otherwise it cannot occur in the pre-condition. Since we can represent $\overline{x}\mathsf{write}(yc)P$ as $(\boldsymbol{\nu}\, yc)(\overline{x}\mathsf{write}\langle yc\rangle|P)$, we obtain the following bound output version of the rule.

$$(\mathsf{Write'})\ \frac{P \vdash \{C\}\,\mathbf{rely}\,A^{\text{-}y} \wedge c = ()^{\uparrow}\,\mathbf{guar}\,B^{\text{-}y} \wedge E^{y}\,\{C'\} \quad E[!x/y] \wedge \exists x.C \supset C'}{\overline{x}\mathsf{write}(yc)P \vdash \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}}$$

The rule says that if P guarantees E for the written value y, then the immediately resulting specification C' should be implied by E (with y replaced by $!x$) as well as C for references othert than x.

Finally, with respect to the use of entailment in proofs, the following axiom schema is useful in reasoning.

Proposition 6. *Assume a and $\vec{b}$ are respectively typed $(\tau\rho)^{!}_{\Upsilon}$ and $\vec{\tau}$, both C and C' are typed under $\Upsilon\cdot\Gamma$, and E and E_0 are typed $\Upsilon'\cdot\Gamma$ and Γ, respectively, with $\mathsf{dom}(\Upsilon')\cap\mathsf{dom}(\Upsilon) = \emptyset$. Then the following is valid in the logic with open state.*

$$(\mathsf{weaken\text{-}p})\quad (\{C\}[a\bullet\vec{b}/z]A\{C'\}\wedge(E\supset E_0))\ \supset\ \{C\wedge E\}[a\bullet\vec{b}/z](A\wedge E_0)\{C'\wedge E\} \tag{9.22}$$

Note the implication mirrors (Weak-$\mathbf{?}_{rw}$) in its shape. The validity of this entailment is immediate from the interpretation of invocation formulae.

Remark 8. (true linearity in reference agents) One may find the specifications on the linear action at c in both (Write) and (Write') rather trivial and, to that extent, cumbersome: indeed if we refine the type of c to be "truly linear", which means c is locally a linear channel in the sense of the linear type discipline in Section 3, there is no need to mention this name in formulae: this is because, in that case, $c = ()^{\uparrow}$ is logically equivalent to T.

[10] This rule was revised in the update in May 24th, 2004.

Remark 9. (structural rules with open state) In addition to the presented rules, it is convenient to use various structural rules, following those rules discussed for the stateless process logic (see §3.6, Page 35). For non-threads, we can use the same structural rules. For threads, the rules become:

$$(\mathsf{Conj})\ \frac{P^{\Gamma} \vdash \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B_i\,\{C_i\}\quad (i=1,2)}{P^{\Gamma} \vdash \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B_1 \wedge B_2\,\{C_1 \wedge C_2\}}$$

$$(\mathsf{Disj})\ \frac{P^{\Gamma} \vdash \{C_i\}\,\mathbf{rely}\,A_i\,\mathbf{guar}\,B\,\{C\}\quad (i=1,2)}{P^{\Gamma} \vdash \{C_1 \vee C_2\}\,\mathbf{rely}\,A_1 \vee A_2\,\mathbf{guar}\,B\,\{C\}}$$

$$(\mathsf{Aux}\text{-}\wedge,\vee)\ \frac{P^{\Gamma} \vdash \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}\quad \mathsf{fn}(E)\cap\mathsf{dom}(\Gamma)=\emptyset}{P^{\Gamma} \vdash \{C \star E\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C' \star E\}}\qquad (\star \in \{\wedge,\vee\})$$

In (Aux-$\wedge,\vee$), combining E only at pre-post conditions does not lack generality since, by (Consequence), we can move E to the rely-guarantee formulae. (Aux-$\wedge$) gives the equivalent of the standard "conjunction" rule in Hoare Logic when combined with the weakening rule (note the usual side condition "if a program does not write to a variable" is automatically guaranteed because a weakened reference can never be touched, either for reading or writing, by the process). This point becomes semantically non-trivial when we consider logics for local state. and will be discussed in Remark 11.7, §11.6, Page 182 later.

The following is proved in [53, §4] for a model equivalent to the present one.

Theorem 7. (soundness of proof rules for logic with open state) *Let P be a process with open state. Then $P^{\Gamma;\Delta} \vdash \mathbf{rely}\,A\,\mathbf{guar}\,B$ implies $P^{\Gamma;\Delta} \models \mathbf{rely}\,A\,\mathbf{guar}\,B$. Also $P^{\Gamma;\Delta} \vdash \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}$ implies $P^{\Gamma;\Delta} \models \{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}$.*

9.7 Reasoning Examples

We list simple reasoning examples as illustration of the proof rules, taking assertions in Section 9.4. To focus on the essential aspects of the reasoning, we let processes send and receive natural numbers (or, more generally, atomic values). For outputs we write, for example, $\overline{x}\langle 2\rangle$ or $\overline{x}\langle 2 \times \mathsf{n}\rangle$, using free output notations. If we wish to mix free and bound outputs, we write $\overline{x}\langle 3(c)\rangle$ etc., similarly for writing. For input we use the same notations, using in particular $n,\ldots$ for formal parameters standing for natural numbers. Formulae in the logic use atomic expressions just as names. For newly introduced syntax, we easily obtain the proof rules via encoding. We list those rules we shall use in the following in Figure 32. The rules for free output follows those in §3.7. There is no need for the read rule for atomic expressions since it does not directly mention a value to be read.

(LIn-n)
$\forall n.\ P \vdash \{C\}\,\mathbf{rely}\ A[(n)^{\uparrow}/x]\,\mathbf{guar}\ B\,\{C'\}$

$x(n).P \vdash \{C\}\,\mathbf{rely}\ A\,\mathbf{guar}\ B\,\{C'\}$

(LOut-n)
$-$

$C \vdash \{\overline{x}\langle \mathsf{e}\rangle\}\,\mathbf{rely}\ A[(\mathsf{e})^{\uparrow}/x]\,\mathbf{guar}\ A\,\{C\}$

($\mathsf{In}^{!}$-n) $(\forall n.(B_1 \supset \{C\}[x \bullet n/z]B_2\{C'\}) \supset B)$
$P \vdash \{C\}\,\mathbf{rely}\ A^{\text{-}n} \wedge B_1^n\ \mathbf{guar}\ B_2\,\{C'\}$

$!x(nz).P \vdash \mathbf{rely}\ A\ \mathbf{guar}\ B$

($\mathsf{Out}^{?}$-n) $(A \supset [\{C\}x \bullet \mathsf{e}\{C'\}/z]B)$
$-$

$\overline{x}\langle \mathsf{e}z\rangle \vdash \{C\}\,\mathbf{rely}\ A\,\mathbf{guar}\ B\,\{C'\}$

(Ref-n)
$A \wedge\ !x = \mathsf{e} \supset B$

$\mathsf{Ref}\langle x\mathsf{e}\rangle \vdash \mathbf{rely}\ A\,\mathbf{guar}\ B$

(Write-n)
$-$

$\overline{x}\mathsf{write}\langle \mathsf{e}c\rangle \vdash \{C[\mathsf{e}/!x]\}\,\mathbf{rely}\ A[()^{\uparrow}/c]\,\mathbf{guar}\ A\,\{C\}$

Fig. 32. Proof Rules for Value-Passing Processes

Using the newly introduced notations, we can concisely write down the behaviours in Section 9.2 (which are called P_1, P_2 and P_3 in Section 9.4) as follows.

$$
\begin{aligned}
P_1' &\stackrel{\text{def}}{=} \overline{y}\mathsf{read}(c)c(n).\overline{z}\mathsf{write}(2\times n, w)\\
P_2' &\stackrel{\text{def}}{=}\ !u(w).\overline{y}\mathsf{read}(c)c(n).\overline{z}\mathsf{write}(nw)\\
P_3' &\stackrel{\text{def}}{=} \overline{x}\mathsf{read}(c)c(c').\overline{c'}\langle 3(z)\rangle z(n).\overline{w}\langle n\rangle
\end{aligned}
$$

We shall derive the following assertions for these processes (cf. § 9.4).

$$P_1' \vdash \{!y = 3\}\,\mathbf{rely}\ \mathsf{T}\,\mathbf{guar}\ z = ()^{\uparrow}\ \{!z = 6\} \tag{9.23}$$

$$P_2' \vdash \mathbf{rely}\ \mathsf{T}\,\mathbf{guar}\ \forall i^{\mathsf{Nat}}.\ \{!y = i\}\ u \bullet \varepsilon = ()^{\uparrow}\ \{!z = 2\times i\} \tag{9.24}$$

$$P_3' \vdash \{\mathsf{T}\}\,\mathbf{rely}\ \forall i^{\mathbb{N}}.\ (!y \bullet i = 2 \times i)\,\mathbf{guar}\ \mathsf{T}\ \{z = (6)^{\uparrow}\}\mathsf{T} \tag{9.25}$$

We start from (9.24), omitting inessential steps as before.

1. $\overline{z}\mathsf{write}\langle 2\times n, w\rangle \vdash \{!y = 3\}\,\mathbf{rely}\ n = 3\,\mathbf{guar}\ w = ()^{\uparrow}\ \{!z = 6\}$ (Write-n)

---

2. $c(n).\overline{z}\mathsf{write}\langle 2\times n, w\rangle \vdash \{!y = 3\}\,\mathbf{rely}\ c = (3)^{\uparrow}\,\mathbf{guar}\ w = ()^{\uparrow}\ \{!z = 6\}$ (In-n)

---

3. $\overline{y}\mathsf{read}(c)c(n).\overline{z}\mathsf{write}\langle 2\times n, w\rangle \vdash \{!y = 3\}\,\mathbf{rely}\ \mathsf{T}\,\mathbf{guar}\ w = ()^{\uparrow}\ \{!z = 6\}$ (Read')

as required. The reasoning for (9.24) just adds the input prefix on the basis of the above derivation, whose conclusion we write $(*)$.

1. $P_1' \vdash \{!y = 3\}\,\mathbf{rely}\ \mathsf{T}\,\mathbf{guar}\ w = ()^{\uparrow}\ \{!z = 6\}$ $(*)$

---

2. $!u(w).P_1' \vdash \mathbf{rely}\ \mathsf{T}\,\mathbf{guar}\ \forall i^{\mathsf{Nat}}.\ \{!y = i\}\ u \bullet \varepsilon = ()^{\uparrow}\ \{!z = 2\times i\}$ (In-n)

It may be interesting to look at the inference of a specification for a dual behaviour of P_2', setting A to be $\forall i^{\mathsf{Nat}}.\,\{!y = i\}\, u \bullet \varepsilon = ()^{\uparrow}\, \{!z = 2\times i\}$.

1.	$\overline{v} \vdash \{!y = 3\}\, \mathbf{rely}\, A\, \mathbf{guar}\, v = ()^{\uparrow}\, \{!y = 3\}$	(Sel)
2.	$w.\overline{v} \vdash \{!y = 3\}\, \mathbf{rely}\, A \wedge w = ()^{\uparrow}\, \mathbf{guar}\, v = ()^{\uparrow}\, \{!y = 3\}$	(Bra)
3.	$\overline{u}(w)w.\overline{v} \vdash \{!y = 3\}\, \mathbf{rely}\, A\, \mathbf{guar}\, v = ()^{\uparrow}\, \{!z = 6\}$	(Out)

Note the effect of an output at u is recorded in the pre/post conditions, reminiscent of the rule for procedure in Hoare Logic [46].

Finally we derive (9.25). Below for brevity we let $C \stackrel{\text{def}}{=} \forall i^{\mathbb{N}}.\,(!y \bullet i = 2 \times i)$ and $E \stackrel{\text{def}}{=} \exists e.\,(c = (e)^{\uparrow} \wedge e \bullet 3 = (6)^{\uparrow})$.

1.	$\overline{w}\langle n\rangle \vdash \{C\}\, \mathbf{rely}\ n = 6\, \mathbf{guar}\, w = (6)^{\uparrow}\, \{\mathsf{T}\}$	(LOut-n)
2.	$z(n).\overline{w}\langle n\rangle \vdash \{C\}\, \mathbf{rely}\ z = (6)^{\uparrow}\, \mathbf{guar}\, w = (6)^{\uparrow}\, \{\mathsf{T}\}$	(LIn-n)
3.	$\overline{c'}\langle 3(z)\rangle z(n).\overline{w}\langle n\rangle \vdash \{C\}\, \mathbf{rely}\, c' \bullet 3 = (6)^{\uparrow}\, \mathbf{guar}\, w = (6)^{\uparrow}\, \{\mathsf{T}\}$	(Out-n)
4.	$c(c').\overline{c'}\langle 3(z)\rangle z(n).\overline{w}\langle n\rangle \vdash \{C\}\, \mathbf{rely}\, B\, \mathbf{guar}\, w = (6)^{\uparrow}\, \{\mathsf{T}\}$	(Bra)
5.	$\overline{y}\mathsf{read}(c)c(c').\overline{c'}\langle 3(z)\rangle z(n).\overline{w}\langle n\rangle \vdash \{C\}\, \mathbf{rely}\, \mathsf{T}\, \mathbf{guar}\, w = (6)^{\uparrow}\, \{\mathsf{T}\}$	(Read')

For going from Line 3 to Line 4, we use:

$$c' \bullet 3 = (6)^{\uparrow} \quad \supset \quad \exists e.\,(\,c = (e)^{\uparrow} \wedge\ e \bullet 3 = (6)^{\uparrow})[(c')^{\uparrow}/c].$$

which immediately holds by the definition of $\exists$. For going from Line 4 to Line 5, we use:

$$\forall i^{\mathbb{N}}.\,(!y \bullet i = 2 \times i) \quad \supset \quad \exists e.\,(\,(!y)^{\uparrow} = (e)^{\uparrow} \wedge\ e \bullet 3 = (6)^{\uparrow}\,)$$

which is also easy to derive by the definitions of $\forall$ and $\exists$.

10 Program Logics for Higher-Order Imperative Programs (1): Languages with Open State

10.1 An Imperative Extension of PCF$_v$

The purpose of this section is to present the compositional program logics for two basic instances of higher-order programming language as applications of the process logic with open state. Because of the combination of higher-order computation and statefulness, a wide range of language constructs are now treatable. Further the reasoning principles for the corresponding languages form the fundamental basis of those for more complex languages.

We start from an imperative extension of PCF$_v$, which we call PCF$_{v\mu}$. The grammar of terms add:

$$M \;::=\; \dots \quad | \quad \mathtt{skip} \quad | \quad !x \quad | \quad x := M$$

`skip` is a constant standing for the skip command (causing one step reduction without no visible behaviour), $!x$ is a dereference (reading the content of x), and $x := M$ is the standard assignment (first evaluates M and assigners it to x). Since `skip` is a constant, it is added to the set of values.

For types, we list the grammar in full for clarity.

$$\begin{aligned} \alpha \;&::=\; \mathbb{B} \quad | \quad \mathbb{N} \quad | \quad \mathsf{Com} \quad | \quad \alpha \stackrel{\Upsilon}{\Rightarrow} \beta \\ \delta \;&::=\; \mathit{ref}(\alpha) \end{aligned}$$

Com is a command type (which is essentially the unit type, with the same process encoding, cf. Section 8), $\alpha \stackrel{\Upsilon}{\Rightarrow} \beta$ is an arrow type with an *effect typing* or often simply *effect* Υ, where Υ is a finite map from names to types of the form $\mathit{ref}(\alpha_i)$. $\mathit{ref}(\alpha)$ is a reference type which stores a value of type α. Υ in $\alpha \stackrel{\Upsilon}{\Rightarrow} \beta$ (which comes from the process typing in Section 9) says that, when an abstraction is fed with an argument, the resulting term may read and/or write at Υ during evaluation.[11] This minimal notion of effects is needed for providing a precise typing for assertions, as we shall discuss soon. We say two effects are *consistent* if they assign the same reference type to each common name. We assume two effects occurring in one type should always be consistent with each other. When Υ is the empty map, we write $\alpha \Rightarrow \beta$ for $\alpha \stackrel{\Upsilon}{\Rightarrow} \beta$. As before, we write $\Gamma, \Delta, \dots$ for bases, whose range never include reference types.

The typing uses two kinds of sequents, $\Gamma \vdash V : \alpha$ for values (which do not mention effects explicitly) and $\Gamma; \Upsilon \vdash M : \alpha$ for general terms (which do mention effects explicitly). The typing rules are given in Figure 33. In each rule, we assume all effects occurring in any pair of two types, including those in conclusion, are mutually consistent. The rules for variables and constants are as before, using

[11] Operationally folding the effect in this way makes sense since an abstraction does not have any effect to the outside unless it is invoked, a fundamental observation in effect and type discipline [110].

$$[Var]\frac{-}{\Gamma, x:\alpha \vdash x:\alpha} \quad [Num]\frac{-}{\Gamma \vdash \mathbf{n}:\mathbb{N}} \quad [Succ]\frac{\Gamma;\Upsilon \vdash M:\mathbb{N}}{\Gamma;\Upsilon \vdash \mathtt{succ}(M):\mathbb{N}} \quad [Eq]\frac{\Gamma;\Upsilon \vdash M_{1,2}:\mathbb{N}}{\Gamma;\Upsilon \vdash M_1 = M_2:\mathbb{B}}$$

$$[Abs]\ \frac{\Gamma, x:\alpha;\Upsilon \vdash M:\beta}{\Gamma \vdash \lambda x^{\alpha}.M:\alpha \overset{\Upsilon}{\Rightarrow} \beta} \quad [App]\ \frac{\Gamma;\Upsilon \vdash M:\alpha \overset{\Upsilon'}{\Rightarrow} \beta \quad \Gamma;\Upsilon \vdash N:\alpha \quad \Upsilon' \subset \Upsilon}{\Gamma;\Upsilon \vdash MN:\beta} \quad [Val]\ \frac{\Gamma \vdash V:\alpha}{\Gamma;\Upsilon \vdash V:\alpha}$$

$$[Rec]\ \frac{\Gamma, x:\alpha \overset{\Upsilon}{\Rightarrow} \beta \vdash \lambda y^{\alpha}.M : \alpha \overset{\Upsilon}{\Rightarrow} \beta}{\Gamma \vdash \mu x^{\alpha \overset{\Upsilon}{\Rightarrow} \beta}.\lambda y^{\alpha}.M : \alpha \overset{\Upsilon}{\Rightarrow} \beta} \qquad [If]\ \frac{\Gamma\,;\,\Upsilon \vdash M : \mathbb{B} \quad \Gamma\,;\,\Upsilon \vdash N_{1,2} : \alpha}{\Gamma\,;\,\Upsilon \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : \alpha}$$

$$[Skip]\ \frac{-}{\Gamma \vdash \mathtt{skip} : \mathsf{Com}} \qquad [Deref]\ \frac{x:ref(\alpha) \in \Upsilon}{\Gamma;\Upsilon \vdash\ !x : \alpha} \qquad [Ass]\ \frac{\Gamma;\Upsilon \vdash M : \alpha \quad x:ref(\alpha) \in \Upsilon}{\Gamma;\Upsilon \vdash x := M : \mathsf{Com}}$$

Fig. 33. Typing Rules for $\mathrm{PCF}_{v\mu}$

sequents for values. $[Abs]$ folds an effect of a thread to turn it to a value. $[App]$ adds an effect of an arrow type. $[Val]$ turns a value sequent to a thread sequent since a value itself is a thread. Other rules should be self-explanatory.

Any PCF_v-term is typable in the present typing with precisely the same typing as before. As a simple term with side effect:

$$x:ref(\mathbb{N}), y:ref(\mathbb{N}) \ \vdash \ y \ := \ !x \times 2 \ : \ \mathsf{Com} \tag{10.1}$$

which reads a number from y and assigns its double to x. Above the empty non-effect base is omitted. We can also store a function (closure) to a variable:

$$x:ref(\mathbb{N}\Rightarrow\mathbb{N}), y:ref(\mathbb{N}) \ \vdash \ x \ := \ \lambda z^{\mathbb{N}}.z \times 2 \ ; \ y := (!x)(!y) \ : \ \mathsf{Com} \tag{10.2}$$

where we define $M;N$ (with M being of type Com) as, with z fresh:

$$M;N \ \stackrel{\text{def}}{=} \ (\lambda z^{\mathsf{Com}}.N)M \tag{10.3}$$

Thus (10.2) first assigns a doubling function to x, reads x to get the function, reads y to get a number, and feeds the former with the latter, whose result is again stored in y. As an example of λ-abstraction encapsulating an effect:

$$\vdash \ \lambda y^{\mathbb{N}}.\,(!x)y \ : \ \mathbb{N} \overset{\Upsilon}{\Rightarrow} \mathbb{N} \tag{10.4}$$

where we set $\Upsilon \stackrel{\text{def}}{=} x : ref(\mathbb{N} \Rightarrow \mathbb{N})$. The term, when invoked with an integer argument, calls a function stored in a reference x with the value it has just received, and return the result. Note the latent effect gets explicit when we do application:

$$x:ref(\mathbb{N}\Rightarrow\mathbb{N}) \vdash \ (\lambda y^{\mathbb{N}}.\,(!x)y)\,3 \ : \ \mathbb{N} \tag{10.5}$$

which, if x stores a doubling function (for example by the preceding command $x := \lambda z^{\mathbb{N}}.z \times 2$), would return 6.

10.2 Assertions for $\mathrm{PCF}_{v\mu}(1)$: General Ideas

Following the technical development in Section 9, assertions for $\mathrm{PCF}_{v\mu}$-terms augment those for PCF_v-terms with the pre/post-conditions, recording a passage of time. For example, the following is a specification for the term in (10.1).

$$\{!x = 3\}\,\mathsf{T} \vdash\ y := !x \times 2\ :_u u = \mathtt{skip}\,\{!y = 6\} \tag{10.6}$$

As before, the judgement is in fact under the strict typing of primary names: the pre/post conditions assert on $x : \mathit{ref}(\mathbb{N}), y : \mathit{ref}(\mathbb{N})$, the rely condition assert on the (empty) base, and the guarantee condition asserts on the typed anchor u. We can read (10.6) as: if the initial state is such that x stores 3, then the program is ensured to terminate with the state such that y stores 6, with the resulting value `skip`. The formula $u = \mathtt{skip}$ is understood in the same way as (say) $u = 6$ in $x = 3 \vdash x \times 2 :_u u = 6$. However, since `skip` is the sole value of type Com, this is in fact vacuous under the assumption of termination, so one may as well write:

$$\{!x = 3\}\,\mathsf{T} \vdash\ y := !x \times 2\ \{!y = 6\} \tag{10.7}$$

which may be seen as a retrospective origin of Hoare triple.

When a function is stored in a variable, we dereference the variable and then write down the constraint. The following shows this for the first part of (10.4).

$$\{\mathsf{T}\}\,\mathsf{T} \vdash\ x := \lambda z^{\mathbb{N}}.z \times 2\ \{\forall i.\, !x \bullet i = 2 \times i\} \tag{10.8}$$

In the same way we can specify the second part of (10.4).

$$\{\forall i.\, !x \bullet i = 2 \times i\ \wedge\ !y = 3\}\,\mathsf{T} \vdash\ y := (!x)y\ \{\forall i.\, !x \bullet i = 2 \times i\ \wedge\ !y = 6\} \tag{10.9}$$

which asserts on both x and y in its pre/post conditions.

Let us next consider a specification for abstraction which involves (latent) effects, taking the term given in (10.5).

$$\mathsf{T} \vdash \lambda y.(!x)y\ :_u \{\forall i.\, !x \bullet i = 2 \times i\}\, u \bullet 3 \searrow 6\, \{\mathsf{T}\} \tag{10.10}$$

The guarantee formula in (10.10) says that, if 3 is applied to u under the initial state $\{\forall i.\, !x \bullet i = 2 \times i\}$, then the resulting term evaluates to 6, with some final state. Note the type of u is $\mathbb{N} \overset{x:\mathit{ref}(\mathbb{N}\Rightarrow\mathbb{N})}{\Rightarrow} \mathbb{N}$, indicating an invocation at u may involve access to a reference x. The internalised precondition in the formula conforms to this typing. The expression $\{\forall i.\, !x \bullet i = 2 \times i\}$ is syntactically a single formula, so that we may add, for example using a conjunction, similar predicates for other possible hypothetical initial states in the guarantee condition. Since the abstraction can be invoked many times, and each time an invocation can be done under a different state, this is indeed reasonable. We also note the formula of the form $\{C\}e_1 \bullet e_2 = e'\{C'\}$ is in fact an abbreviation of a semantically more general predicate, discussed in the next subsection.

In the preceding example (10.10), the program hypothetically stored in x is purely functional. Next we treat the case when a program with side effects is stored in a variable.

$$\mathsf{T} \vdash \lambda y.(!x)y\ :_u \{C \wedge\ !w = 0\}\, u \bullet 3 \searrow 6\, \{C \wedge\ !w = 1\} \tag{10.11}$$

where we set C to be the following formula:

$$\forall i, n.\ \{!w = n\}\, !x \bullet i \searrow 2 \times i\, \{!w = n+1\} \tag{10.12}$$

The pre-condition C assumes that x stores a function which, when invoked, would increment w as well as returning the double of the argument. If u uses it under the condition $!w = 0$, then the result is (other than the unchanged x) $!w = 1$, and the double of the argument as the return value.

10.3 Assertions for $\mathbf{PCF}_{v\mu}(2)$: Syntax and Semantics

Below we introduce the formal syntax of terms and formulae of $\mathrm{PCF}_{v\mu}$-logic, extending that of the PCF_v-logic.

$$\begin{array}{rcl} e & ::= & \ldots \mid \texttt{skip} \mid\ !x^{ref(\alpha)} \\ A & ::= & \ldots \mid \{C\}[e_1 \bullet e_2/z]A\{C'\} \end{array}$$

We have already seen the two added terms. `skip` stands for the constant of $\mathrm{PCF}_{v\mu}$ with the same symbol internalised in the logical language (just as other constants), which has the type Com; while $!x^{ref(\alpha)}$ is a dereference of x, and has the type α. The added formula $\{C\}[e_1 \bullet e_2/z]A\{C'\}$ is the general form of a similar predicate used in the previous subsection, and is a form of substitution. The formula can intuitively be read as follows.

> *The application of e_2 to e_1 under the initial state C terminates with the final state C' and the resulting value, named as z, satisfies A.*[12]

Note this closely corresponds to the judgement $\{C\}A \vdash M :_z B\{C'\}$. Using this predicate, we can define the predicate used in §10.2, as well as its simplified version, as follows. Let z be fresh below.

$$\{C\}\, e_1 \bullet e_2 \searrow e'\{C'\} \quad \stackrel{\text{def}}{=} \quad \{C\}[e_1 \bullet e_2/z]\,(z = e')\,\{C'\} \tag{10.13}$$

$$\{C\}\, e_1 \bullet e_2\, \{C'\} \quad \stackrel{\text{def}}{=} \quad \{C\}[e_1 \bullet e_2/z]\,\mathsf{T}\,\{C'\} \tag{10.14}$$

Names in the formula $\{C\}[e_1 \bullet e_2/z]A\{C'\}$ are strictly typed, by the following well-typedness condition: For some α, β, Υ:

- e_1 is typed as $\alpha \stackrel{\Upsilon}{\Rightarrow} \beta$ while e_2 is typed as α;
- Each of C and C' is typed under $\Upsilon \cdot \Upsilon' \cdot \Theta$ where Υ' is a (weakened) effect typing disjoint from Υ and Θ is a typing for auxiliary names; and
- A is typed under $z : \beta$ together with a global typing for auxiliary names.

Note the effect typing of the arrow type of e_1 completely determines the primary names and their types in pre/post conditions. The interpretation of this formula, which we shall discuss soon, crucially depends on this typing.

We have already seen two forms of judgement, one for values and the other for general terms, which we formally introduce below.

[12] Such z cannot be determined uniquely in general because C represents a set of states rather than a single state.

- $A \vdash V^{\Gamma;\alpha} :_u B$ (for values), assuming $\Gamma \vdash V : \alpha$. For typing we require, apart from the common typing on auxiliary names, A is typed under Γ and B is typed under $u\!:\!\alpha$.
- $A \vdash \{C\}\, M^{\Gamma;\Upsilon;\alpha} :_u B\{C'\}$ (for general terms), assuming $\Gamma;\Upsilon \vdash M : \alpha$. For typing we require, apart from the common typing on auxiliary names, C and C' are typed under Υ, A is typed under Γ, and B is typed under $u\!:\!\alpha$.

Note these two forms are in accordance with the two kinds of sequents in $\mathrm{PCF}_{v\mu}$-typing. The primary names are the union of those which occur in its base, effect (if any) and the anchor. Auxiliary names are names in the formulae which are not primary (we note it often suffices to have only non-reference names among auxiliary names, though having auxiliary reference names is sometimes convenient, and becomes necessary when we allow non-trivial distinctions).

In the following we briefly outline a basic model construction, based on which we give semantics of formulae and judgement. The semantics is given for total correctness, which is easily adapted to other kinds of logics. Before starting, First let $\cong_\lambda$ (in $\mathrm{PCF}_{v\mu}$) be defined as the maximum congruence satisfying, for each $\Upsilon \vdash M_{1,2} : \mathsf{Com}$,

$$M_1 \cong_\lambda M_2 \qquad \equiv \qquad \langle M_1, \sigma\rangle \Downarrow \langle \mathtt{skip}, \sigma'_1\rangle \;\text{ iff }\; \langle M_2, \sigma\rangle \Downarrow \langle \mathtt{skip}, \sigma'_2\rangle$$

Above we assume the standard reduction of the form $\langle M, \sigma\rangle \longrightarrow \langle M', \sigma'\rangle$, cf. [32] (note the difference between $\sigma'_{1,2}$ is observable by enclosing contexts).

Call M *semi-closed* if it is typed as $\Upsilon \vdash M : \alpha$ (i.e. the base is empty except for an effect typing, the latter being present since we cannot hide it!). Then we let, as before (cf. §6.4), $\lambda, \kappa, \ldots$ range over the $\cong_\lambda$-congruence classes of typed semi-closed terms, which we call *behaviours*. We write $\lambda^{\Upsilon;\alpha}$ if λ is of type α under the effect typing Υ. Again there is a unique congruent class of the diverging terms in each such typing, which we write $\perp$. If $\lambda \neq \perp$, we say λ is *total*. Then we introduce:

1. A *model* ξ *of type* Γ, often written ξ^Γ, is a finite map from $\mathsf{dom}(\Gamma)$ to total behaviours conforming to Γ.
2. A *store* $\boldsymbol{\sigma}$ *of type* Υ, often written $\boldsymbol{\sigma}^\Upsilon$, is a finite map from $\mathsf{dom}(\Upsilon)$ to behaviours conforming to Υ.
3. An *interpretation* $\mathcal{I}$ *of type* Γ is a finite map from $\mathsf{dom}(\Gamma)$ to (possibly non-total) behaviours conforming to Γ.

Following §6.4, Page 68, we interpret $x^{ref(\alpha)}$ as x itself, while $!x$ under $\boldsymbol{\sigma}$ is interpreted simply as $\boldsymbol{\sigma}(x)$. There are two forms of the satisfaction relations, $\xi^\Gamma \models^{\mathcal{I}} A$ for behaviours and $\boldsymbol{\sigma}^\Upsilon \models^{\mathcal{I}} C$ for stores. The defining clauses are the same for both, adding the following definition for stateful application (all assuming appropriate typings). We first define $\langle [\![e_1]\!]_{\mathcal{I}\cdot\xi} \bullet [\![e_2]\!]_{\mathcal{I}\cdot\xi}, \boldsymbol{\sigma}\rangle \Downarrow \langle \lambda, \boldsymbol{\sigma}'\rangle$ from the reduction of the corresponding concrete terms/environments. We then set:

$$\xi \models^{\mathcal{I}} \; \{C\}[e_1 \bullet e_2/z]A\{C'\}$$
$$\overset{\mathrm{def}}{\Leftrightarrow}$$
$$\boldsymbol{\sigma} \models^{\mathcal{I}} C \;\;\supset\;\; \exists\lambda.\,(\,\langle [\![e_1]\!]_{\mathcal{I}\cdot\xi} \bullet [\![e_2]\!]_{\mathcal{I}\cdot\xi}, \boldsymbol{\sigma}\rangle \Downarrow \langle \lambda, \boldsymbol{\sigma}'\rangle \;\wedge\; \xi \models^{\mathcal{I}\cdot z:\lambda} \; A \;\wedge\; \boldsymbol{\sigma}' \models^{\mathcal{I}} C').$$

The rest follows the structure of formulae, with $\boldsymbol{\sigma}^{\Upsilon} \models^{\mathcal{I}} \forall x^{ref(\alpha)}.C$ interpreted with x ranging over $\mathsf{dom}(\Upsilon)$, similarly for $\boldsymbol{\sigma}^{\Upsilon} \models^{\mathcal{I}} \exists x^{ref(\alpha)}.C$ (cf. §9.5). From this we set, with the obvious definition of $\models_{\mathsf{b}}$:

- $\vec{x}:\vec{M}^{\Gamma} \models^{\mathcal{I}} A$ iff $\vec{x}:\vec{M}^{\Gamma} \models_{\mathsf{b}} \xi^{\Gamma}$ and $\xi^{\Gamma} \models^{\mathcal{I}} A$ for some ξ.
- $\sigma^{\Upsilon} \models^{\mathcal{I}} C$ iff $\sigma^{\Upsilon} \models_{\mathsf{b}} \boldsymbol{\sigma}^{\Upsilon}$ and $\boldsymbol{\sigma}^{\Upsilon} \models^{\mathcal{I}} C$ for some $\boldsymbol{\sigma}$.

We also set $M \models^{\mathcal{I}}_{u} B \ \equiv\ u:M \models^{\mathcal{I}} B$. Semantics of judgements follows.

Definition 10. (semantics of assertions)

1. *$A \models M^{\Gamma;\alpha} :_u B$ iff, under each $\mathcal{I}$ and each $\vec{V}$ such that $\vec{x}:\vec{V}^{\Gamma} \models^{\mathcal{I}} A$, we have $M[\vec{V}/\vec{x}] \models^{\mathcal{I}}_{u} B$.*
2. *$\{C\}\, A \models M^{\Gamma;\Upsilon;\alpha} :_u B\, \{C'\}$ iff, under each $\mathcal{I}$, each A such that $\vec{x}:\vec{V} \models^{\mathcal{I}} A$ and each σ such that $\sigma \models^{\mathcal{I}} C$, we have $\langle M[\vec{V}/\vec{x}], \sigma\rangle \Downarrow \langle M', \sigma'\rangle$ such that $\sigma' \models^{\mathcal{I}} C'$.*

With the natural extension of translation of formulae (cf. §6.4), the validity is fully abstractly embeddable into the stateful process logic in Section 9 (for this all we need is in fact definability of processes and environments, which is best done using additional let constructs for the source language and the bound name version for the target calculus).

We record two basic observations on the semantics just defined.

Remark 10. (treatment of reference names) Given two distinct reference names $x^{ref(\alpha)}$ and $y^{ref(\alpha)}$, we never have $\sigma \models^{\mathcal{I}} x = y$ for any σ, that is $x^{ref(\alpha)} = y^{ref(\alpha)}$ is logically equivalent to absurdity.

By Remark 10, we may safely restrict the use of reference names in formulae to the form $!x$, even though we shall find the use of equality over bare occurrences of reference names in Section 12.

Remark 11. (treatment of `skip`) Given $\Gamma;\Upsilon \vdash M : \mathsf{Com}$, we have $\{C\}A \vdash M :_u B\{C'\}$ if and only if $\{C\}A \vdash M :_u u = \mathtt{skip}\{C'\}$ [because: since u is primary, we know it is total, but the all and only total value in Com is `skip`].

The following Convention does not lose precision by Remark 11.

Convention 5. *If $\Gamma;\Upsilon \vdash M : \mathsf{Com}$, we shall often write $\{C\}A \vdash M\{C'\}$ for $\{C\}A \vdash M^{\Gamma;\Upsilon;\mathsf{Com}} :_u B\{C'\}$.*

10.4 Proof Rules for PCF$_{v\mu}$

The poof rules for the PCF$_{v\mu}$-logic are given in Figure 34 (all rules are under Convention 1, §3.5, Page 31, as always). Each rule corresponds to the typing rule of the same name in Figure 33. These proof rules are a natural extension of those for PCF$_v$(cf. Figure 11, Section 6.5, Page 11), adjusting the shape of

$$[\mathit{Var}]\ \frac{-}{A[x/u] \vdash x :_u A} \qquad [\mathit{Skip}]\frac{-}{A[\mathtt{skip}/u] \vdash \mathtt{skip} :_u A} \qquad [\mathit{Val}]\frac{A \vdash M :_u B}{A \vdash \{C\}\, M :_u B\{C\}}$$

$$[\mathit{Op}]\ \frac{\{C_{i-1}\}A \vdash M_i :_{m_i} B_i\{C_i\}\ \ (1 \le i \le n) \qquad \wedge_i B_i \supset B[\mathsf{op}(m_1, .., m_n)/u]}{\{C_0\}A \vdash \mathsf{op}(M_1, .., M_n) :_u B\{C_n\}}$$

$$[\mathit{If}]\ \frac{A \vdash \{C\}M :_b A'\{D\} \quad A \wedge A'[\mathsf{T}/b] \vdash \{D\}N_1 :_u B\{C'\} \quad A \wedge A'[\mathsf{F}/b] \vdash \{D\}N_2 :_u B\{C'\}}{A \vdash \{C\}\ \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 :_u B\{C'\}}$$

$$[\mathit{Abs}]\ \frac{\{C\}\, A^{\text{-}x\vec{i}} \wedge A_1^x \vdash M :_m A_2\,\{C'\} \quad \forall x\vec{i}.\,(A_1 \supset \{C\}[u \bullet x/m]A_2\{C'\}\,) \supset B}{A \vdash \lambda x.M :_u B}$$

$$[\mathit{App}]\ \frac{A \vdash \{C\}\, M :_m B_1\{D\} \quad A \vdash \{D\}\, N :_x B_2\{E\} \quad B_1 \wedge B_2 \supset \{E\}[m \bullet x/u]B\{C'\}}{A \vdash \{C\}MN :_u B\{C'\}}$$

$$[\mathit{Rec}]\ \frac{A \vdash \lambda y.M :_u B(0) \qquad A \wedge B(i)[x/u] \vdash \lambda y.M :_u B(i+1)}{A \vdash \mu x.\lambda y.M :_u \forall i.B(i)}$$

$$[\mathit{Deref}]\ \frac{C \supset B[\,!x/u]}{A \vdash \{C\}\ \ !x :_u B\,\{C\}} \qquad [\mathit{Ass}]\ \frac{\{C\}\, A \vdash M :_y E\,\{D^{\text{-}x}\} \quad D \wedge E[!x/y] \supset C'}{\{C\}\ A \wedge B[\mathtt{skip}/u] \vdash x := M :_u B\,\{C'\}}$$

$$[\mathit{Consequance}]\ \frac{A \wedge C \supset A_0 \wedge C_0 \quad A_0 \vdash \{C_0\}\, M :_u B_0\{C_0'\} \quad B_0 \wedge C_0' \supset B \wedge C'}{A \vdash \{C\}\, M :_u B\{C'\}}$$

Fig. 34. Proof Rules for $\mathrm{PCF}_{v\mu}$ (total correctness, main rules)

sequents, refining the application operator to stateful ones, as well as adding the rules for the new constructs.

[*Var*] is as before. [*Skip*] is listed since `skip` is newly introduced: the rule is in fact another instance of constant rules. [*Val*] corresponds to (Val) in Figure 33, turning a judgement for values to the corresponding one for general programs, adding pre/post conditions (with no state change). It is worth noting this rule, which incurs no change on term form, in fact corresponds to the following derivation in the stateful process logic which does accompany a term change. Below $\langle\!\langle V \rangle\!\rangle_m^{\mathrm{val}}$ is a value-mapping given as in Section 6.2; $[\![V]\!]_u$ is a non-value mapping, given as $\overline{u}(m)\langle\!\langle V \rangle\!\rangle_m^{\mathrm{val}}$; and $\langle\!\langle A \rangle\!\rangle^{\vec{x}}$ is an encoding of a formula as given in Section

6.4, all naturally extended to the added constructs. We assume well-typedness.

$$\frac{\dfrac{1.\quad \langle\!\langle V\rangle\!\rangle_m^{\text{val}} \vdash \textbf{rely}\ \langle\!\langle A\rangle\!\rangle^{\vec{x}}\ \textbf{guar}\ \langle\!\langle B[m/u]\rangle\!\rangle^{m} \qquad \textsf{(assumption)}}{2.\quad \langle\!\langle V\rangle\!\rangle_u \vdash \{\langle\!\langle C\rangle\!\rangle^{\varepsilon}\}\ \textbf{rely}\ \langle\!\langle A\rangle\!\rangle^{\vec{x}}\ \textbf{guar}\ \langle\!\langle B\rangle\!\rangle^{\varepsilon}\ \{\langle\!\langle C\rangle\!\rangle^{\varepsilon}\} \qquad \textsf{(Sel)}}}{3.\quad \langle\!\langle V\rangle\!\rangle_u \vdash \{\langle\!\langle C\rangle\!\rangle^{\vec{y}}\}\ \textbf{rely}\ \langle\!\langle A\rangle\!\rangle^{\vec{x}}\ \textbf{guar}\ \langle\!\langle B\rangle\!\rangle^{\varepsilon}\ \{\langle\!\langle C\rangle\!\rangle^{\vec{y}}\} \qquad \textsf{(Weak-Thread)}}$$

The next rule, $[Op]$, is the stateful refinement of the original rule with the same name. It assumes, given $\mathsf{op}(\vec{M})$, that: (1) op is strict on each argument and is itself total; and (2) evaluation is done from the left to the right. Because of (2) the postcondition for M_i is compensated by the precondition for M_{i+1} for each i. The assumption (2) means the rule cannot be applied if the order of evaluations differ, or, as is more likely, is not fixed. In fact many programming languages do *not* fix the evaluation order in many operations. Against this background, it is clearly undesirable to let programs and their specifications depend on the order of evaluations of operations except for few cases (independence from the order of evaluation means, in the context of the process encoding, the rule is valid under several possible translations). In fact the present framework allows a natural way to solve this issue, which we shall discuss in §10.5 to follow.

$[If]$ also refines the original rule with compensating pre/post conditions (in this case the order of evaluation is the only feasible one). The next two rules are worth illustration in some detail. Firstly,

$$[Abs]\ \frac{\{C\}\, A^{\text{-}\vec{xi}} \wedge A_1^{x} \vdash M :_m A_2\, \{C'\} \quad \forall \vec{xi}.\, (A_1\ \supset\ \{C\}[u \bullet x/m]A_2\{C'\}\,)\ \supset\ B}{A \vdash \lambda x.M :_u B}$$

turns a non-value sequent into a value sequent, folding the state change incurred by M (from C to C') into the statement about the abstraction. The side condition $\forall \vec{xi}.\,(A_1 \supset \{C\}[u \bullet x/m]A_2\{C'\}\,)$ can be read as:

> *For all $\vec{xi}$ for which A_1 holds, if u is applied to x under the initial state C, then it terminates with the final state C' and the resulting value is m for which A_2 holds.*

The application rule is its dual.

$$[App]\ \frac{A \vdash \{C\}\, M{:}_m B_1\{D\} \quad A \vdash \{D\}\, N{:}_x B_2\{E\} \quad B_1 \wedge B_2 \supset \{E\}[m \bullet x/u]B\{C'\}}{A \vdash \{C\} MN{:}_u B\{C'\}}$$

The rule assumes, as seen in the sequence of state predicates C, D and E, that, in MN, we first evaluates M, then N, and finally MN (note it is necessary to specify the evaluation order in this way just for having precise semantics in the presence of side effects) The side condition $B_1 \wedge B_2 \supset \{E\}[m \bullet x/u]B\{C'\}$ can be read as:

> *If B_1 holds for m and B_2 holds for x, then, when m is applied to x under the initial state E, it terminates with the final state C' and the resulting value is u for which B holds.*

Note the predicate entails termination of application, so no additional predicate for convergence is necessary.

The next rule [*Rec*] for recursion is identical with the original rule for PCF_v since it is always about a value. The rule for dereference:

$$[\mathit{Deref}]\ \frac{C \supset B[\,!x/u]}{A \vdash \{C\}\ \ !x :_u B\,\{C\}}$$

says that, for $!x$ to satisfy B, the same or stronger specification should be given to x in its precondition C. No state change is incurred by the reading, so the postcondition is also C. This rule is almost a direct translation of (Read) in Figure 31, observing the mapping $\langle\!\langle !x \rangle\!\rangle_u \stackrel{\text{def}}{=} \overline{x}\mathsf{read}\langle u\rangle$ and noting the side condition of (Read), "$C \supset B[(!x)^{\uparrow}/u\,]$", is equivalent to the side condition of [*Deref*], "$C \supset B[!x/u\,]$" by taking off the indirection by the linear output.

The rule for assignment uses the substitution $C[y/!x]$ which stands for substitution of y for all occurrences of $!x$, just as in (Write) in §9.6. Apart from the state change incurred by evaluation of the assigned term M, the rule is reducible to the following rule when the assigned term M is a value (we also omit the trivial guarantee formula, cf. Remark 10, Page 144).

$$[\mathit{Ass\text{-}val}]\ \frac{A \vdash V :_y E \quad \exists x.C \wedge E[x/y] \supset C}{\{\exists x.C\}\, A \vdash x := V\,\{C\}}$$

saying: for $x := V$ (which surely terminates) to ensure the final state to be C, C should come from $E[x/y]$ for its x-part (which is about V) together with C on other variables.

It would be instructive to see how [*Ass-val*] is decomposed into inferences in the process logic, with $\langle\!\langle x := V\rangle\!\rangle_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, yu')(\overline{x}\mathsf{write}\langle yu'\rangle | \langle\!\langle V\rangle\!\rangle_y^{\mathrm{val}} | [u' \to u])$. Let $C_0 \stackrel{\text{def}}{=} \exists y.\,(\langle\!\langle C\rangle\!\rangle^{\vec{z}}[y/!x] \wedge \langle\!\langle E\rangle\!\rangle^{y})$ and $R \stackrel{\text{def}}{=} \overline{x}\mathsf{write}\langle yu'\rangle | \langle\!\langle V\rangle\!\rangle_y^{\mathrm{val}} | [u' \to u]$ for brevity.

1.	$\overline{x}\mathsf{write}\langle yu'\rangle \vdash \{C_0\}\,\mathbf{rely}\ \langle\!\langle E\rangle\!\rangle^{y} \wedge \langle\!\langle B[\mathtt{skip}/u']\rangle\!\rangle^{\varepsilon}\ \mathbf{guar}\ \langle\!\langle B\rangle\!\rangle^{\varepsilon}\ \{C\}$	(Write)
2.	$\langle\!\langle V\rangle\!\rangle_y^{\mathrm{val}} \vdash \mathbf{rely}\ \langle\!\langle A\rangle\!\rangle^{\vec{z}}\ \mathbf{guar}\ \langle\!\langle E\rangle\!\rangle^{y}$	(assumption)
3.	$[u' \to u] \vdash \mathbf{rely}\ \langle\!\langle B[\mathtt{skip}/u]\rangle\!\rangle^{\varepsilon}\ \mathbf{guar}\ \langle\!\langle B[\mathtt{skip}/u']\rangle\!\rangle^{\varepsilon}$	(copycat)
4.	$R \vdash \{C_0\}\,\mathbf{rely}\ \langle\!\langle A\rangle\!\rangle^{\vec{z}} \wedge \langle\!\langle B[\mathtt{skip}/u]\rangle\!\rangle^{\varepsilon}\ \mathbf{guar}\ \langle\!\langle B\rangle\!\rangle^{\varepsilon}\ \{C\}$	(Par)
5.	$\langle\!\langle x := V\rangle\!\rangle_u \vdash \{C_0\}\,\mathbf{rely}\ \langle\!\langle A\rangle\!\rangle^{\vec{z}} \wedge \langle\!\langle B[\mathtt{skip}/u]\rangle\!\rangle^{\varepsilon}\ \mathbf{guar}\ \langle\!\langle B\rangle\!\rangle^{\varepsilon}\ \{C\}$	(Res)

For the consequence rule, Figure 34 lists:

$$[\mathit{Consequence}]\ \frac{A \wedge C \supset A_0 \wedge C_0 \quad A_0 \vdash \{C_0\}\, M :_u B_0\{C_0'\} \quad B_0 \wedge C_0' \supset B \wedge C'}{A \vdash \{C\}\, M :_u B\{C'\}}$$

which is for general terms (note primary names in A and C, resp. B and C', are never mixed by the well-typedness assumption). The rule for values is identical with the original rule for PCF_v. Both rules (as well as others) use validity on open stateful sequential behaviour, for which we may as well stipulate basic reasoning rules in addition to the standard rules (which include the generic rules for the predicate calculus with equality as well as those for natural numbers). Among others the extensionality axiom is better given using the stateful invocation allowing divergence, cf. Remark 7. The use of possibly nonconvergent invocation is suitable for generalised total correctness.

Finally we list the basic structural rules other than the consequence rules. All are from the corresponding rules in the process logic with open state Remark 9, §9.6, Page 136.

$$\text{(Conj)}\ \frac{A \vdash \{C\}\, M^{\Gamma,\Upsilon;\alpha} :_u B_i \{C_i'\} \quad (i=1,2)}{A \vdash \{C\}\, M^{\Gamma,\Upsilon;\alpha} :_u B_1 \wedge B_2 \{C_1' \wedge C_2'\}}$$

$$\text{(Disj)}\ \frac{A_i \vdash \{C_i\}\, M^{\Gamma,\Upsilon;\alpha} :_u B \{C'\} \quad (i=1,2)}{A_1 \vee A_2 \vdash \{C_1 \vee C_2\}\, M^{\Gamma,\Upsilon;\alpha} :_u B \{C'\}}$$

$$\text{(Aux-}\wedge,\vee\text{)}\ \frac{A \vdash \{C\}\, M^{\Gamma,\Upsilon;\alpha} :_u B \{C'\} \quad \mathsf{fn}(E) \cap \mathsf{dom}(\Gamma) = \emptyset}{P^{\Gamma} \vdash \{C \star E\}\, \mathbf{rely}\, A\, \mathbf{guar}\, B\, \{C' \star E\}} \quad (\star \in \{\wedge, \vee\})$$

As a structural rules of another kind, the following weakening rule changes the typing of an effect name.

$$\text{(Weak-Ref)}\ \frac{A \vdash \{C\}\, M^{\Gamma,\Upsilon;\alpha} :_u B \{C'\}}{A \vdash \{C\}\, M^{\Gamma,\Upsilon,x:ref(\beta);\alpha} :_u B \{C'\}}$$

Combining (Aux-$\wedge$) and (Weak-Ref) gives us the following rule.

$$\text{(Weak-Ref-}\wedge\text{)}\ \frac{A \vdash \{C\}\, M^{\Gamma,\Upsilon;\alpha} :_u B \{C'\}}{A \vdash \{C \wedge E\}\, M^{\Gamma,\Upsilon,\Upsilon';\alpha} :_u B \{C' \wedge E\}}$$

which corresponds to the following structural rule in Hoare logic:

$$\frac{\{C\}P\{C'\} \quad P \text{ does not write to a variable in } \mathsf{fn}(E)}{\{C \wedge E\}P\{C' \wedge E\}}$$

The main use of the standard rule above is when one does not wish to mention a predicate on some part of the store which a program does not touch at all: (Weak-Ref-$\wedge$) precisely corresponds to this case. Note the standard rule needs the side condition whose meaning is clear in the first-order imperative language but may not be so when complex data structures including higher-order datum are stored in references. The condition however becomes completely unnecessary in (Weak-Ref-$\wedge$), since a program can never write (and read) on weakened names. Note the type information plays the fundamental role. This is one of the examples where the use of type information in the present logical method leads to the principled treatment of subtle features in reasoning. In Sections 11 and 12, we shall see how we can extend the use of tthe identical rule even in the presence of computation with local state which may, among others, involve shared data with destructive update.

10.5 Evaluation Order Independence: a simple solution

As we already briefly mentioned when illustrating the rule $[Op]$ in §10.4, Page 146, the independence from evaluation order is an important property of programs' behaviours and, perhaps more importantly, in specifications. When one looks at a program $M \times N$, it is not immediate that which of M and N is evaluated first. This does not matter if M and N are both side-effect-free: but if they do have side effects, writing on some variables either directly or indirectly (through, say, another procedure call), the order can affect both the resulting state and the value itself.

A natural idea from the viewpoint of compositional program logic is to only allow the derivation of evaluation-order independent specifications in a compositional way. That is, we want our proof rules to be *evaluation-order independent*, so that the derived properties should work whatever order of evaluation of arguments the implementor of a language chooses for a given operator. This is in fact elegantly achievable in the present context by the use of typed predicates.

For concreteness, let us consider the multiplication operator, $M \times N$. We start from a slightly restrictive, but simpler, rule. As before, we are bound by Convention 1: always well-typedness, and no mix of primary and auxiliary names.

$$(\mathsf{mult\text{-}eoi\text{-}1})\ \frac{\{C_i\}\, A \vdash M_i^{\Gamma;\Upsilon_i} :_{m_i} B_i\, \{C_i'\} \quad B_1 \wedge B_2 \supset B[m_1 \times m_2/u]}{\{C_1 \wedge C_2\}\, A \vdash M_1 \times M_2^{\Gamma;\Upsilon_1,\Upsilon_2} :_u B\, \{C_1' \wedge C_2'\}}$$

(eoi stands for "evaluation order independence"). The rule hinges on the type annotation of M_2, $\Gamma;\Upsilon_1,\Upsilon_2$. By convention, the expression "Υ_1,Υ_2" indicates $\mathsf{dom}(\Upsilon_1) \cap \mathsf{dom}(\Upsilon_1) = \emptyset$ (note this differs from $\mathsf{fn}(\Upsilon_1) \cap \mathsf{fn}(\Upsilon_1) = \emptyset$ which in general is more strict). The rule says that, as far as the evaluation of M_1 and that of M_2 touch (read/write) disjoint names, we can safely combine individual assertions into one via conjunction [13] without caring the evaluation order of M_1 and M_2. We can check the rule is indeed sound in both evaluation orders, one in which M_1 is first evaluated and then M_2 is, and another in which M_2 is first evaluated and then M_1 is (but *not* if they are done concurrently). The soundness of ($\mathsf{mult\text{-}eoi\text{-}1}$) can be verified by syntactically deriving the rule using the original (fixed-order) multiplication rule together with (Weak-Ref-$\wedge$).

($\mathsf{mult\text{-}eoi\text{-}1}$) completely prohibits shared reference names between programs, which may as well be too restrictive in practice. Essentially the same derivation for justifying ($\mathsf{mult\text{-}eoi\text{-}1}$) can also derive the following strengthening version.

$$(\mathsf{mult\text{-}eoi\text{-}2})\ \frac{\{C^{\Upsilon} \wedge C_i^{\Upsilon_i}\}\, A \vdash M_i^{\Gamma;\Upsilon,\Upsilon_i} :_{m_i} B_i\, \{C^{\Upsilon} \wedge C_i'^{\Upsilon_i}\} \quad B_1 \wedge B_2 \supset B[m_1 \times m_2/u]}{\{C \wedge C_1 \wedge C_2\}\, A \vdash M_1 \times M_2^{\Gamma;\Upsilon,\Upsilon_1,\Upsilon_2} :_u B\, \{C \wedge C_1' \wedge C_2'\}}$$

We now allow an invariant C of primary names at Υ to be specified on common references which both M_1 and M_2 may touch. A typical situation is when $\mathsf{dom}(\Upsilon)$

[13] The nature of this conjunction is close to the so-called *separated conjunction* advocated by Reynolds, O'Hearn and others [95, 104], though, in their approach, one may further have to specify, at the level of predicates, that (say) M_1 does not touch names in $\mathsf{dom}(\Upsilon_2)$. Further discussions on this and related points are found in Section 11.

are only read by two programs, even though the invariance allows greater freedom. If this invariant at common references is maintained and if that invariance together with the conditions on exclusively owned parts of stores guarantees the stipulated post/guarantee conditions, then the composite program would guarantee the conjunction of individual predicates together with the invariant. The same idea is easily extended to the general first-order operations and application.

10.6 Reasoning Examples

In this subsection we explore the reasoning in the $\mathrm{PCF}_{v\mu}$-logic using the proof rules presented in the previous subsection. Along the way we shall present the derived rules for imperative constructs. In the following it is natural, though not necessary, to assume all names of reference types occur in the dereferenced form in formulae. We first derive the standard assignment axiom [45] from the present one. First, a preparation.

Definition 11. *A semi-closed term $\Upsilon \vdash M : \alpha$ is* simple *if M contains neither an application nor an abstraction as its subterm.*

If M is simple, it contains nothing except dereferences, constants and first-order operators, so that we can regard M itself as a term in our logical language. It has no side effects, by which the behaviour becomes highly transparent: but if so, we naturally ask whether there is a better reasoning rule than the assignment rule we gave before. First we note:

$$\text{(Simple)}\ \frac{M \text{ simple}}{\{C\}\ \mathsf{T} \vdash M :_u u = M\ \{C\}} \tag{10.15}$$

which is direct by the structural induction on M. Using (Simple) we infer as follows.

1.	$\{\exists y.(C[y/!x] \wedge y = M)\}\mathsf{T} \vdash M :_y y = M\{\exists y.(C[y/!x] \wedge y = M)\}$	(Simple)
2.	$\{\exists y.(C[y/!x] \wedge y = M)\}\ \mathsf{T} \vdash x := M\ \{C\}$	(Ass)
3.	$\{C[M/!x]\}\ \mathsf{T} \vdash x := M\ \{C\}$	(Consequence)

Thus we arrive at the required rule.

$$\text{(Ass-Simple)}\ \frac{M \text{ simple}}{\{C[M/!x]\}\ \mathsf{T} \vdash x := M\ \{C\}}$$

Using (Ass-Simple), we can quickly infer assignments involving only simple terms. As an example, the following is a detailed inference for (10.7) in §10.2. Below $(*)$ is calculated under stores of type $x : \mathit{ref}(\mathbb{N})$.

1.	$\{!x \times 2 = 6\}\ \mathsf{T} \vdash\ y\ :=\ !x \times 2\ \{!y = 6\}$	(Ass-simple)
2.	$!x = 3 \quad \supset \quad !x \times 2 = 6$	$(*)$
3.	$\{!x = 3\}\ \mathsf{T} \vdash\ y\ :=\ !x \times 2\ \{!y = 6\}$	(1,2, Consequence)

We next consider the assignment of a higher-order term (hence we can no longer use (Ass-Simple), taking (10.8) from §10.2. Below we assume the typing $x : \mathit{ref}(\mathbb{N} \Rightarrow \mathbb{N})$ and let $E \stackrel{\text{def}}{=} \forall i.(y \bullet i = i \times 2)$.

1.	$\{\mathsf{T}\}\, z = i \vdash z \times 2 :_m m = i \times 2\, \{\mathsf{T}\}$	(Var, Num, $\times$)
2.	$\mathsf{T} \vdash \lambda z.(z \times 2) :_y E$	(Abs)
3.	$\{\mathsf{T}\}\, \mathsf{T} \vdash \lambda z.(z \times 2) :_y E\, \{\mathsf{T}\}$	(Val)
4.	$\mathsf{T} \;\supset\; \exists y.(\, E[!x/y][y/!x] \;\wedge\; E\,)$	
5.	$\{\mathsf{T}\}\, \mathsf{T} \vdash \lambda z.(z \times 2) :_y E\, \{\mathsf{T}\}$	(3, 4, Consequence)
6.	$\{\mathsf{T}\}\, \mathsf{T} \vdash y := \lambda z.(z \times 2)\, \{E[!x/y]\}$	(Ass)

The next example, an inference for assertions in (10.9), dereferences a variable and uses it for application. We assume the type of x is $\mathbb{N} \Rightarrow \mathbb{N}$ (which allows us to use stateless application), and let $C \stackrel{\text{def}}{=} \forall i.\, !x \bullet i = 2 \times i$.

1.	$\{C \;\wedge\; !y = 3\}\, \mathsf{T} \vdash !x :_m m \bullet 3 = 6\, \{C \;\wedge\; !y = 3\}$	(Deref)
2.	$\{C \;\wedge\; !y = 3\}\, \mathsf{T} \vdash !y :_n n = 3\, \{C \;\wedge\; !y = 3\}$	(Deref)
3.	$\{C \;\wedge\; !y = 3\}\, \mathsf{T} \vdash (!x)(!y) :_u n = 3\, \{C \;\wedge\; !y = 3\}$	(App)
4.	$\{C \;\wedge\; !y = 3\}\, \mathsf{T} \vdash (!x)(!y) :_u u = 6\, \{C \;\wedge\; !y = 6\}$	(Conseq)
5.	$\{C \;\wedge\; !y = 3\}\, \mathsf{T} \vdash y := (!x)(!y) :_u u = 6\, \{C \;\wedge\; !y = 6\}$	(Ass)

The following example (cf. (10.11)) uses the same untyped term as the above example, storing the result of application to a variable, but involves state change in application, using the full power of stateful $[\mathit{App}]$. We let x be of type $\mathbb{N} \stackrel{\Upsilon}{\Rightarrow} \mathbb{N}$ with $\Upsilon \stackrel{\text{def}}{=} w : \mathit{ref}(\mathbb{N})$, and let C' be given by:

$$\forall i, n.\ \{!w = n\}\, !x \bullet i \searrow 2 \times i\, \{!w = n + 1\} \tag{10.16}$$

that is x, when invoked. returns the double of the argument and increments w.

1.	$\{C' \wedge !w = 0 \;\wedge\; !y = 3\}$ $\mathsf{T} \vdash !x :_m \{C' \wedge !w = 0\}\, m \bullet 3 \searrow 6\, \{C' \wedge !w = 1\}$ $\{C' \wedge !w = 0 \;\wedge\; !y = 3\}$	(Deref)
2.	$\{C' \wedge !w = 0 \;\wedge\; !y = 3\} \mathsf{T} \vdash !y :_l l = 3 \{C' \wedge !w = 0 \;\wedge\; !y = 3\}$	(Deref)
3.	$\{C' \wedge !w = 0\}\, m \bullet 3 \searrow 6\, \{C' \wedge !w = 1\} \;\wedge\; l = 3 \quad \supset$ $\{C' \wedge !w = 0 \wedge !y = 3\} m \bullet l \searrow 6 \{C' \wedge !w = 1 \wedge !y = 3\}$	
4.	$\{C' \wedge !w = 0 \;\wedge\; !y = 3\} \mathsf{T} \vdash (!x)(!y) :_u u = 6 \{C' \wedge !w = 1 \;\wedge\; !y = 3\}$	(1,2,3,App)
5.	$\{C' \wedge !w = 0 \;\wedge\; !y = 3\}\, \mathsf{T} \vdash y := (!x)(!y)\, \{C' \wedge !w = 1 \;\wedge\; !y = 6\}$	(Ass)

We next extend $\mathrm{PCF}_{v\mu}$ with a couple of standard imperative constructs via encoding, derive the standard proof rules for them, and apply them to non-trivial examples. We treat the following extensions, with $p, q, \ldots$ ranging over *procedure labels*.

$$M \ ::= \ \ldots \mid M; N \mid \mathtt{while}\ M\ \mathtt{do}\ N \mid p \mid \mathtt{proc}\ p(x) = \ M\ \mathtt{in}\ N$$

These constructs are naturally typed, and are encoded into the plain $\mathrm{PCF}_{v\mu}$-terms as follows (procedure labels are simply taken to be variables).

$$M; N \stackrel{\text{def}}{=} (\lambda().N)M$$
$$\mathtt{while}\ M\ \mathtt{do}\ N \stackrel{\text{def}}{=} (\mu w.\lambda().\mathtt{if}\ M\ \mathtt{then}\ N; (w\,\mathtt{skip})\ \mathtt{else}\ \mathtt{skip})\,\mathtt{skip}$$
$$\mathtt{proc}\ p(x) = \ M\ \mathtt{in}\ N \stackrel{\text{def}}{=} (\lambda p.N)(\lambda x.M) \qquad (f \notin \mathsf{fn}(M))$$

From these encodings we can derive the standard proof rules for these constructs. Firstly, we have already defined $M; N$, for which we can easily derive the following rule.

$$(\mathsf{Seq})\ \frac{\{C\}A \vdash M\{D\} \quad \{D\}A \vdash N :_u B\{C'\}}{\{C\}A \vdash M; N :_u B\{C'\}}$$

If N is also of type Com, then we can entirely omit B.

The next rule is for the total correctness of loop command, and would be worth looking at its derivation in detail. Let M be simple below.

$$(\mathsf{While})\ \frac{A \vdash \{C\}\, M :_m B\{C\} \ \ A \vdash \{C \wedge B \wedge \varphi = n\}\, N \{C \wedge \varphi \lneq n\} \ \ C \wedge B \supset \varphi \gneq 0}{\{C\}\, A \vdash \mathtt{while}\ M\ \mathtt{do}\ N\, \{C \wedge B[\mathbf{f}/m]\}}$$

We need to justify the rule using the recursion rule which uses mathematical induction. A formula which links this rule to the mathematical induction follows.

$$\begin{aligned} E(m) \stackrel{\text{def}}{=} & \{C \wedge \neg B\}\, w \bullet \mathtt{skip}\, \{C \wedge \neg B\} \ \wedge \\ & \forall n \text{ s.t. } 0 \lneq n \leq m.\ \{C \ \wedge\ \varphi = n \wedge B\}\, w \bullet \mathtt{skip}\, \{C \ \wedge\ \varphi \lneq n \wedge \neg B\} \end{aligned}$$

Note $\forall m.E(m) \supset \{C\} w \bullet \mathtt{skip} \{C \wedge \neg B\}$ under the assumption $C \wedge B \supset \ \varphi \geq 1$.

Let $L \stackrel{\text{def}}{=} \mathtt{if}\ B\ \mathtt{then}\ M; (w \cdot \mathtt{skip})\ \mathtt{else}\ \mathtt{skip}$. Then, by the encoding of while, we have $\mathtt{while}\ B\ \mathtt{do}\ M \stackrel{\text{def}}{=} \mu w.\, \lambda().L$. First we show the base case. (assumption) stands for $C \wedge B \supset \ \varphi \geq 1$ which is assumed throughout.

1.	$\mathsf{T} \vdash \{C \ \wedge\ \neg B \ \wedge\ B\}\ M; (w \cdot \mathtt{skip})\ \{C \ \wedge\ \neg B\}$	(falsity)
2.	$\mathsf{T} \vdash \{C \ \wedge\ \ \neg B \ \wedge\ \neg B\}\ \mathtt{skip}\ \{C \ \wedge\ \ \neg B\}$	(Skip)
3.	$\mathsf{T} \vdash \{C \ \wedge\ \neg B\}\ L\ \{C \ \wedge \neg B\}$	(If)
4.	$\mathsf{T} \vdash \ \lambda().L \ :_u E(0)[w/u]$	(Abs, Conseq)

where (falsity) is the lemma $A \vdash \{\mathsf{F}\}M\{C\}$ for any A and C, which is easily verified by rule induction. For induction step:

1.	$E(n) \vdash \{C \ \wedge \ \ \varphi = n+1 \ \wedge \ B\} \ M \ \{C \ \wedge \ \ \varphi \lneq n+1\}$	(assumption)
2.	$E(n) \vdash \{C \ \wedge \varphi \lneq n+1\} \ (w \cdot \texttt{skip}) \ \{C \ \wedge \ \varphi \lneq n+1 \wedge \neg B\}$	(Var, App)
3.	$E(n) \vdash \{C \ \wedge \ \varphi = n{+}1 \ \wedge \ B\} \ M; (w \cdot \texttt{skip}) \ \{C \ \wedge \ \varphi \lneq n{+}1 \wedge \neg B\}$	(Seq)
4.	$E(n) \vdash \{C \wedge \ \varphi = n+1 \wedge B \wedge \neg B\} \ \texttt{skip} \ \{C \wedge \ \varphi \lneq n+1 \wedge \neg B\}$	(falsity)
5.	$E(n) \vdash \{C \ \wedge \varphi = n+1 \wedge B\} \ L \ \{C \ \wedge \varphi \lneq n+1 \ \wedge \neg B\}$	(If)
6.	$E(n) \vdash \ \lambda().L \ \ :_u E(n+1)[u/w]$	(Abs, Conseq)

By combining the two conclusions by (Rec) we are done. Finally the following two rules for procedures are easily verified.

$$\text{(Call)}\ \frac{A \supset \{C\} f \bullet M \{C'\}}{A \vdash \{C\}\, f(M) \{C'\}}$$

$$\text{(Proc)}\ \frac{A \ \wedge \ E^{x\vec{i}} \vdash \{C_0\}\, N \{C_0'\} \quad A \ \wedge \ \forall x\vec{i}.(E \supset \{C_0\} f \bullet x \{C_0'\}) \vdash \{C\}\, M \{C'\}}{A \vdash \{C\}\, \texttt{proc}\ f(x) = \ N \ \texttt{in}\ M\{C'\}}$$

Using these rules, we reason for a program which calculates a factorial function, just as we did in Section 6: But this time the program uses a simple iterator and higher-order store. The iterator uses two cells, x and y. x stores a thunked boolean condition, while y stores a thunked action for each iteration. Below we list all expressions we shall use below. $!x()$ is parsed as $(!x)()$ (where we write "()" for "skip" for legibility; further we write $\lambda().M$ for $\lambda z^{\mathsf{Com}}.M$ with z fresh.

- $\mathsf{itr} \stackrel{\text{def}}{=} \texttt{while}\ !x()\ \texttt{do}\ !y()$, the iterator.
- $M \stackrel{\text{def}}{=} \lambda().\,(!i \le n)$, the thunked boolean.
- $N \stackrel{\text{def}}{=} \lambda().(v := !v * !i \ ; \ i := !i + 1)$, the iterative action.
- $\mathsf{body} \stackrel{\text{def}}{=} \lambda n.\ (i := 0; v := 1; x := M; y := N; g(); !v)$, the main body.
- $\mathsf{ifact} \stackrel{\text{def}}{=} \texttt{proc}\ g() = \mathsf{itr}\ \texttt{in}\ \mathsf{body}$, the iterative factorial program.

The configuration of code given above may not be a natural choice in high-level programming, but such a structure often arises in intermediate and low level representation of executable code. For properties we use the following.

$B(u)$	"$\forall z.u \bullet z = \ z!$"	(main specification)
I	"$!i = 0$"	(initial condition)
C	"$v = !i \ \wedge \ !i \ \le n+1$"	(loop invariant)
G	"$!i \le n$"	(loop range)
$E_0(w)$	"$\{C\}w()\{C\}$"	(property of M)
$E_1(w)$	"$\{G\}[w()/z](z = \mathbf{f})\{G'\}$"	(property of M)
$E_2(w)$	"$\{C \wedge \varphi = m\} \ w() \ \{C \wedge \varphi = m\}$"	(property of N)
E	"$E_1(!x) \wedge E_2(!x) \wedge E_3(!y)$"	(property of the environment)

where we set $\varphi \stackrel{\text{def}}{=} (n-!i)$. The main assertion we wish to establish is:

$$\mathsf{T} \vdash \mathsf{ifact} :_u B(u) . \tag{10.17}$$

The reasoning for the iterator uses (While-HO).

1.	$\mathsf{T} \vdash \{E \wedge C\}\ (!x)()\ \{E \wedge C\}$	(Var, Deref, App)
2.	$\mathsf{T} \vdash \{E \wedge C \wedge \varphi = m\}\ !x()\ \{E \wedge C \wedge \varphi = m\}$	(Var, Deref, App)
3.	$(C \wedge i \lneq n) \quad \supset \quad \varphi \geq 1$	$(\varphi = (n-!i))$
4.	$E \vdash \{E \wedge C\}$ itr $\{E \wedge C \wedge \neg G\}$	(1,2,3, While)
5.	$E \vdash \{I \wedge E \wedge C\}$ itr $\{E \wedge C \wedge \neg G\}$	(4, Conseq)

In the following we refer this conclusion by $(\star)$.

Next we show $\{I \wedge C\}\ x := M\ ;\ y := N\ \{C \wedge E\}$. These commands store higher-order values, hence reasoned using (Ass). Below for brevity we assume the assertion $\{I \wedge C\}x := M\{I \wedge E_0(!x) \wedge E_1(!x)\}$ is already inferred.

1.	$\mathsf{T} \vdash \{I \wedge C\}x := M\{I \wedge E_0(!x) \wedge E_1(!x)\}$	(...)
2.	$\mathsf{T} \vdash \{C \wedge \varphi = m\}\ v := v{*}\,!i\,;\, i := !i + 1\ \{C \wedge \varphi \lneq m\}$	(Seq, Ass, $\varphi = (n-!i)$)
3.	$\mathsf{T} \vdash N :_n E_2(n)$	(2, Abs)
4.	$\mathsf{T} \vdash \{C \wedge E_0(!x) \wedge E_1(!x)\}\ y := N\ \{C \wedge E_0(!x) \wedge E_1(!x) \wedge E_2(!y)\}$	(3, Ass)
5.	$\mathsf{T} \vdash \{I \wedge C\}x := M\ ;\ y := N\{C \wedge E\}$	(1, 4, Seq)

The conclusion above is referred to as $(\star\star)$. We are now in the final stage of inference. Let $A(g) \stackrel{\text{def}}{=} \{E \wedge I \wedge C\}\, g()\, \{E \wedge C \wedge \neg G\}$.

1.	$z = n \wedge A(g) \vdash \{\mathsf{T}\}i := 0\ ;\ v := 1\{I \wedge C\}$	(Ass, Seq)
2.	$z = n \wedge A(g) \vdash \{I \wedge C\}x := M\ ;\ y := N\{C \wedge E\}$	$(\star\star)$
3.	$z = n \wedge A(g) \vdash \{C \wedge E\}\ g()\ \{C \wedge E \wedge \neg G\}$	(Call)
4.	$z = n \wedge A(g) \vdash \{C \wedge E \wedge \neg G\}\, !v :_u u = n\mathbf{!}\ \{\mathsf{T}\}$	(Deref)
5.	$z = n \wedge A(g) \vdash \{\mathsf{T}\}\,(i := 0; v := 1; x := M; y := N; g(); !v) :_u u = n\mathbf{!}\{\mathsf{T}\}$	(1-4, Seq)
6.	$A(g) \vdash \lambda z.(i := 0; v := 1; x := M; y := N; g(); !v) :_u B(u)$	(5, Abs)
7.	$\mathsf{T} \vdash$ `proc` $g() =$ itr `in` body $:_u B(u)$	($\star$, 6, Proc)

We have reached the assertion (10.17).

$$\frac{(\text{var}) \quad -}{\Gamma;\Upsilon \vdash x : \Upsilon(x)} \qquad \frac{(\text{num}) \quad -}{\Gamma;\Upsilon \vdash \mathsf{n} : \mathbb{N}} \qquad \frac{(\text{proc-call}) \quad (\Gamma(p) = \vec{\alpha} \overset{\Upsilon}{\Rightarrow} \beta) \quad \Gamma;\Upsilon,\Upsilon' \vdash e_i : \alpha_i}{\Gamma;\Upsilon,\Upsilon' \vdash p(\vec{e}) : \beta}$$

$$\frac{(\text{skip}) \quad -}{\Gamma;\Upsilon \vdash \texttt{skip}} \qquad \frac{(\text{assign}) \quad \Gamma;\Upsilon \vdash e : \Upsilon(x)}{\Gamma;\Upsilon \vdash x := e} \qquad \frac{(\text{seq}) \quad \Gamma;\Upsilon \vdash P_1, \quad \Gamma;\Upsilon \vdash P_2}{\Gamma;\Upsilon \vdash P_1; P_2} \qquad \frac{(\text{if}) \quad \Gamma;\Upsilon \vdash e : \mathbb{B}, \quad \Gamma;\Upsilon \vdash P_{1,2}}{\Gamma;\Upsilon \vdash \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2}$$

$$\frac{(\text{while}) \quad \Gamma;\Upsilon \vdash e : \mathbb{B}, \quad \Gamma;\Upsilon \vdash P}{\Gamma;\Upsilon \vdash \texttt{while}\ e\ \texttt{do}\ P} \qquad \frac{(\text{var-dec}) \quad (x \notin \mathsf{fn}(\Gamma,\Upsilon)) \quad \Gamma;\Upsilon, x{:}\alpha \vdash P, \quad \Gamma;\Upsilon \vdash e : \alpha}{\Gamma;\Upsilon \vdash \mathsf{var}\ \alpha\ x := e\ \mathsf{in}\ P} \qquad \frac{(\text{return}) \quad \Gamma;\Upsilon \vdash e : \alpha}{\Gamma;\Upsilon \vdash \mathsf{return}\ e : \alpha}$$

$$\frac{(\text{seq-ret}) \quad \Gamma;\Upsilon \vdash P_1, \ \Gamma;\Upsilon \vdash P_2 : \alpha}{\Gamma;\Upsilon \vdash P_1; P_2 : \alpha} \qquad \frac{(\text{if-ret}) \quad \Gamma;\Upsilon \vdash e : \mathbb{B}, \ \Gamma;\Upsilon \vdash P_{1,2} : \alpha}{\Gamma;\Upsilon \vdash \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 : \alpha} \qquad \frac{(\text{var-ret}) \quad (x \notin \mathsf{fn}(\Gamma,\Upsilon)) \quad \Gamma;\Upsilon, x{:}\alpha \vdash P : \beta, \ \Gamma;\Upsilon \vdash e : \alpha}{\Gamma;\Upsilon \vdash \mathsf{var}\ \alpha\ x := e\ \mathsf{in}\ P : \beta}$$

$$\frac{(\text{proc dec}) \quad (\Gamma(p_i) = \vec{\alpha_i} \overset{\Upsilon_i}{\Rightarrow} \beta_i) \quad \Gamma;\Upsilon_i, \vec{y_i}{:}\vec{\alpha_i} \vdash P_i : \beta_i, \quad \Gamma;\Upsilon \vdash Q}{\Gamma/\vec{p}\ ;\Upsilon \vdash \{\mathsf{proc}\ p_i(\vec{\alpha}_i\ \vec{y}_i)P_i\}\ \mathsf{in}\ Q}$$

Fig. 35. An Imperative Language with Open State: Typing

10.7 Hoare Logic for Higher-Order Procedure

We next treat a logic for a simple imperative program with higher-order procedure, which in effect extends the original Hoare Logic with higher-order procedure. There is a report that such languages (which in effect include the $\mathrm{PCF}_{v\mu}$ we treated above) are inherently difficult to treat in program logics [8]: but they form the core part of many programming languages we everyday use, can be quite well-structured, and surely need and deserve good reasoning principles. As we shall see in the following we indeed have a clean logical articulation for specifying and reasoning about such a language. The language we shall treat (which is for partial correctness to make its connection to the original Hoare's logic as clear as possible) has a limited amount of locality at the level of formal parameters of procedures (which are, as in the standard imperative languages, regarded as local references). Perhaps surprisingly, this class of local references is precisely what we can treat in the present open stateful logic.

Commands, Expressions and Types. Let $P, Q, \ldots$ range over commands, while $e, e', ldots$ range over expressions. We use (imperative) variables $x, x', \ldots$ and procedure labels $p, p', \ldots$. The grammar is given by:

$$\begin{array}{rcl} P & ::= & \mathtt{skip} \mid x := e \mid P;Q \mid \mathtt{if}\; e\; \mathtt{then}\, P\, \mathtt{else}\, Q \mid \mathtt{while}\; e\; \mathtt{do}\, P \mid \\ & & \{\mathsf{proc}\; p_i(\vec{y}_i)P_i\}\; \mathsf{in}\; Q \mid \mathsf{return}\; e \\ e & ::= & x \mid p(\vec{e}) \mid \mathsf{c} \mid e+e' \mid e-e' \mid e=e' \mid \ldots \end{array}$$

For brevity the grammar assumes all procedures return some values. p in $p(\vec{e})$ is called *procedure name*. The procedure declaration $\{\mathsf{proc}\; p_i(\vec{y}_i)P_i\}\; \mathsf{in}\; Q$ can involve mutual recursive calls. We do not include anonymous functions (λ-abstractions) for brevity even though they can be equally treated. In encoding into the π-calculus, all variables become names of reference types, while procedure names become names of replicated types. This correspondence becomes important when we design proof systems for this language.

While we can understand these commands without typing them, annotation by types offers a clean structural understanding of these syntactic entities which becomes particularly useful, in fact even indispensable, when its syntactic structures become complex, as in the case when higher-order procedures are involved. Milner said: "types are the leaven for programming languages — it makes them digestible". How well-said this is, especially when our enterprise is of the kind which demands a careful attention to each syntactic entity and its interplay with others. We use the following types.

$$\alpha \quad ::= \quad \mathbb{B} \mid \mathbb{N} \mid \vec{\alpha} \overset{\Upsilon}{\Rightarrow} \beta$$

$\vec{\alpha} \overset{\Upsilon}{\Rightarrow} \beta$ is a type for a procedure: Υ is a map from variables to types, showing possible reference names a procedure may touch. We consider the following three kinds of sequents. Let $\Gamma, \ldots$ indicate a map from procedure labels to arrow types, and Υ a map from variables to any types.

$$\Gamma;\Upsilon \vdash e : \alpha \text{ (expression)}, \quad \Gamma;\Upsilon \vdash C \text{ (command)}, \quad \Gamma;\Upsilon \vdash C : \alpha \text{(procedure body)}.$$

Typing rules are given in Figure 35. In each rule, the well-formedness of types and effects are assumed, i.e. the same reference name should always be given the same type in all effects occurring in the sequent. In (var) we assume x is in the domain of Ψ. In (proc-call), the references Υ which a procedure may touch (and perhaps more) should be included in the conclusion. The rules (return), (seq-ret) and (if-ret) are for procedure bodies. In (proc-dec), all procedure names of declared procedures are bound.

As a simple example, the following command returns a procedure p_1 or p_2 depending on the value of a variable x.

$$x:\mathbb{N}, p_1:\mathbb{N}\Rightarrow\mathbb{N}, p_2:\mathbb{N}\Rightarrow\mathbb{N} \vdash \mathtt{if}\; x \geq 0\; \mathtt{then}\; \mathsf{return}\; p_1\; \mathtt{else}\; \mathsf{return}\; p_2 \;:\; \mathbb{N}\Rightarrow\mathbb{N}$$

Assertions. Formulae are from the standard Hoare Logic, except we use partial application for higher-order procedures. This requires us to use only admissible formulae, whose definition precisely follows the one given in §5.1, Page 51. Terms and formulae are given as follows. $\star$ ranges over $\{\wedge, \vee, \supset\}$ and $\mathcal{Q}$ over $\{\forall, \exists\}$.

$$
\begin{array}{lcl}
t & ::= & x \mid i \mid p \mid \mathsf{c} \mid t+t \mid \ldots \\
A & ::= & t_1 = t_2 \mid \neg A \mid A_1 \star A_2 \mid \mathcal{Q}i.A \mid \mathcal{Q}x.A \mid \langle\!\langle \{C\} p \bullet \vec{t}\{C'\}/i \rangle\!\rangle A
\end{array}
$$

Note we lack the dereference. We write $x, y, \ldots$ for program variables, p, q, for procedure lables, and $i, j, \ldots$ (also $u, v, \ldots$) for other names. c is a constant. $\langle\!\langle \{C\} p(\vec{t})\{C'\}/i \rangle\!\rangle A$ is a stateful partial evaluation. We set:

$$\{C\}p(\vec{t}) \rightsquigarrow^{\mathsf{p}} t'\{C'\} \quad \equiv \quad \langle\!\langle \{C\}p(\vec{t})\{C'\}/i \rangle\!\rangle (i = t')$$

The judgements have the three forms.

- $\{C\}A \vdash e :_u B\{C'\}$ for expressions.
- $\{C\}A \vdash P\{C'\}$ for commands.
- $\{C\}A \vdash P :_u B\{C'\}$ for procedure bodies.

Formally we annotate e and P with their types: in practice we usually omit them. The typing of a judgement is given as follows.

> If $\Gamma, \Delta \vdash e : \alpha$ such that Γ is to variables and Δ is to procedure labels, then, in $\{C\}A \vdash e :_u B\{C'\}$, the primary names of C and C' are $\mathsf{dom}(\Gamma)$, those of A are $\mathsf{dom}(\Delta)$, and that of B is the anchor u.

Similarly for the other two forms of judgements. As an example of assertions, we can write down a specification similar to the one given in (10.11) as follows. Below let $C \stackrel{\text{def}}{=} \forall i, n.\{w = n\}x \bullet i \rightsquigarrow^{\mathsf{p}} 2 \times i\{w = n + 1\}$.

$$\{C \wedge y = 3 \wedge w = 0\}\, \mathsf{T} \vdash \ y := x(y)\ \{C \wedge y = 6 \wedge w = 1\}$$

Except the assertion given in C, the judgement is just like that of Hoare's Logic. Another simple example follows, using the same C as above.

$$\{\mathsf{T}\}\, C[p/x] \vdash \ x := p\ \{C\}$$

which transfers the rely condition to the post-condition by assignment.

Proof Rules The proof rules are given in Figure 36 (for expressions) and Figure 37 (for commands) and Figure 38 (for procedure bodies). The rules in Figure 36 use named expressions, uniformly treating first-order/higher-order side-effect-contaminated/side-effect-free expressions. (add) uses a notation "e^{Υ}" which indicates Υ is the variables part of the base of e. This annotation is used for making the rule evaluation-order independence. Hoare's original rule, in which the side effects are prohibited, is the case when $\Upsilon_1 = \Upsilon_2 = \emptyset$. The same idea is applicable to all other expressions. In (proc-call), we only consider procedures with a single

$$\text{(var)}\ \frac{C \supset B[x/u]}{\{C\}\, A \vdash x :_u B\, \{C\}} \qquad \text{(num)}\ \frac{-}{\{C\}\, A[\mathsf{n}/u] \vdash \mathsf{n} :_u A\, \{C\}}$$

$$\text{(add)}\ \frac{\{C^{\Upsilon} \wedge C_i^{\Upsilon_i}\}\, A^{\Delta} \vdash e_i^{\Upsilon,\Upsilon_i} :_{u_i} E_i\, \{C^{\Upsilon} \wedge C_i'^{\Upsilon_i}\} \quad E_1 \wedge E_2 \supset B[u_1 + u_2/u]}{\{C \wedge (\wedge_i C_i)\}\, A \vdash e_1 + e_2{}^{\Upsilon,\Upsilon_1,\Upsilon_2} :_u B\, \{C \wedge (\wedge_i C_i')\}}$$

$$\text{(proc-call)}\ \frac{\{C\}\, A \vdash e :_v E\, \{D\} \quad A \wedge E \supset \langle\!\langle \{D\} p \bullet v \{C'\}/u \rangle\!\rangle B}{\{C\}\, A \vdash p(e) :_u B\, \{C'\}}$$

Fig. 36. Proof Rules for an Higher-Order Imperative Language (1) Expression

argument: the treatment can be easily extended to the general case, incorporating the evaluation-order independent as in (add) rule. It is instructive to see its simplified version with neither an argument nor a return value.

$$\text{(proc-call-bare)}\ \frac{A \ \supset\ \{C\} p \bullet \varepsilon \{C'\}}{\{C\}\, A \vdash p(\varepsilon)\, \{C'\}}$$

Above $\{C\}p \bullet \varepsilon\{C'\}$ is understood in the obvious way. This rule gives (essentially) one half of Hoare's proof rule for procedure — it assumes that we have already inferred the property of a procedure p, and, assuming it is given in A, gives what would happen if p is ever called. The rule (proc-call) generalises this so that it can treat the cases when a procedure uses other (possibly higher-order) procedures and may further return a procedure of a certain type.

Among the rules in Figure 37, each of (assign), (if) and (while) becomes the corresponding rule in the standard Hoare logic when e in the rule is "simple" in the sense of Definition 11, Page 150, i.e. if it has no procedure calls in it. The rule is easily justifiable via encoding into the partial open state logic using the recursion rule. For inferring the properties of procedures, we divide the original typing rule (proc-dec) into two, using the following grammatical categories for simplicity.

$$\mathcal{P} \quad ::= \quad \mathsf{proc}\ p(\vec{y})P$$

The symbol $\mathcal{P}$ stands for a stand-alone procedure definition, before being declared as part of a command. A procedure declaration can then be written as $\{\mathcal{P}_i\}$ in P, with each procedure name of $\mathcal{P}_i$, say p_i, being mutually distinct. The original single typing rule is decomposed as:

$$\text{(proc)}\quad (\Gamma(p) = \vec{\alpha} \overset{\Upsilon}{\Rightarrow} \beta)\ \frac{\Gamma\,;\, \Upsilon, \vec{y} : \vec{\alpha} \ \vdash\ P : \beta}{\Gamma \vdash \mathsf{proc}\ p(\vec{\alpha}\ \vec{y})P} \qquad \text{(proc-dec)}\quad (p_i \neq p_j \text{ if } i \neq j)\ \frac{\Gamma \ \vdash\ \mathcal{P}_i^{p_i} : \Gamma(p_i) \quad \Gamma; \Upsilon \ \vdash\ Q}{\Gamma/\vec{p}; \Upsilon \vdash \{\mathcal{P}_i\}\ \mathsf{in}\ Q}$$

$$
\textsf{(skip)}\quad \frac{-}{\{C\}\,A \vdash \texttt{skip}\,\{C\}}
\qquad
\textsf{(assign)}\quad \frac{\{C\}\,A \vdash e :_v B\,\{D^{\text{-}x}\} \quad D \wedge B[x/v] \supset C'}{\{C\}\,A \vdash x := e\,\{C'\}}
\qquad
\textsf{(seq)}\quad \frac{\{C\}\,A \vdash P_1\,\{E\} \quad \{E\}\,A \vdash P_2\,\{C'\}}{\{C\}\,A \vdash P_1; P_2\,\{C'\}}
$$

$$
\textsf{(if)}\quad \frac{\{C\}\,A \vdash e :_v B\,\{E\} \quad \{E \wedge B[\mathsf{T}/v]\}\,A \vdash P\,\{C'\} \quad \{E \wedge B[\mathsf{F}/v]\}\,A \vdash Q\,\{C'\}}{\{C\}\,A \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q\,\{C'\}}
\qquad
\textsf{(while)}\quad \frac{\{C\}\,A \vdash e :_v B\,\{C'\} \quad \{C' \wedge B[\mathsf{T}/v]\}\,A \vdash P\,\{C\}}{\{C\}\,A \vdash \texttt{while}\ e\ \texttt{do}\ P\,\{C' \wedge B[\mathsf{F}/v]\}}
$$

$$
\textsf{(var-dec)}\quad \frac{\{D^{\text{-}x} \wedge E^{x}\}\,A \vdash P\,\{C'^{\text{-}x}\} \quad \{C\}A \vdash e :_x \{D\}}{\{C\}\,A \vdash \mathsf{var}\,\alpha\,x := e\ \mathsf{in}\ P}
$$

$$
\textsf{(proc)}\ \ (\forall \vec{y}\vec{i}.\,(D \supset \langle\!\langle \{C\}p \bullet \vec{y}\{C'\}/u \rangle\!\rangle B) \supset E)\quad \frac{\{C^{\text{-}\vec{y}} \wedge D^{\vec{y}}\}\,A \wedge E^{\text{-}\vec{i}} \vdash P :_u B\,\{C'^{\text{-}\vec{y}}\}}{A \vdash \mathsf{proc}\ p(\vec{y})P\ :\ E}
\qquad
\textsf{(proc-dec)}\ \ (\textstyle\bigwedge_i E_i \supset E,\ )\quad \frac{A^{\text{-}\vec{p}} \wedge E^{\vec{p}} \vdash \mathcal{P}_i : E_i^{p_i} \quad \{C\}\ A \wedge E \vdash Q\ \{C_i'\}}{\{C\}\ A \vdash \{\mathcal{P}_i\}\ \mathsf{in}\ Q\ \{C'\}}
$$

Fig. 37. Proof Rules for an Higher-Order Imperative Language (2) Command

In the first typing rule, Υ are "folded" into the arrow type, just as in the typing of a stateful replication. In the second typing rule, we demand the procedure name in each $\mathcal{P}_i$ to be distinct from others. This decomposition conforms to the practice in many programming languages, where procedures are defined independently, whose identifiers are used in other parts of programs.

With these preparations, (proc) in Figure 37 derives the property of a procedure from that of its body. Note the rule may assume some property of p in A. (proc-dec) locally declares procedures in Q allowing mutual recursion. The use of admissible formulae is essential for the soundness of the rule. This rule can safely be augmented with the side condition, $A \supset \exists\vec{p}.E$: however this is not necessary since $\exists\vec{p}.E$ is always valid. This is because the partial substitution in (proc) is always valid for the least defined p. The rule is directly justifiable by (Rec) in (Figure 8, §5.4, Page 56) combined with (In) in (*op. cit.*) using the encoding of multi-letrec in (6.10), §6.5, Page 73, where the encoding of an abstraction of reference names in a procedure is translated into the abstraction of

$$\dfrac{\begin{array}{l}(\mathsf{return})\\ \{C\}A \vdash e :_v B\{C'\}\end{array}}{\{C\}A \vdash \mathsf{return}\ e :_v B\{C'\}} \qquad \dfrac{\begin{array}{l}(\mathsf{seq'})\\ \{C\}A \vdash P_1\{E\}\\ \{E\}A \vdash P_2 :_v B\{C'\}\end{array}}{\{C\}A \vdash P_1; P_2 :_v B\{C'\}} \qquad \dfrac{\begin{array}{l}(\mathsf{if'})\\ \{C\}A \vdash e :_v B\{E\}\\ \{E \wedge B[\mathsf{T}/v]\}A \vdash P :_v B\{C'\}\\ \{E \wedge B[\mathsf{F}/v]\}A \vdash Q :_v B\{C'\}\end{array}}{\{C\}A \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q :_v B\{C'\}}$$

Fig. 38. Proof Rules for an Higher-Order Imperative Language (3) Procedure Body

non-reference names combined with assignment of their values to local variables (which is typable within the open stateful typing).

Let us relate these rules to the standard rule for recursive procedure [46], deriving a rule which is the dual of (proc-call-bare) given before. We strip off arguments/values as before, and, for further simplicity, assume the rule is applied at the last step of a derivation of a closed program (a program is *closed* if it does not have a free procedure name).

$$\dfrac{\begin{array}{l}(\mathsf{proc\text{-}bare}) \quad (\ \bigwedge_i \{C_i\}p_i \bullet \varepsilon\{C'_i\} \supset E)\\ \{C_i\}\, E \vdash P_i\, \{C'_i\}, \quad \{C\}\, E \vdash Q\ \{C'_i\}\end{array}}{\{C\}\ \mathsf{T} \vdash \{\mathsf{proc}\ p_i()P_i\}\ \mathsf{in}\ Q\ \{C'\}}$$

In this way, the present rule subsumes the original proof rule. Some discussions on the present proof rules for procedures would be due.

Remark 12. One of the main technical differences of the present treatment of procedures from those found in the standard Hoare-like axiom systems is its treatment of specifications for procedures as part of logical formulae, such as:

$$\{x = 3\}\, \forall n.(\{x = n\}p \bullet \varepsilon \rightsquigarrow^{\mathsf{p}} n\{x = n + 1\}) \vdash x := p() + 1; x := p() + 1;\ \{x = 7\}$$

As seen above, this necessitates a non-standard predicate, partial substitution, as well as its restriction to positive positions by admissibility (note however the inference which only use admissible formulae is not at all restrictive in comparison with the standard Hoare logic with procedure, since the standard case corresponds to the case when procedure bodies are always used alone — hence positively). At the same time, it allows consistent extensions to higher-order procedures, including the direct treatment of their side effects in reasoning. Another merit of the present approach may be that the use of quantifiers and weakening on such formulae gives us a natural analogue of the adaptation rules.

Finally the rules in Figure 38 are for procedure body. In (return), notice the evaluation of e can have side effects. Other two rules should need no illustration. We also have the structural rules (among others the consequence rules for each of the three forms of the judgement), which we omit.

Discussions. Apart from (and related with) the extension to higher-order procedure, one of the main interests of the proof rules is its uniform treatment of "expressions with side effects", which is the standard practice in modern programming languages. Our intention is to experiment with reasoning principles for imperative programming using the ideas from the process theory, in particular the typed process logic. From the viewpoint of processes, it is natural for a replicated process to affect other processes before returning an answer at a linear channel: since the process logic can assert on properties of such behaviour directly, we may as well adopt the framework to the same class of behaviour as realised in programming languages. We believe the present approach would be worthwhile to pursue for the following reasons.

- It leads to a fully compositional program logic for languages with objects, higher-order procedures and mutual data structures, where the behaviour of a program is directly asserted and reasoned with no need of rewriting (except when a program is written in such a way that it depends on evaluation-order, which is anyway a practice which is not recommendable). This has an advantage that it offers a structural understanding of the behaviour of expressions/programs as prepared by programmers.
- When we restrict programs to the setting which the standard rules treat (side-effect free expressions and first-order procedures), we regain the standard reasoning rules. Thus we maintains all standard reasoning principles while extending the applicability of program logics to more complex cases.
- The use of types in reasoning can be cumbersome in practice: however, modern programming languages are often themselves typed extensively, using the type disciplines as basic structuring principles. For such languages, program logics based on types are a natural medium to describe properties of programs, offering powerful methods for structural reasoning on programs which types alone cannot achieve

For the third point, the notion of types (not necessarily in its syntactic presentation) lies at the heart of the present logical method. But the same idea can, in hindsight, be found in the original Hoare's logic. We shall examine these points in our investigation in the remaining sections.

11 Logics for Sequential Processes with Local State

11.1 Behaviour with Local State

We recall, from our discussions in Section 9.1, that a notion of local state arises when a reference, which may be repeatedly used by a replicated process, is hidden, or localised. The following gives a typical instance of such behaviour (we assume value passing communication as we did in Section 9.7).

$$P \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(R \,|\, \mathsf{Ref}\langle y0\rangle) \tag{11.1}$$

where we set:

$$R \stackrel{\text{def}}{=} !x(z).\overline{y}\mathsf{read}(c)c(n).\overline{y}\mathsf{write}(c')c'(n+1).\overline{z}(n+1). \tag{11.2}$$

Here n denotes a natural number as a constant (which can easily be translated into branching). P increments its local counter (the reference at y) whenever it receives an invocation, as well as sending out the new counter value: thus it first sends 1, then 2, then 3, and so on. Note its behaviour is far from functional: each time it is invoked, it has a different reaction. Because the source of this distinct reaction — the state $\mathsf{Ref}\langle yn\rangle$ for some n — is hidden, the behaviour would look enigmatic to the outside observer.

The situation gets complicated if a program needs to *use* such a stateful process. For example, consider

$$Q \stackrel{\text{def}}{=} \overline{x}(z)z(n).\overline{x}(z')z'(n').\overline{w}\langle n+n'\rangle. \tag{11.3}$$

The behaviour of Q relies on the behaviour of the environment located at x. For example, if the environment is a simple replication, then we know that it would return the same value for all invocations: thus if it returns 1 at first, it would return 1 again at the second time: therefore, at w, Q would return the double of some natural number. But if the environment is a process with a local state, the first return value and the second one may not be identical, due to the hidden state invisible to Q. As another related example, consider a situation where two processes have a shared local state, as in:

$$S \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(R[x_1/x] \,|\, R[x_2/x] \,|\, \mathsf{Ref}\langle y0\rangle) \tag{11.4}$$

where R is the same process as defined in (11.2). If another process interacts with this process at x_1 at this state, it would get 1; the same thing happens it first interacts at x_2, receiving 1. But if a process interacts at x_1 and x_2 one by one (in whichever order), as done by:

$$Q' \stackrel{\text{def}}{=} \overline{x_1}(z)z(n).\overline{x_2}(z')z'(n').\overline{w}\langle n+n'\rangle. \tag{11.5}$$

then the interaction at x_2 returns 2, not 1, since the initial action of Q' at x_1 influences the behaviour of S at x_2 by the write to the locally shared reference.

$$
\begin{array}{l}
(\mathsf{Hide}) \quad (\mathsf{md}(\Gamma(x)) \in \{\mathbf{!}, \mathbf{!}_{rw}\}) \\
\vdash P \triangleright \Gamma \\
\hline
\vdash (\boldsymbol{\nu}\, x)P \triangleright \Gamma/x
\end{array}
$$

Fig. 39. Typing Rule for Local Reference

Further subtleties associated with the behaviours with local state arise when we consider those behaviours which generate such behaviours.

The typing for sequential processes with local state generalises the original (Hide-global) rule to (Hide) rule given in Figure 39, which also subsumes (Res) rule. In the rule, Γ/x takes off the typing for x including its usage as effects from Γ. Other than this extension, processes, their dynamics and typing remain precisely the same as before. Simple as the change may look, the resulting class of process behaviour is quite different from the original one, whose glimpse we have seen above. The purpose of the remaining part of this section is to investigate how we can reach a simple and powerful reasoning principle for this enlarged class of behaviours.

11.2 Assertions for Sequential Processes with Local State

Assertions with Local State. Given the additional complexity incurred by the introduction of local state, we may as well aim to articulate this complexity incrementally, that is, without adding anything except what would be in accordance with the syntactic increment, in this case the general hiding of references. How can we achieve this? Let us start from what we have, that is assertions for open state. At this point we do not know how to specify the behaviour of P in (11.1), but if we take off its restriction, we obtain R in (11.2), for which we surely know how to assert (note R is indeed typable in the type discipline in Section 9). A specification of R in the open stateful logic may be given by the following formula:

$$A(x,y) \stackrel{\text{def}}{=} \forall n.\{!y = n\}\, x \bullet \varepsilon \searrow (n+1)^{\uparrow}\, \{!y = n+1\} \tag{11.6}$$

Now let us consider how can we specify the behaviour of P, which has a local state. It has an interaction point x, via which we wish to specify its behaviour as a logical predicate. The behaviour is not functional, in the sense that, even with the same null argument, it returns different numbers at different occasions. But this is because it has a local state hidden by y: if we peal off this hiding, then its behaviour is a familiar open stateful behaviour, plain and simple. Further we observe, in addition to this open stateful behaviour, P also has its local state $\mathsf{Ref}\langle y0\rangle$, which will be updated at each invocation. Note the open stateful behaviour of R together with its (initial) local state $\mathsf{Ref}\langle y0\rangle$ completely describes the behaviour of P, together with the information that y is hidden.

These observations suggests that we may describe the behaviour of P using two components, an open stateful behaviour and a hidden state, combined with information on hidden names, leading to the following predicate. Below i ranges over natural numbers.

$$B(x,i) \stackrel{\text{def}}{=} \mathsf{open}\ x\ \mathsf{as}\ (\boldsymbol{\nu} y)\,\underline{x}\ \mathsf{in}\ (A(\underline{x},y) \wedge\, !y = i) \tag{11.7}$$

The predicate $B(x,i)$ is meant to specify a behaviour which has:

1. an opaque behaviour at x, hiding a state (reference) with subject y.
2. Under its cloak, it hides a local reference y with the current local state 0 of y, as well as an open stateful behaviour given by A.

The formula in (11.7) is called *opening formula for x*. In this formula, the occurrences of y and $\underline{x}$ after "in" are bound by the binders $(\boldsymbol{\nu}\,y)\underline{x}$. The underlined name $\underline{x}$, considered to be distinct from x, emphasises that it is now treated as the open behaviour (formally it can be any fresh name). Note that we can infer, from this description, that, at an initial invocation it would answer with 1, with the resulting state $B(x,1)$; the next time it is invoked it would answer with 2, with the resulting state $B(x,2)$, and so on. We note the formula can also be written, using an abbreviation:

$$\mathsf{open}\ (\boldsymbol{\nu} y)x\ \mathsf{in}\ (A(x,y) \wedge\, !y = i)$$

which is less flexible as syntax (for example we should prohibit opening x again after it is opened) but which is surely more concise (we note we cannot abbreviate this further by omitting the opened name x, which will become clear when the semantics of opening formulae discussed, see §11.5, Page 173). In this subsection, however, we shall stick to the formal notation to avoid confusion.

Using $B(x,i)$, we can now specify the behaviour of Q in (11.3) under the assumption that the (initial) behaviour of the environment is given in $B(x,0)$, as follows.

$$Q \vdash \{B(x,0)\}\,\mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\, w = (3)^{\uparrow}\ \{B(x,2)\} \tag{11.8}$$

Note the state change for an opaque behaviour is expressed by the combination of pre/post conditions. We shall later see how such an assertion is derived on the basis of the reasoning principles for open stateful logic.

Array. As further examples of simple usage of opening formulae, let us see how we can specify a simple instance of a standard data structure with hidden state, an array of natural numbers. A specification for such an array of length 3 can be written as, using replicated branching (which is easily reducible to the original system, both behaviourally and logically, but which is useful for tractable encodings):

$$T \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, x_0x_1x_2)(!a[(nc).Read\ \&\ (nvc).Write] \mid \Pi_{0\le i\le 2}\mathsf{Ref}\langle x_i n_i\rangle) \tag{11.9}$$

where we set, using a short-hand notation and assuming, for simplicity, that the array diverges when the array bound is violated:

$$\begin{aligned}
Read &\stackrel{\text{def}}{=} \texttt{if } 0 \le n \le 2 \texttt{ then } \overline{x_n}\mathsf{read}(e)e(m).\overline{c}\langle m\rangle \texttt{ else } \omega_c \\
Write &\stackrel{\text{def}}{=} \texttt{if } 0 \le n \le 2 \texttt{ then } (\boldsymbol{\nu}\, e)\overline{x_n}\mathsf{write}\langle ve\rangle e(m).\overline{c}\langle m\rangle \texttt{ else } \omega_c
\end{aligned}$$

We may write down a specification for T as follows.

$$Ar \quad \stackrel{\text{def}}{=} \quad \mathsf{open}\ a\ \mathsf{as}\ (\boldsymbol{\nu} x_1x_2x_3)\underline{a}\ \mathsf{in}\ \forall 0 \le n \le 2.\ (readspec \wedge writespec)$$

where we set:

$$\begin{aligned}
readspec &\stackrel{\text{def}}{=} \{\wedge_i !x_i = m_i\}a \bullet \mathsf{read}(n) \leadsto (m_n)^{\uparrow}\{\wedge_i !x_i = m_i\} \\
writespec &\stackrel{\text{def}}{=} \{\wedge_i !x_i = m_i\}a \bullet \mathsf{write}(nm') \leadsto ()^{\uparrow}\{\wedge_{i\neq n} !x_i = m_i\ \wedge\ x_n = m'\}.
\end{aligned}$$

For a reader of an array, we may assert, assuming $A \supset\ !x_1 = v$:

$$\begin{aligned}
\overline{x}\mathsf{read}\langle 1, c\rangle \vdash\ & \{\mathsf{open}\ a\ \mathsf{as}\ (\boldsymbol{\nu} x_0x_1x_2)\underline{a}\ \mathsf{in}\ A\} \\
& \textbf{rely}\ \mathsf{T}\ \ \textbf{guar}\ c = (v)^{\uparrow} \\
& \{\mathsf{open}\ a\ \mathsf{as}\ (\boldsymbol{\nu} x_0x_1x_2)\underline{a}\ \mathsf{in}\ A\};
\end{aligned}$$

while, for a writer to an array:

$$\begin{aligned}
\overline{x}\mathsf{write}\langle 0, v, c\rangle \vdash\ & \{\mathsf{open}\ a\ \mathsf{as}\ (\boldsymbol{\nu} x_0x_1x_2)\underline{a}\ \mathsf{in}\ A[v/!x_0]\} \\
& \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ c = ()^{\uparrow} \\
& \{\mathsf{open}\ a\ \mathsf{as}\ (\boldsymbol{\nu} x_0x_1x_2)\underline{a}\ \mathsf{in}\ A\}.
\end{aligned}$$

These assertions are analytical in nature, stipulating the behaviour by unfolding it one by one. As we have repeatedly seen in the previous sections, one may as well have more concise notation when the shape of behaviour is fixed. In Section 12 we shall see how the above assertion precisely corresponds to the standard treatment of array in Hoare Logic, as initially given in [50].

Shared Local State. Next we consider when a local state is shared by two or more processes, taking the process S in (11.4) in §11.1 as an example. The following specification of S is a straightforward generalisation of the preceding opening formula, where we simultaneously open the subjects of two replicated processes that share state.

$$C(x_1x_2, i) \stackrel{\text{def}}{=} \mathsf{open}\ x_1x_2\ \mathsf{as}\ (\boldsymbol{\nu} y)\underline{x_1x_2}\ \mathsf{in}\ \ (B(\underline{x_1}, \underline{x}) \wedge B(\underline{x_2}, \underline{x}) \wedge !y = i) \quad (11.10)$$

Once we have $C(x_1x_2, 0)$, we can infer observable properties such as:

1. The process would return 1 when it is invoked for the first time, may it be at x_1 or x_2;

2. At whichever name the process is invoked first, the next time it is again invoked (again at whichever name), it would return the increment of the initial return value.

On the other hand, we can describe the behaviour of the client process Q' in (11.11) when it is combined with S as follows.

$$Q' \vdash \{C(x_1x_2, 0)\}\,\mathbf{rely}\,\mathsf{T}\ \mathbf{guar}\,w = (5)^{\uparrow}\ \{C(x_1x_2, 2)\} \tag{11.11}$$

which does describe how Q' would behave when composed with S.

As a related example, suppose we have the following process, where two reference agents store a hidden opaque behaviour.

$$S' \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,x)(\,\mathsf{Ref}\langle z_1x\rangle \mid \mathsf{Ref}\langle z_2x\rangle \mid R \mid \mathsf{Ref}\langle y0\rangle\,) \tag{11.12}$$

The specification for S' may be written down as follows.

$$\mathsf{open}\ z_1z_2\ \mathsf{as}\ (\boldsymbol{\nu}y)\underline{z_1z_2}\ \mathsf{in}\ \exists x.\,(\,!\underline{z_1}=x \wedge !\underline{z_2}=x \wedge A(x,y) \wedge !y=0\,)$$

where $A(x,y)$ is the predicate which already appeared in (11.6) before. The predicate is directly about z_1 and z_2, saying their content is — if we make the local state y open — the behaviour which satisfies A together with its initial value y. Note x, which is a non-opaque name, is hidden by an existential quantifier, while the local reference y is hidden by an opening formula, similar to (11.10).

Exporting Opaque Names. A process may export opaque names to the outside, whose logical treatment involve certain subtleties. First let's consider the following process, which outputs an opaque name via a linear output.

$$T \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,xy)(\overline{w}\langle x\rangle \mid R \mid \mathsf{Ref}\langle y0\rangle) \tag{11.13}$$

An assertion for this behaviour is as we gave before, except a linear output is opened rather than a replication.

$$E \stackrel{\text{def}}{=} \mathsf{open}\ w\ \mathsf{as}\ (\boldsymbol{\nu}y)\underline{w}\ \mathsf{in}\ \ (\,\exists c.\,(\,\underline{w}=(c)^{\uparrow} \wedge B(c,i)\,)\ \wedge\ !y=i\,) \tag{11.14}$$

Note a linear output is opened rather than the carried replicated name c. Intuitively we are regarding the linear output to be opaque which, when opened, now reveal the internal opened state including the carried replicated name (it is awkward, both syntactically and semantically, to use an opening formula for a carried replicated behaviour). In E, the name c is regarded as non-opaque, so that it is hidden by an existential quantifier; while a local reference name y is hidden by an opening formula. Since T does nothing except outputting at w, we would have $T \vdash \{\mathsf{T}\}\,\mathbf{rely}\,\mathsf{T}\,\mathbf{guar}\,E\,\{\mathsf{T}\}$ for T.

Let us now consider a refinement of T, which further writes the same opaque name to a reference.

$$T' \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,xyc)(\overline{w}\mathsf{write}\langle xc\rangle \mid c.\overline{w}\langle x\rangle \mid R \mid \mathsf{Ref}\langle y0\rangle) \tag{11.15}$$

How can we specify the behaviour of T'? We first consider a process which is the same as T' except in which y is not hidden so that x is not opaque.

$$T'_0 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, xc)(\overline{w}\mathsf{write}\langle xc\rangle \mid c.\overline{z}\langle x\rangle \mid R \mid \mathsf{Ref}\langle y0\rangle) \tag{11.16}$$

For T'_0 (which does not have local state) we may simply assert:

$$T'_0 \vdash \{\mathsf{T}\}\, \mathbf{rely}\, \mathsf{T}\, \mathbf{guar}\, \exists c.\, (\, z = (c)^{\uparrow} \wedge A(c,y) \ \wedge\ !y = i)\, \{A(!w,y)\} \tag{11.17}$$

So all we have to do is hide y in this assertion. But notice the post-condition mentions y (or, more generally, the typing for its primary names mention y as an effect), so that, at the level of formulae, y should be hidden both in the guarantee formula and in the post condition, which cannot be done by the opening formulae. One possible solution[14] is to refine the shape of the judgement in the following way.

$$T' \vdash \{\mathsf{T}\}\, \mathbf{rely}\, \mathsf{T}\, \mathbf{guar}^{\nu y}\, \exists c.\, (\, w = (c)^{\uparrow} \wedge A(c,y) \ \wedge\ !y = i\,)\, \{A(!z,y)\} \tag{11.18}$$

Above, the hiding "νy" binds the occurrences of y in both the guarantee formula and the post condition. Intuitively, $P \vdash \{C\}\, \mathbf{rely}\, A\, \mathbf{guar}^{\nu\vec{y}}\, B\, \{C'\}$ means P has the behaviour as specified and, moreover, the references are local to P. Names in $\vec{y}$ may as well occur in B and C': however they *never* occur in C or A, since any of y_i cannot be known to the environment before it is outputted/written by the process P. We also notice an assertion $P^{\Gamma;\Delta} \vdash \{C\}\, \mathbf{rely}\, A\, \mathbf{guar}^{\nu\vec{y}}\, B\, \{C\}$ can always be rewritten into the standard assertion using an opening formula, as far as no names from $\vec{y}$ occur in B.

Generation of New References. We now consider assertions which specify properties of processes which *generate* behaviours with local state. As the first example, let us consider the following process.

$$O \stackrel{\text{def}}{=} !u(nz).\overline{z}(x)(\boldsymbol{\nu}\, y)(R \mid \mathsf{Ref}\langle yn\rangle) \tag{11.19}$$

where $R \stackrel{\text{def}}{=} !x(z).\overline{y}\mathsf{read}(c)c(n).\overline{y}\mathsf{write}(c')c'(n+1).\overline{z}(n+1)$, which is from (11.2) in §11.1, Page 162. This process generates the incrementor P which we have treated before, each time with its own (distinct) local state: in other words, a local reference name y is generated at each invocation, and each newly generated replication uses a distinct newly generated reference. It is instructive to see how it interacts with its invoker (which one may consider as the "new" statement in the object-oriented language, while O acts as a class template).

$$O \mid \overline{u}\langle 0c\rangle \quad \longrightarrow^{2} \quad O \mid \overline{c}(x)P.$$

[14] Another idea is to make the guarantee formula and the post condition into one. Because references in the postcondition are those which exist in the *environment*, while those in the guarantee formula are part of the process itself. Mixing them thus necessitates clear notational indication of polarities at occurring names.

As seen from this reduction, each time this agent is invoked, it creates and returns a behaviour which has a fresh subject name (x above) and a fresh local reference. The following assertion specifies the behaviour which is generated by O.

$$E(z,i) \stackrel{\text{def}}{=} \mathsf{open}\ z\ \mathsf{as}\ (\boldsymbol{\nu} y)\underline{z}\ \mathsf{in}\ \exists x.(\,\underline{z} = (x)^{\uparrow} \wedge A(x,y) \wedge\ !y = i\,) \tag{11.20}$$

Let us write $E'(z,y,i)$ for the body part of $E(z,i)$ above, that is:

$$E'(z,y,i) \stackrel{\text{def}}{=} \exists x.(\,z = (x)^{\uparrow} \wedge A(x,y) \wedge\ !y = i\,)$$

Then we may specify the behaviour of the output of the opaque behaviour using the refined judgement we have seen above:

$$\overline{z}(x)(\boldsymbol{\nu}\, y)(R \mid \mathsf{Ref}\langle yn\rangle) \vdash \{\mathsf{T}\}\,\mathbf{rely}\ \mathsf{T}\ \mathbf{guar}^{\nu y}\, E'(z,y,i)\,\{\mathsf{T}\}$$

From this we arrive at the specification of the process O.

$$O \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ [u \bullet i/(\nu y)z]\, E'(z,y,i) \tag{11.21}$$

which precisely reflects the shape of refined assertions. Note y and z both bind the body formula $E'(z,y,i)$. As a refinement, an assertion for the following process:

$$O' \stackrel{\text{def}}{=}\ !u(nc).T' \tag{11.22}$$

for T' we have treated already, would become

$$O \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \forall i.(\{\mathsf{T}\}\{u \bullet i\}[y/(\nu z)\, E'(z,y,i)]\{\{\}\}A(!z)). \tag{11.23}$$

Extensionality and Local State. We close this subsection with the discussions on one significant aspect of assertions for behaviour with local state. One of the notable engineering merits of the use of local state in programming is the freedom with which we can realise identical observable behaviours by different implementations, each of which uses a different internal structure but exhibits the same behaviour to the outside. From a logical viewpoint, however, this means an assertion which is valid with respect to the extensional behaviour of programs/processes can be hard to infer from their syntactic structures. In fact, while the assertions we have treated so far precisely correspond to the syntactic structures of given processes, this is not always the case, as can be seen from the following two variants of P treated above.

$$P_1 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, yz)(R \mid \mathsf{Ref}\langle y0\rangle \mid \mathsf{Ref}\langle z1\rangle) \tag{11.24}$$

$$P_2 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(U \mid \mathsf{Ref}\langle y1\rangle) \tag{11.25}$$

where R and U are given by:

$$R \stackrel{\text{def}}{=}\ !x(z).\overline{y}\mathsf{read}(c)c(n).\overline{y}\mathsf{write}(c')c'(n+1).\overline{z}(n+1). \tag{11.26}$$

$$U \stackrel{\text{def}}{=}\ !x(z).\overline{y}\mathsf{read}(c)c(n).\overline{y}\mathsf{write}(c')c'(n+1).\overline{z}(n). \tag{11.27}$$

Note all of P, P_1 and P_2 behave precisely in the same way to the outside observer. However the syntactic structures of P_1 and P_2 do not exactly match $B(x, 0)$. Let us write down a straightforward specification for P_1:

$$B'(x,i) \stackrel{\text{def}}{=} \mathsf{open}\ x\ \mathsf{as}\ (\boldsymbol{\nu} yz)\,\underline{x}\ \mathsf{in}\ (A(\underline{x}, y) \wedge !y = i \wedge !z = 0) \tag{11.28}$$

In fact, in the of assertions which we shall stipulate later, $B(x, i)$ and $B'(x, i)$ are logically equivalent — they are inhabited by precisely by the same set of processes. An easy way to calculate is the entailment in the following direction:

$$B(x,i) \qquad \supset \qquad B'(x,i) \tag{11.29}$$

Indeed, if a process satisfies $B(x.i)$, then it may as well have a redundant state at z (for which no constraint is given), so that it would also satisfy $B(x, i)$ (this is in fact a general rule). Let us use this entailment for the specification of Q in its composition with S. Since Q satisfies (11.8), the entailment (11.29) gives us:

$$P \vdash \{B'(x,0)\}\,\mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\, w = (3)^{\uparrow}\ \{B(x,2)\} \tag{11.30}$$

so that we know:

$$(\boldsymbol{\nu}\, x)(S|Q) \vdash \{\mathsf{T}\}\,\mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\, w = (3)^{\uparrow}\ \{\mathsf{T}\} \tag{11.31}$$

as expected. While such an inference may often be done mechanically (some of the basic laws would be discussed later), in some cases it necessitates proving the equivalence of behaviours, for which we can use, among others, transition-based reasoning (we note the extensional equality for behaviours with local state is completely characterisable by a refined typed transition).

11.3 A Logic for Local State (1): Language

The logical description of stateful behaviour in the present logic is based on, as we have seen in the preceding subsection, stratification of behaviours into those with open state and those with local state, the latter being given in terms of the former. Thus we need to treat two kinds of behaviours in assertions. To distinguish them clearly in formulae, we use two distinct type annotations for terms except for those of reference types. If we write $a^{[\tau]}$, then a is of an *opaque type*, while if we write a^{τ} then a is of an *open type*, which, among others, should be annotations for names occurring in formulae.[15] We sometimes use the underlined name (e.g. $\underline{x}$) to emphasise it is open. As we shall make clear in the following, it is *not* the case that we can use opaque terms and open terms interchangeably: the well-typedness assumption strictly regulates their usage.

[15] The distinction between opaque names and open names is only effective in formulae; processes are still typed with the (corresponding) standard channel types. In fact, even in formulae, we can infer the former from the latter once a formula and its expected typing is given, which would be useful in computer-assisted verification process.

We use the same set of terms as before, except names may now be annotated with opaque types. The well-typedness of these terms is inherited from the open stateful logic, in the sense that the stipulated operators such as $\bullet$ are only meaningful for open terms. This convention is essential since it allows us to stipulate the semantics of operations precisely following that of the open stateful logic without any change. Among others we stipulate:

- Let a be of a reference type. Then $!a$ should be open: that is, we can only store an open behaviour to a reference.
- $\vec{a}$ should be open in $\mathtt{in}_e^{\rho}(\vec{a})$ (that is, a linear output is specified as it carries an open behaviour, which we may later turn to be opaque).
- In $\{C\}[a \bullet \vec{b}/(\nu\vec{y})z]A\{C'\}$, a and $\vec{b}$ should be open (that is, invocation is first specified for open replications, which may be made opaque later), and z should be open in A.

The reader may check examples in §11.2 do adhere to these conditions, which are central to the stratification of the stateful logic. In particular, at the semantic level, this allows us to build validity of formulae on the basis of the one in the open state logic (cf. §9.5) : syntactically they lead to simple proof rules for local state on the basis of those for the open state.

Formulae are augmented as:

$$A \ ::= \ \ldots \ \mid \ \mathsf{open}\ \vec{x}^{[\vec{\tau}]}\ \mathsf{as}\ (\boldsymbol{\nu}\vec{y}^{\vec{\delta}})\vec{w}^{\vec{\rho}}\ \mathsf{in}\ A \ \mid \ \{C\}[a^{\tau} \bullet \vec{b}^{\vec{\rho}}/(\nu\vec{y}^{\vec{\delta}})z^{\sigma}]A\{C'\}$$

In both, each δ_i is a reference type. In $\mathsf{open}\ \vec{x}\ \mathsf{as}\ (\boldsymbol{\nu}\vec{y})\vec{w}\ \mathsf{in}\ A$, we say:

- x_i is an opaque name which is *opened* in this opening formula.
- w_i (resp. y_i) is an open name (resp. a local reference) *introduced* in this opening formula.

The part before "in" is called *guard*, while A is called *body*. $\vec{y}$ and $\vec{w}$ are binders with their scope in A ($(\nu\vec{y})$ is omitted if $\vec{y}$ is empty). We assume the standard binding convention. The following well-formedness condition is always assumed.

- Names in the guard, $\vec{x}\vec{y}\vec{w}$, should be pairwise distinct, and $\vec{x}$ and $\vec{w}$ should be of the same length.
- In each ρ_i, at least one name from $\vec{y}$ occurs as an effect name, and that $\tau_i = \rho_i/\vec{y}$ for each i.

In $\{C\}[a^{\tau} \bullet \vec{b}^{\vec{\rho}}/(\nu\vec{y}^{\vec{\delta}})z^{\sigma}]A\{C'\}$, $\vec{y}z$ should be pairwise distinct and should be distinct from free names except those in A and C'. For the well-typedness, we demand, implicitly assuming a common typing for auxiliary names:

- For $\mathsf{open}\ \vec{x}^{[\vec{\tau}]}\ \mathsf{as}\ (\boldsymbol{\nu}\vec{y}^{\vec{\delta}})\vec{w}^{\vec{\rho}}\ \mathsf{in}\ A$ to be well-typed under $\Gamma, \vec{x}:[\vec{\tau}]$, the body A should be typed under $\Gamma, \vec{w}:\vec{\rho}, \vec{y}:\vec{\delta}$.
- For $\{C\}[a^{\tau} \bullet \vec{b}^{\vec{\rho}}/(\nu\vec{y}^{\vec{\delta}})z^{\sigma}]A\{C'\}$ to be well-typed under Γ, a and $\vec{b}$ are typed by τ and $\vec{\rho}$ under Γ, respectively; $\tau = (\vec{\rho}\sigma')^{!}$ such that $\sigma' = \sigma/\vec{y}$; C and C' should be each typed under $\vec{y}:\vec{\delta}$; and A should be typed under $\Gamma, z:\sigma$.

Finally we introduce judgements. Those for threads are annotated with local references, while those for non-threads stay the same as before. For typing:

- The sequent $P^{\Gamma;\Delta} \vdash \mathbf{rely}\, A\, \mathbf{guar}\, B$ is as before, where we assume primary names in A and B may or may not be open.
- The sequent $P^{\Gamma,\Upsilon;\Delta} \vdash \{C\}\, \mathbf{rely}\, A\, \mathbf{guar}^{\nu\vec{y}:\vec{\delta}}\, B\, \{C'\}$ with Γ not containing reference types, is typed as:
  1. C is typed by Υ together with part of $\overline{\Gamma}$, say Θ, for replicated names, which are used in C as opaque names.[16]
  2. A is typed by part of $\overline{\Gamma}$. Primary names in A may or may not be open.
  3. B is typed by $\Delta, \vec{y} : \vec{\delta}$. In B, all primary names should be open except reference names which should be among $\vec{y}$.
  4. C' is typed by Υ together with Θ, which are used in C' as opaque names.

  The binder "$\nu\vec{y}$" is called *hiding* of the assertion.

Keeping primary names in B to be open makes the proof rules simpler than otherwise, allowing us to make the most of the stratification of the present logic on the basis of the open stateful logic.

11.4 Names, Local State and Axioms for Equality

Before introducing semantics of assertions for local state, one topic deserves a brief discussion, the treatment of names. In the logics for stateless behaviours, all names are treated as their denotation: that is, $x = y$ is valid under a model ξ iff $\xi(x) = \xi(y)$. It is notable that this interpretation is consistent with the standard treatment of equations in the predicate logics with equality, where we should always have (cf. [73, §8]), in addition to reflexivity, the following axiom schema:

$$x = y \supset (A(x,x) \supset A(x,y)) \tag{11.32}$$

which says that any entity can be substituted for its equal without changing validity. This is the fundamental principle which we wish to maintain in our logics, both from the viewpoint of having a natural specification for sequential stateful behaviours and from the viewpoint of having a consistent logic. The interpretation of names in the stateless logic is consistent with (11.32) because having two replicated stateless agents of the same behaviour at (say) x and y is the same thing as sharing one such behaviour by coalescing x to y (or vice versa): their identification, or *distinction* as called by Milner, Parrow and Walker [84], does not matter in this setting, as far as their representative behaviour turns out to be identical.

In the open stateful logic, we continued this interpretation (which is natural if we regard names as standard variables, but is non-standard if we regard names as

[16] This and other restrictions on the usage of names do not constrain expressiveness, because all open names can be immediately made (trivially) opaque, and, conversely, any opaque primary name used in B may be opened by adding the involved hidden references to the hiding of the assertion.

interaction points representing general behaviours) except for names of reference types (denoting the subject names of reference agents). Intuitively this is because two references with an identical content for their store (say 1000 Stirling pounds) is not the same thing as having a unique reference with that content (2000 pounds and 1000 pounds are indeed different). More formally, let us consider the following specification.

$$x = y \ \supset \ (!x = !y \wedge \{!x = 1 \ \wedge \ !y = 1\} \, \overline{x}\mathsf{write}\langle 2c\rangle \rightsquigarrow c = ()^{\uparrow} \, \{!x = 2 \ \wedge \ !y = 1\}) \quad (11.33)$$

If we interpret x and y in terms of its (name-free) behaviour, then $x = y$ is the same thing as the equality of their content, so this statement seems reasonable. Yet (11.33) does *not* conform to the fundamental principle of equality in (11.32). This is because having two references (at two distinct places) do lead to different behaviour than sharing one: in the former, when you write on one, the content of that reference changes, but not the other. While in the latter this is not so. Thus the equality $x = y$ should mean x and y denote the same reference name, in which case we obtain the following statement:

$$x = y \ \supset \ (!x = !y \wedge \{!x = 1 \ \wedge \ !y = 1\} \, \overline{x}\mathsf{write}\langle 2c\rangle \rightsquigarrow c = ()^{\uparrow} \, \{!x = 2 \ \wedge \ !y = 2\}) \quad (11.34)$$

Note, under the interpretation of $x^{ref(\tau)}$ as the literal name x (or alternatively a named behaviour, which is essentially equivalent), the statement does make sense operationally. And it does too logically, conforming to the principle (11.32), which we wish to save as far as we intend the equality sign to mean equality (this is why we used a different symbol $\mathrel{\mathring{=}}$ for partial equality in Section 5).

The situation is precisely the same with opaque names: an opaque name denotes a behaviour with local state, so we cannot equate them considering them as values. Assume, for example, we have two incrementors, $P_a \stackrel{\text{def}}{=} P[a/x]$ and $P_b \stackrel{\text{def}}{=} P[b/x]$, where P is from §11.1, (11.1), Page 162, Note the corresponding abstract process is identical, which we may write p. Thus if we equate them by their denotation, we obtain:

$$P(a) \,|\, P(b) \ \models \ a = b \quad (11.35)$$

But if we allow this, we have the following analogue of (11.33) in the present setting, which can be a specification for, for example, $!y(c).\overline{a}(e)e(n).\overline{c}\langle n\rangle$.

$$a = b \ \supset \ (\{B(a,0) \ \wedge \ B(b,0)\} \, y \bullet \varepsilon \rightsquigarrow 1 \, \{B(a,1) \ \wedge \ B(b,0)\}) \quad (11.36)$$

which again does not conform to the substitutivity principle. Thus a and b above should be interpreted as distinctions: and, in that case, $a = b$ means they denote the same name, so that we reach:

$$a = b \ \supset \ (\{B(a,0) \ \wedge \ B(b,0)\} \, y \bullet \varepsilon \rightsquigarrow 1 \, \{B(a,1) \ \wedge \ B(b,1)\}) \quad (11.37)$$

Thus opaque names are again interpreted as names (or distinctions), preparing for the formal semantics of assertions in the next subsection. One may also see, from (11.34) and (11.37) above, that this interpretation gives a natural setting to discuss coalescing of names, which becomes necessary if we allow processes to pass around reference names in the final subsection.

11.5 A Logic for Local State (2): Semantics

Models with Local State. Exploiting the strict distinction between opaque names and open names, the semantics of the logic for local state can be built on the basis of the semantics of the logic for open state. Let $\cong$ be the contextual congruence defined precisely as before (we can in fact just take the observables at $()^{\uparrow}$ with the same effect, because we can now hide all references). An *abstract process* is extended so that it allows a process with multiple replicated channels. We write $p^{\vec{x}}$ for an abstract process, and instantiate it as $p(\vec{x})$. If an abstract process has a unique interaction point, as abstract processes were in the preceding sections, we say it is *pointed.*

A pointed abstract process is *open* if it has an element of the form $!x(\vec{y}z).P$ or of the form $\overline{x}\mathtt{in}_i(\vec{y})\Pi_i p_i^{y_i}$ with each p_i being open. Note an open p can generate a non-open process after invocation. We let $\underline{p}, \underline{q}, \ldots$ range over open abstract processes. Note any abstract process of replicated types can be written as (confusing concrete and abstract processes) in the way:

$$p(\vec{x}) = (\boldsymbol{\nu}\,\vec{y})(\Pi_i \underline{p_i}(x_i) \mid \Pi_j r_j(y))$$

where r_j is an abstract process of a reference type.

A *model of type Γ* is an abstract process instantiated with a vector of names whose typed channels conform to Γ. $\xi, \ldots$ range over models as before. In this section we again concentrate on total correctness, so that we only consider total models, i.e. models which do not diverge. If ξ and ξ' are models with distinct free names, then $\xi \cdot \xi'$ defines the obvious model.

We need to refine the convergence predicate for stateful invocation so that it involves local state. Let $\underline{p}$ be of type $x : (\vec{\rho}\sigma)^{!}_{\Upsilon}$ with $\Upsilon = \vec{w} : \overline{\vec{\delta}}$, $\underline{\vec{q}}$ of types $\vec{\rho}$, $\vec{r}$ of types $\vec{\delta}$, as well as $\underline{s}$ of type σ' such that $\sigma'/\vec{y} = \sigma$ and $\vec{r'}$ of types $\vec{\delta'}$ such that $\vec{\delta'}/\vec{y} = \vec{\delta}$. Then we have:

$$\langle \underline{p},\ \underline{\vec{q}},\ \vec{r}^{\vec{w}} \rangle \Downarrow \langle \underline{s},\ \vec{r'}^{\vec{w}}, \vec{t}^{\vec{y}} \rangle$$

iff, for each $P \in \underline{p}(x)$, $Q_i \in \underline{q}_i(u_i)$ and $R_j \in r_j(w_j)$, we have:

$$P \mid (\boldsymbol{\nu}\,\vec{u})(\overline{x}\langle \vec{u}z\rangle|\Pi_i Q_i) \mid \Pi_j R_j \quad \Downarrow \quad P \mid (\boldsymbol{\nu}\,\vec{y})(\ \overline{z}(c)(S|\Pi_h T_h|\Pi_j R'_j)\ )$$

where $S \in \underline{s}(c)$, $R'_j \in r'_j(w_i)$ and $T_h \in t_h(y_h)$.

Interpretation. We can now augment the interpretation of terms in the open state logic with the following two clauses.

- $[\![x^{[\tau^!]}]\!]_{\mathcal{I}\cdot\xi} \stackrel{\text{def}}{=} x$.
- $[\![\underline{x}^{\tau^!}]\!]_{\mathcal{I}\cdot\xi} \stackrel{\text{def}}{=} \underline{p}$ when $I \cdot \xi = \xi' \cdot x : \underline{p}$.

In the first clause, an opaque name is interpreted as a distinction, just as a reference name (cf. discussions in §11.4). The second clause is precisely as in the logic for open state, which deserves listing since it assumes, other than other part of the model, that the interpreted name is at least open. Other rules are interpreted precisely as before.

Satisfaction Relation and Semantics of Assertions. The satisfaction relation is also given by augmenting the one for the open logic. Following the previous section we shall treat reference and opaque names under identity distinction, which is not restrictive as far as we do not include the literal equality comparison on labels in the logic. Again for simplicity no equations/quantifications on opaque names are considered. We merely add two classes. First, the validity of the opening formula is given by:

$$\xi \models^{\mathcal{I}} \mathsf{open}\ \vec{x}\ \mathsf{as}\ (\boldsymbol{\nu}\vec{y})\vec{w}\ \mathsf{in}\ A \quad \equiv \quad \xi = (\boldsymbol{\nu}\,\vec{y})(\vec{w}:\underline{\vec{p}}\cdot\xi') \text{ s.t. } \xi\cdot\underline{\vec{w}}:\underline{\vec{p}}\cdot\xi' \models^{\mathcal{I}} A[\vec{w}/\vec{x}].$$

Note the clause stipulates that the replicated abstract processes after opening should indeed be open, as indicated by the underline notation. Intuitively it says that an opening can be done only in one go — one cannot open an opaque behaviour by pealing off its names one by one in distinct opening formulae: all hidden states of an opaque behaviour should be mentioned in one opening formula, after which it becomes open.

The semantics of stateful invocation with new reference generation is given as follows (assume a has type $(\vec{\rho}\sigma)^{!}_{\Upsilon}$ with $\mathsf{dom}(\Upsilon) = \{\vec{w}\}$).

$$\begin{array}{c} \xi \models^{\mathcal{I}} \{C\}[a \bullet \vec{b}/(\nu\vec{y})z]A\{C'\} \\ \equiv \\ \vec{w}:\underline{\vec{r}},\ \eta \models^{\mathcal{I}} C \ \supset\ \exists \underline{q}\vec{r'}.\ (\ \langle [\![a]\!]_{\mathcal{I}\cdot\xi}, [\![\vec{b}]\!]_{\mathcal{I}\cdot\xi}, \vec{r}^{\vec{w}}\rangle \Downarrow \langle \underline{s}, \vec{r'}, \vec{t}\rangle\ \wedge \\ \qquad\qquad z:\underline{s},\ \vec{y}:\vec{t},\ \vec{w}:\vec{r'},\ \xi,\ \eta \models^{\mathcal{I}} (A[z'/z] \wedge C)\ \ ). \end{array}$$

Finally we record the semantics of assertions.

- Let $P^{\overline{\Gamma};\Delta}$ be a non-thread. Then we set:

$$P^{\overline{\Gamma};\Delta} \models^{I} \mathbf{rely}\, A\, \mathbf{guar}\, B \quad \equiv \quad R^{\Gamma} \models^{\mathcal{I}} A \supset (\boldsymbol{\nu}\,\mathsf{fn}(\Gamma))(P|R) \models^{\mathcal{I}} B.$$

- Let $P^{\overline{\Gamma};\Delta}$ be a thread with $\overline{\Gamma} = \Upsilon \cdot \mathbf{?}\!\downarrow\!\Theta$, $\mathsf{dom}(\Theta) = \{\vec{w}\}$ and $\mathsf{dom}(\Delta) = \{\vec{z}\}$. Then we set:

$$\begin{array}{c} P^{\overline{\Gamma};\Delta} \models^{\mathcal{I}} \{C\}\mathbf{rely} A\, \mathbf{guar}^{\nu\vec{y}} B\, \{C'\} \\ \equiv \\ R^{\Gamma} \models^{\mathcal{I}} C \wedge A\ \supset\ (\nu\vec{w})(P|R) \models^{\mathcal{I}} \mathsf{open}\ \vec{z}\ \mathsf{as}\ (\boldsymbol{\nu}\vec{y})\underline{\vec{z}}\ \mathsf{in}\ (B \wedge C')[\underline{\vec{z}}/\vec{z}]. \end{array}$$

We also use the same sequent for threads as before to keep the compatibility from the open stateful logic,

$$P^{\Gamma} \models^{\mathcal{I}} \{C\}\mathbf{rely} A\, \mathbf{guar} B\, \{C'\}$$

which simply stands for

$$P^{\Gamma} \models^{\mathcal{I}} \{C\}\mathbf{rely} A\, \mathbf{guar}^{\nu\varepsilon} B\, \{C'\}$$

where ε is the empty string.

Adding Quantifiers on Opaque Names. If we are to add quantifiers over opaque/reference names, we would interpret them by setting the domain over which a quantified opaque/reference name range over, to be those primary names of the same type used in the judgement, together with an anonymous fresh name (which, because it is fresh, can be any such). So $\forall x^{[\tau]}.A$ means either A holds with x fresh, or $A[y_i/x]$ holds for each y_i of the same type in the action type considered (note the domain of the action type is extended by entering the scope of the hiding ν). However practically speaking the need to quantify opaque names (or references) may not often arise, even though exploration of further examples may eventually necessitate their usage. Nevertheless we can safely say that the lack of quantification over opaque names does not lead to the lack of generality in assertions since, after all, the number of primary names used in an assertion are always finite.

Notational Convention. Before moving into the discussion of the proof rule, we formally introduce the abbreviation for opening formula (this abbreviation was briefly mentioned in Section 11.2). The introduction of abbreviation is partly because an opening formula, for all its generality, is often too verbose for writing down a specification: the abbreviated notation also gives a better way to structure assertions, which is particularly useful for clear presentation of proof rules. The abbreviated notation is formally given as follows.

$$\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}^{\vec{\tau}}\ \mathsf{in}\ A \tag{11.38}$$

which stands for:

$$\mathsf{open}\ \vec{x}^{[\vec{\tau}/\vec{y}]}\ \mathsf{as}\ (\boldsymbol{\nu}\vec{y})\underline{\vec{x}}\ \mathsf{in}\ A[\underline{\vec{x}}/\vec{x}]. \tag{11.39}$$

In (11.38), we say $\vec{x}$ are *opened* in the formula. For ensuring the well-typedness of (11.39), we require the following syntactic conditions for A in (11.38).

1. Each name in $\vec{x}$ is used with type $\tau_i/\vec{y}$; and
2. No names in $\vec{x}$ are opened in A again.

Otherwise we assume the same well-typedness constraint as before.

We note the notation (11.38) cannot be further abbreviated to $(\boldsymbol{\nu}\,\vec{y})A$, unless we annotate A as A^{Γ} where Γ includes a typing for $\vec{x}$. Note that, if we do use the simplified notation $(\boldsymbol{\nu}\,x)A$ without this proviso, we lose the sharing information (recall when two opaque behaviours share the same local state, they should be opened together). This has two consequences. First, we cannot use the same semantics of the logic as we have been using so far: in fact, consistent semantics of $(\boldsymbol{\nu}\,x)A$ is hard to come by. Second, in accordance with this semantic issue, the proof rules presented in the next subsection are no longer consistent. Among others it is hard to make the weakening of effect names consistent without explicit information on sharing.

Because we can always restore the original formulae, using the abbreviation (11.38) does not mean we lose precision: neither, in fact, do we lose generality since we can always unfold inner opening formulae for the same name(s) into the outermost opening formula ("telescoping"), as we shall discuss later.

(Bra$^{\downarrow}$)
$$\frac{\forall i.\, P_i \vdash \{C\}\,\mathbf{rely}\; A[\mathtt{in}_i(\vec{y}_i)^{\uparrow}/x]\,\mathbf{guar}^{\nu\vec{w}}\,B\,\{C'\}}{x[\&_i(\vec{y}_i).P_i] \vdash \{C\}\,\mathbf{rely}\; A\,\mathbf{guar}^{\nu\vec{w}}\,B\,\{C'\}}$$

(FSel$^{\uparrow}$) $(A \supset B[\mathtt{in}_i(\vec{y})^{\uparrow}/x])$
$$\frac{-}{\overline{x}\mathtt{in}_i\langle\vec{y}\rangle \vdash \{C\}\,\mathbf{rely}\; A\;\mathbf{guar}^{\nu\varepsilon}\;B\;\{C\}}$$

(In$^{!}$) $(\forall\vec{y}.(B_1 \supset \{C\}[x\bullet\vec{y}/(\nu\vec{w})z]B_2\{C'\}) \supset B)$
$$\frac{P \vdash \{C\}\,\mathbf{rely}\;\; A^{\text{-}\vec{y}} \wedge B_1^{\vec{y}}\;\;\mathbf{guar}^{\nu\vec{w}}\;\; B_2\;\;\{C'\}}{!x(\vec{y}z).P \vdash \mathbf{rely}\;\; A\;\mathbf{guar}\;\; B}$$

(FOut$^{?}$) $(A \supset \{C\}[x\bullet\vec{y}/(\nu\vec{w})z]B\{C'\})$
$$\frac{-}{\overline{x}\langle\vec{y}z\rangle \vdash \{C\}\,\mathbf{rely}\; A\,\mathbf{guar}^{\nu\vec{w}}\,B\,\{C'\}}$$

(Ref) $(A[!x/y] \supset B\,)$
$$\frac{-}{\mathsf{Ref}\langle xy\rangle \vdash \mathbf{rely}\; A\,\mathbf{guar}\; B}$$

(Read) $(C \supset B[(!x)^{\uparrow}/c\,])$
$$\frac{-}{\overline{x}\mathsf{read}\langle c\rangle \vdash \{C\}\,\mathbf{rely}\; A\,\mathbf{guar}^{\nu\varepsilon}\,B\,\{C\}}$$

(Write)
$$\frac{A[!x/y] \wedge \exists x.C \supset C'}{\overline{x}\mathsf{write}\langle yc\rangle \vdash \{C\}\,\mathbf{rely}\; A \wedge B[()^{\uparrow}/c]\,\mathbf{guar}^{\nu\varepsilon}\,B\,\{C'\}}$$

Fig. 40. Refined Proof Rules for Local State (prefix)

11.6 Proof Rules

The proof rules for sequential processes with local state use essentially identical rules of the logic for open state in Figure 31 and Figure 30. In addition, they contain the rules for opening formulae and hiding. The proof system as a whole is organised following the reasoning principle which we may summarise as:

> *To reason about behaviours with local state, including those in the environment and in the process, we first reason for its opened behaviour, then reason for the result of hiding some of its states.*

Behaviourally this principle works because all stateful behaviours represented by typed processes are based on the combination of replicated behaviours and references, part of the latter can be hidden.

Among the prefix rules from the open stateful logic listed in Figure 40, (Bra$^{\downarrow}$) and (FSel$^{\uparrow}$) are as before. Note that, by the well-typedness of terms, the linear output name x and the names it carries should all occur as open names in the respective formula in these rules. (In$^{!}$) and (Out$^{?}$) relate, in mutually dual directions, a local stateful invocation and a local thread judgement. Again x, $\vec{y}$ and z should occur as open names in the respective formula. (Ref), (Read)

(Par-$\downarrow\!\uparrow$)

$$\frac{P_1^{\Gamma;x:\tau^{\uparrow}} \vdash \{C\}\,\mathbf{rely}\ A_1\,\mathbf{guar}^{\nu\vec{w}_1}\ E\,\{C''\} \qquad P_2^{\Gamma\cdot x:\overline{\tau}\cdot\vec{w}_1:\vec{\delta}\,;\,\uparrow\Delta} \vdash \{C''\}\,\mathbf{rely}\ A_2\wedge E\,\mathbf{guar}^{\nu\vec{w}_2}\ B\,\{C'\}}{P_1|P_2{}^{\Gamma;\Delta} \vdash \{C\}\,\mathbf{rely}\ A_1\wedge A_2\ \mathbf{guar}^{\nu\vec{w}_1\vec{w}_2}\ B\,\{C'\}}$$

(Weak-$\mathbf{?}_{rw}$)

$$\frac{P^{\Gamma} \vdash \{C^{\text{-}x}\}\,\mathbf{rely}\ A\,\mathbf{guar}^{\nu\vec{w}}\ B\,\{C'^{\text{-}x}\}}{P^{\Gamma,x:\overline{ref}(\tau)} \vdash \{C\wedge E^{x}\}\,\mathbf{rely}\ A\,\mathbf{guar}^{\nu\vec{w}}\ B\,\{C'\wedge E\}}$$

Fig. 41. Refined Proof Rules for Local State (composition)

and (Write) are precisely the same as their original rules for open state. Again concerned primary names should all be open.

Remark 13. (Rules for Bound Output) For reference, we present the proof rules for bound output in Figure 45. The rules are given assuming names which are sent in these bound outputs are opaque (which are the general cases). These rules are easily derivable by combining the corresponding rules for free outputs by (Par) and (Hide) rules. Each rule assumes the conditions are given as opening formulae, so that the expressions in the side condition, which use open names, are well-defined. In (Sel), opaque names carried by the output are assumed to be opened together (when there are other opaque names which share the same local state, the free output counterpart is more flexible). These rules are useful when developing proof rules for programming languages with local state.

The rules for composition are again essentially identical, of which two updated rules are given in Figure 41. In (Par-$\downarrow\!\uparrow$), $\vec{w}_1$, the hidden names of the sequent from the preceding process, should occur as references in the environment in the following process, and that they are again bound in the conclusion. In (Weak-$\mathbf{?}_{rw}$), we assume y should also be fresh from the hidden names $\vec{w}$. The rule is in fact far more non-trivial than the corresponding rule in the open state, due to possible sharing: we can however justify the rule using the interpretation of formulae in §11.5. The original (Hide-Global) for hiding reference names is now redundant, since the new hiding rule subsumes it: however the original rules are still sound and are useful as specialised rules.

On the basis of the proof rules from the open stateful logic, we use the rules proper to the local stateful logic, presented in Figures 42a and 43. The rules in Figure 42 introduce opening formulae and hiding; while the rules in Figure 43 shift formulae between rely/pre/post conditions. As always, we assume the well-typedness for processes and sequents in each of these rules, including in

(Open)

$$\frac{P^{\Gamma} \vdash \textbf{rely } A \textbf{ guar } B}{P^{\Gamma} \vdash \textbf{rely } A \textbf{ guar } \mathsf{open}\,(\boldsymbol{\nu}\varepsilon)\vec{x}\,\mathsf{in}\,B}$$

(Co-Open)

$$\frac{P^{\Gamma} \vdash \textbf{rely } A \textbf{ guar } B}{P^{\Gamma} \vdash \textbf{rely } \mathsf{open}\,(\boldsymbol{\nu}\varepsilon)\vec{x}\,\mathsf{in}\,A \textbf{ guar } B}$$

(Hide) $(\boldsymbol{\sigma} = [\boldsymbol{\nu}\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\vec{y}_i x\vec{z}_i]_i)$

$$\frac{P^{\Gamma\cdot x:ref(\tau)} \vdash \textbf{rely } A \textbf{ guar } B\boldsymbol{\sigma}}{(\boldsymbol{\nu}\,x)P^{\Gamma/x} \vdash \textbf{rely } A \textbf{ guar } B}$$

(Co-Hide) $(\boldsymbol{\sigma} = [\boldsymbol{\nu}\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\vec{y}_i x\vec{z}_i]_i)$

$$\frac{P^{\Gamma} \vdash \textbf{rely } A\boldsymbol{\sigma} \textbf{ guar } B}{P^{\Gamma/x} \vdash \textbf{rely } A \textbf{ guar } B}$$

(Hide-thread)

$$\frac{P^{\Gamma\cdot x:ref(\tau)} \vdash \{C\}\, \textbf{rely } A\, \textbf{guar}^{\nu\vec{w}}\, B\, \{C'\}}{(\boldsymbol{\nu}\,x)P^{\Gamma/x} \vdash \{C\}\, \textbf{rely } A\, \textbf{guar}^{\nu\vec{w}x}\, B\, \{C'\}}$$

(Co-Hide-thread-1) $(\boldsymbol{\sigma} = [\boldsymbol{\nu}\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\vec{y}_i x\vec{z}_i]_i)$

$$\frac{P^{\Gamma} \vdash \{C\boldsymbol{\sigma}\}\, \textbf{rely } A \textbf{ guar } B\, \{C'\boldsymbol{\sigma}\}}{P^{\Gamma/x} \vdash \{C\}\, \textbf{rely } A \textbf{ guar } B\, \{C'\}}$$

(Co-Hide-thread-2) $(\boldsymbol{\sigma} = [\boldsymbol{\nu}\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\vec{y}_i x\vec{z}_i]_i)$

$$\frac{P^{\Gamma} \vdash \{C\}\, \textbf{rely } A\boldsymbol{\sigma} \textbf{ guar } B\, \{C'\}}{P^{\Gamma/x} \vdash \{C\}\, \textbf{rely } A \textbf{ guar } B\, \{C'\}}$$

Fig. 42. Open/Hide Rules for Local State

conclusion. Below we illustrate these rules, starting from the rules on the left column in Figure 42, focussing on how they reflect the above reasoning principle.

- In (Open), assume we have reasoned for open $\vec{x}$ of A. Then we can make them opaque, by introducing a trivial opening. Similarly for (Open-Thread).
- After making open names opaque, we can hide references they use one by one, using (Hide) and (Hide-Thread). These rules use the notation:

  $$A[\boldsymbol{\nu}\,\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\,\vec{y}_i x\vec{z}_i]_i$$

  which indicates multiple substitutions, replacing each occurrence of a binder of shape $\boldsymbol{\nu}\,\vec{y}_i x\vec{z}_i$ with $\boldsymbol{\nu}\,\vec{y}_i\vec{z}_i$. As an example,

  $$(\mathsf{open}\ u\ \mathsf{as}\ (\boldsymbol{\nu}wx)\underline{u}\ \mathsf{in}\ A \wedge \mathsf{open}\ uv\ \mathsf{as}\ (\boldsymbol{\nu}wxw')\underline{uv}\ \mathsf{in}\ B)[\boldsymbol{\nu}\,\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\,\vec{y}_i x\vec{z}_i]_i$$

  yields the following formula, assuming no binding by x occurs in A.

  $$\mathsf{open}\ u\ \mathsf{as}\ (\boldsymbol{\nu}w)\underline{u}\ \mathsf{in}\ A \wedge \mathsf{open}\ uv\ \mathsf{as}\ (\boldsymbol{\nu}ww')\underline{uv}\ \mathsf{in}\ B.$$

The rules on the left column of Figure 42 introduce the opening formulae in the positive (guarantee; process) part of an assertion, first the empty ones then adding local references one by one. The rules on the right column of the same figure do the same for the negative (pre/post/rely; environment) part of an assertion, as illustrated below.

(Shift-Out) ($\vec{x}$ trivially open in E)

$$\frac{P \vdash \{C\}\,\mathbf{rely}\ A \wedge E^{\vec{x}}\,\mathbf{guar}^{\nu\vec{w}}\ B\,\{C'\}}{P \vdash \{C \wedge E\}\,\mathbf{rely}\ A\ \mathbf{guar}^{\nu\vec{w}}\ B\,\{C' \wedge E\}}$$

(Shift-In)

$$\frac{P \vdash \{C \wedge E^{\vec{x}}\}\,\mathbf{rely}\ A\ \mathbf{guar}^{\nu\vec{w}}\ B\,\{C'^{\text{-}\vec{x}}\}}{P \vdash \{C\}\,\mathbf{rely}\ A \wedge E\ \mathbf{guar}^{\nu\vec{w}}\ B\,\{C'\}}$$

(Shift-Right) ($G \wedge C'_0 \supset C'[\boldsymbol{\nu}\,\vec{y}_i\vec{z}_i/\nu\vec{y}_i\vec{w}\vec{z}_i]_i$)

$$\frac{P \vdash \{C\}\,\mathbf{rely}\ A\,\mathbf{guar}^{\nu\vec{u}\vec{w}}\ (B^{\text{-}\vec{w}} \wedge G^{\vec{w}})\,\{C'_0\}}{P \vdash \{C\}\,\mathbf{rely}\ A\ \mathbf{guar}^{\nu\vec{u}}\ B\,\{C'\}}$$

(Shift-Left) ($\boldsymbol{\sigma} \stackrel{\text{def}}{=} [\boldsymbol{\nu}\,\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\,\vec{y}_i\vec{w}\vec{z}_i]_i$)

$$\frac{P \vdash \{C\boldsymbol{\sigma}\}\,\mathbf{rely}\ A\,\mathbf{guar}^{\nu\vec{u}}\ B\,\{C'^{\text{-}\vec{w}} \wedge G^{\vec{w}}\}}{P \vdash \{C\}\,\mathbf{rely}\ A\ \mathbf{guar}^{\nu\vec{u}\vec{w}}\ B \wedge G\,\{C'\}}$$

Fig. 43. Shift Rules

- In (Co-Open), we introduce the empty opening formula in the rely condition. To see its need, suppose we have so far reasoned about a process assuming (part of) the environment is open, i.e. it does not hide its local state (which is essential when using, for example, (Out)). After completing the reasoning, we may need to close it (for example for having it composed with a process), in which case we use this rule, followed by co-hiding illustrated below.
- (Co-Hide) and (Co-Hide-thread-1/2) add local references (hiding) to opening formulae in negative positions, introduced by, among others, (Co-Open)/(Co-Oepn-thread). In (Co-Hide-thread-1), we hide x in both the pre/post-conditions. These rules are essential when we wish to compose a process with another process which has a local state hence whose internal structure is hidden (while having the stipulated behaviour).

The shift rules in Figure 43 move opening formulae among different positions in an assertion.

- Since the reasoning for the environment is done by placing: (1) open replicated names in the rely condition; and (2) associated references in the pre/post conditions, we need a mechanism to move the former to the pre/post conditions to form an opening formula. This is done by (Shift-Out).
- (Shift-In) is its symmetric version, retrieving the opaque replicated behaviour from the pre-condition to the rely condition (the semantics says the rely condition is the initial condition, so it cannot come from the post-condition). For ensuring type conformance, we should have no corresponding primary names in the post-condition.
- (Shift-Right) moves part of the hidden state from the guarantee formula to the post-condition. (Shift-Left) is its precise dual (the substitution in the pre-condition is necessary to turn the initial open state in the pre-condition to its hidden counterpart).

(open-$\boldsymbol{\nu}$-elim)	$\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,A^{\text{-}\vec{x}\vec{y}}$	$\equiv$	A
(open-$\boldsymbol{\nu}$-perm-1)	$\mathsf{open}\,(\boldsymbol{\nu}\vec{z}_1 y_1 y_2 \vec{z}_2)\vec{x}\,\mathsf{in}\,A$	$\equiv$	$\mathsf{open}\,(\boldsymbol{\nu}\vec{z}_1 y_2 y_1 \vec{z}_2)\vec{x}\,\mathsf{in}\,A$
(open-$\boldsymbol{\nu}$-perm-2)	$\mathsf{open}\,(\boldsymbol{\nu}\vec{z})\vec{x}_1 y_1 y_2 \vec{x}_2\,\mathsf{in}\,A$	$\equiv$	$\mathsf{open}\,(\boldsymbol{\nu}\vec{z})\vec{x}_1 y_2 y_1 \vec{x}_2\,\mathsf{in}\,A$
(open-$\boldsymbol{\nu}$-dist-1)	$\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,(A \star B)$	$\supset$	$(\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,A) \star (\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,B)$
(open-$\boldsymbol{\nu}$-dist-2)	$\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,(\mathcal{Q}z.A)$	$\equiv$	$\mathcal{Q}z.\,(\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,A)$
(open-$\boldsymbol{\nu}$-scope)	$(\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,A) \star B^{\text{-}\vec{x}\vec{y}}$	$\equiv$	$\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,(A \star B)$
(open-$\boldsymbol{\nu}$-vacuous)	$\mathsf{open}\,(\boldsymbol{\nu}\vec{y}_1\vec{y}_2)\vec{x}\,\mathsf{in}\,A^{\text{-}z}$	$\supset$	$\mathsf{open}\,(\boldsymbol{\nu}\vec{y}_1 z \vec{y}_2)\vec{x}\,\mathsf{in}\,A$
(open-$\boldsymbol{\nu}$-concat)	$\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,(\mathsf{open}\,(\boldsymbol{\nu}\vec{v})\vec{u}\,\mathsf{in}\,A)$	$\supset$	$\mathsf{open}\,(\boldsymbol{\nu}\vec{y}\vec{v})\vec{x}\vec{u}\,\mathsf{in}\,A$

Above, $\star$ ranges over $\{\wedge, \vee, \supset\}$, while $\mathcal{Q}$ ranges over $\{\forall, \exists\}$.

Fig. 44. Syntactic Axioms for Local State

As we have seen so far, the proof system for local state is syntactically incremental over the proof system for open state. It is however worth noting that the nature of validity with the local state substantially differs from the one with open state, which affects (Consequence) rule and other rules. There are two sources which may cause additional complications.

1. The need for soundly introducing/moving opening formulae (e.g. the opening rules demand the opening occurs in the outermost position).
2. The local state makes it highly non-trivial to reason about satisfaction of formulae by processes, affecting the calculation of entailment.

Regarding the point 1, there are a few basic rules for mechanically manipulating opening formulae, which we list in Figure 44. Note, by the definition of validity of opening formulae, the standard α-conversion is immediately sound.

- (open-$\boldsymbol{\nu}$-elim) eliminates the opening formulae. We implicitly convert the types of $\vec{x}$ from opaque ones to open ones, which is legal if the formula arises as, e.g., as the whole of the rely condition. (open-$\boldsymbol{\nu}$-perm-1/2) simply permute names in the guard part.
- (open-$\boldsymbol{\nu}$-dist-1/2) distributes an opening. In (open-$\boldsymbol{\nu}$-dist-2), if we allow opaque/reference names to be quantified (cf. §11.5), z should be distinct from $\vec{x}\vec{y}$.
- (open-$\boldsymbol{\nu}$-scope) is a special case of the following rule.

Proposition 7. (open-$\boldsymbol{\nu}$-telescope) *For some permutation σ over $\vec{y}$,*

$$(\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,A) \star (\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,B) \quad \supset \quad \mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,(A \star B\sigma).$$

Note the rules is essentially the converse of (open-$\boldsymbol{\nu}$-dist-1). We need a permutation since two bound reference names can in general denote different entities by the definition of validity of an opening formula.

$$\frac{(\mathsf{Sel}^{\uparrow}) \quad P^{\Gamma;\vec{y}:\vec{\tau}} \vdash \mathbf{rely}\ A\ \mathbf{guar}\ \mathsf{open}\,(\boldsymbol{\nu}\vec{w})\vec{y}\,\mathsf{in}\,B[\mathtt{in}_i(\vec{y})^{\uparrow}/x]}{\overline{x}\mathtt{in}_i(\vec{y})P \vdash \{C\}\,\mathbf{rely}\ A\ \mathbf{guar}^{\nu\vec{w}}\ B\,\{C\}}$$

$$\frac{(\mathsf{Out}^{?}) \quad (A \supset \{C^{\text{-}\vec{w}}\}[x\bullet\vec{y}/(\nu\vec{w})z]A_2\{D\}) \quad P \vdash \{D\}\,\mathbf{rely}\ A \wedge A_2^z\ \mathbf{guar}^{\nu\vec{u}}\ A_1^{\vec{y}} \wedge B^{\text{-}\vec{y}}\,\{C'\}}{\overline{x}(\vec{y}z)P \vdash \{C\}\,\mathbf{rely}\ A\,\mathbf{guar}^{\nu\vec{u}\vec{w}}\ B\,\{C'\}}$$

Fig. 45. Bound Output Rules for Local State

– (open-$\boldsymbol{\nu}$-vacuous) adds a new vacuous local reference to an opening formula. The converse is not valid (even ill-typed). (open-$\boldsymbol{\nu}$-concat) concatenate two consecutive guards.

By applying these rules, we can always accumulate all the openings in a formula into a single outermost opening, which is essential for the generality of (Open) and (Co-Open).

Next we illustrate the non-triviality of satisfaction using a simple example. The following two formulae, similar to the ones given in §11.2, in fact define the same set of processes, assuming the typing $x:((\mathbb{N})^{\uparrow})^{!}$ and j range over $\{1,2\}$.

$$A \stackrel{\text{def}}{=} \mathsf{open}\,(\boldsymbol{\nu}y_1y_2)x\,\mathsf{in}\,(\{\wedge_j y_j = i_j\}x \bullet \varepsilon = (i_1+i_2)^{\uparrow}\{\wedge_j !y_j = !i_j\} \wedge\ !y_1 = !y_2 = 0)$$
$$B \stackrel{\text{def}}{=} \mathsf{open}\,(\boldsymbol{\nu}y)x\,\mathsf{in}\,(\{!y = i\}x \bullet \varepsilon = (2\times i)^{\uparrow}\{!y = i+1\} \wedge\ !y = 0)$$

To show these two formulae are logically equivalent, we simply observe that a process which satisfies either of these formulae should first outputs 0, then 2, and so on, which shows both formulae define an identical congruent class. Thus validity of these formulae coincide. Note this shows a process such as

$$(\boldsymbol{\nu}\,y_1y_2)(\ !x(c).\overline{y_1}\mathsf{read}(c_1)c_1(n_1).\overline{y_2}\mathsf{read}(c_2)c_2(n_2).\overline{c}\langle n_1+n_2\rangle \mid \mathsf{Ref}\langle y_1 0\rangle \mid \mathsf{Ref}\langle y_2 0\rangle\ )$$

has not only A but also B as its derivable specification, which becomes essential if we wish to hypothesise the environment such as B and cancel the hypothesis by parallel composition with the above agent. Apart from tractable cases for e.g. the first-order state, the general inference rules for inhabitation/entailment are unknown at present and may not be easy to derive. The transition-based reasoning is the most tractable method, either for processes or programs (the latter via encoding), which is sound because of a characterisation of the contextual congruence by transition. We hope to discuss how the transition-based reasoning would help in such cases on later occasions.

The use of structural rules other than the consequence rule plays an important role in reasoning about local state. This will be discussed in the next subsection.

11.7 Weakening and Separation

The local state logic inherits the same structural rules from the open stateful logic (cf. Remark 9, Page 136). Because the syntax of judgement has changed slightly, we reproduce these rules in th following.

$$\text{(Conj)}\ \frac{P^{\Gamma} \vdash \{C\}\,\textbf{rely}\,A\,\textbf{guar}^{\nu\vec{y}}\,B_i\,\{C_i\}\quad (i=1,2)}{P^{\Gamma} \vdash \{C\}\,\textbf{rely}\,A\,\textbf{guar}^{\nu\vec{y}}\,B_1 \wedge B_2\,\{C_1 \wedge C_2\}}$$

$$\text{(Disj)}\ \frac{P^{\Gamma} \vdash \{C_i\}\,\textbf{rely}\,A_i\,\textbf{guar}^{\nu\vec{y}}\,B\,\{C\}\quad (i=1,2)}{P^{\Gamma} \vdash \{C_1 \vee C_2\}\,\textbf{rely}\,A_1 \vee A_2\,\textbf{guar}^{\nu\vec{y}}\,B\,\{C\}}$$

$$\text{(Aux-}\wedge,\vee)\ \frac{P^{\Gamma} \vdash \{C\}\,\textbf{rely}\,A\,\textbf{guar}^{\nu\vec{y}}\,B\,\{C'\}\quad \mathsf{fn}(E) \cap (\mathsf{dom}(\Gamma) \cup \{\vec{y}\}) = \emptyset}{P^{\Gamma} \vdash \{C \star E\}\,\textbf{rely}\,A\,\textbf{guar}^{\nu\vec{y}}\,B\,\{C' \star E\}}$$

where, in the last rule, we set $\star \in \{\wedge, \vee\}$.

The combination of (Aux-$\wedge$) and (Weak-$?_{rw}$) leads to the following derived rule of a familiar shape.

$$(\wedge)\ \frac{P^{\Gamma} \vdash \{C\}\,\textbf{rely}\,A\,\textbf{guar}^{\nu\vec{y}}\,B\,\{C'\}}{P^{\Gamma\cdot\Upsilon} \vdash \{C \wedge E^{\Upsilon}\}\,\textbf{rely}\,A\,\textbf{guar}^{\nu\vec{y}}\,B\,\{C' \wedge E^{\Upsilon}\}}$$

Above we remember $\Gamma \cdot \Upsilon$ indicates Γ and Υ are disjoint in their domains. Note that, because $\mathsf{dom}(\Upsilon)$ are weakened, they cannot be written during the execution of P (i.e. the change from C to C' has nothing to do with names in $\mathsf{dom}(\Upsilon)$). There are several points which are worth noting.

1. The rule does not need the standard side condition which says E does not mention a variable written by P, since a weakened name is never written by a program.
2. Nor does its syntactic disjointness on names let us lose generality. Further, practically, we may seldom need to strengthen pre/post-conditions on names which are only read after reasoning about them.
3. The present syntactic disjointness condition is much weaker, or more general, than the separated conjunction studied by O'Hearn, Reynolds and others [16, 95, 104, 105], because the present condition allows primary names asserted in C and those in E can have a(n unspecified) shared common data structure via references. This has an immediate practical benefit in reasoning about composite programs which work with such data structures (which are pervasive in object-oriented languages, see Section 12). The rule is however immediately sound since, intuitively, even if part of the shared structure, unfolded from the visible primary names, is ever touched, this cannot be reflected on the change from $C \wedge E$ to $C' \wedge E$, since, if these formulae ever mentioned this common part, they should have used opening of a primary name from C (or C') and a primary name from E, which is impossible by the shape of formulae.

The third point above is important since it elucidates the reason why the simple syntactic disjointness condition is enough to allow consistent reasoning for shared mutable data structure, which properly generalise the separated conjunction in [16, 95, 104, 105] (since if the "memory region" which C or C' is about, and the one which E is about are disjoint, the side condition is trivially applicable). The separated conjunction is however useful in reasoning because the connective automatically guarantees name disjointness and, hence, the soundness of weakening/conjunction, allowing localised reasoning [95] when, as is often the case in practice, different parts of programs touch completely disjoint parts of memory regions. On the other hand, the full generality of the present framework would become useful when we wish to keep the locality of reasoning in the presence of mutation of complex sharing structures whose structures are unfolded and updated during computation. It should also be said that the work by O'Hearn, Reynolds and others [16, 95, 104, 105] combine separated conjunction (and its variants) with manipulation of low-level operationrs, offering a framework for structured reasoning concerning some of the low-level features in programming languages. As we discussed in Introduction, it remains to be seen how the present logical framework may be extended to treat low-level programming constructs as treated in [16, 95, 104, 105].

11.8 Reasoning Examples

In the following we present simple reasoning examples with local state, taking examples from §11.2. We start from:

$$P \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, y)(R \,|\, \mathsf{Ref}\langle y0\rangle)$$

whose R is given by:

$$R \stackrel{\text{def}}{=} !x(z).\overline{y}\mathsf{read}(c)c(n).\overline{y}\mathsf{write}(c')c'(n+1).\overline{z}(n+1).$$

The following reasoning starts from the point where R has been reasoned, which can be done purely using the proof system for open state. Below we let $B \stackrel{\text{def}}{=} \forall i.\, \{!y = i\}\, x \bullet \varepsilon \rightsquigarrow i+1\, \{!y = i+1\}$.

1.	$R \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ B$	(...)
2.	$\mathsf{Ref}\langle y0\rangle \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ !y = 0$	(Ref)
3.	$R \,|\, !y = 0 \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ B \wedge !y = 0$	(1,2,Par)
4.	$R \,|\, !y = 0 \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{open}\ (\boldsymbol{\nu}\varepsilon)x\ \mathsf{in}\ B \wedge !y = 0$	(Open)
5.	$(\boldsymbol{\nu}\, y)(R \,|\, !y = 0) \vdash \mathbf{rely}\ \mathsf{T}\ \mathbf{guar}\ \mathsf{open}\ (\boldsymbol{\nu}y)x\ \mathsf{in}\ B \wedge !y = 0$	(Hide)

as required. Note how the combination of (Open) and (Hide) allows a structured reasoning for local state.

Next we consider a process which acts as a client to P (also from §11.2).

$$Q \stackrel{\text{def}}{=} \overline{x}(z)z(n).\overline{x}(z')z'(n').\overline{w}\langle n+n'\rangle.$$

Q interacts with P twice, and sends the sum of the received values to w. For Q we show its composition with P outputs 3, that is:

$$Q|P \vdash \{\mathsf{T}\}\,\mathbf{rely}\,\mathsf{T}\,\mathbf{guar}\,w = (3)^{\uparrow}\,\{\mathsf{T}\}$$

Because the derivation for Q assumes the property of P as its environment, the inference now uses the dual of (Open) and (Hide), the co-opening and co-hiding. Below we use the proof rules for agents with value passing from Figure 32, and, for brevity, let $Q_0 \stackrel{\text{def}}{=} \overline{x}(z')z'(n').\overline{w}\langle n+n'\rangle$. Also for brevity we write $(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,A$ for $\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,A$, omitting the keyword "open". At the last step, we use the conclusion from the previous derivation, which we name as $(*)$.

1.	$\overline{w}\langle n+n'\rangle \vdash \{\mathsf{T}\}\,\mathbf{rely}\,n=1 \wedge n'=2 \wedge B\,\mathbf{guar}\,w=(3)^{\uparrow}\,\{\mathsf{T}\}$	(LOut-n)
2.	$z'(n').\overline{w}\langle n+n'\rangle \vdash \{n=1 \wedge z'=(3)^{\uparrow} \wedge B\}\,\mathbf{rely}\,\mathsf{T}\,\mathbf{guar}\,w=(3)^{\uparrow}\,\{\mathsf{T}\}$	(LIn-n)
3.	$\overline{x}(z')z'(n').\overline{w}\langle n+n'\rangle \vdash \{!y=1\}\,\mathbf{rely}\,n=1 \wedge B\,\mathbf{guar}\,\mathsf{T}\,\{w = (3)^{\uparrow}\}$	(Out)
4.	$z(n).Q_0 \vdash \{!y=1\}\,\mathbf{rely}\,z=(1)^{\uparrow} \wedge B\,\mathbf{guar}\,w = (3)^{\uparrow}\,\{\mathsf{T}\}$	(LIn-n)
5.	$\overline{x}(z)z(n).Q_0 \vdash \{!y=0\}\,\mathbf{rely}\,B\,\mathbf{guar}\,w = (3)^{\uparrow}\,\{\mathsf{T}\}$	(Out)
6.	$Q \vdash \{!y=0 \wedge (\boldsymbol{\nu}\varepsilon)x\,\mathsf{in}\,B\}\,\mathbf{rely}\,\mathsf{T}\,\mathbf{guar}\,w = (3)^{\uparrow}\,\{(\boldsymbol{\nu}\varepsilon)x\,\mathsf{in}\,B\}$	(Shift-Out)
7.	$Q \vdash \{(\boldsymbol{\nu}\varepsilon)x\,\mathsf{in}\,(B\wedge !y=0)\}\,\mathbf{rely}\,\mathsf{T}\,\mathbf{guar}\,w = (3)^{\uparrow}\,\{\mathsf{T}\}$	($\boldsymbol{\nu}$-scope, Conseq)
8.	$Q \vdash \{(\boldsymbol{\nu}y)x\,\mathsf{in}\,(B\wedge !y=0)\}\,\mathbf{rely}\,\mathsf{T}\,\mathbf{guar}\,w = (3)^{\uparrow}\,\{\mathsf{T}\}$	(Co-Hide-thread)
9.	$Q|P \vdash \{\mathsf{T}\}\,\mathbf{rely}\,\mathsf{T}\,\mathbf{guar}\,w = (3)^{\uparrow}\,\{\mathsf{T}\}$	(9, $(*)$, Par)

as required. Note how (Shift-Out) is used in Line 6. Its application together with the use of (scope) rule in Line 7 is essential for applying (Co-Hide-thread) in Line 8.

Next we consider the case when two replications share a common local state, again taking an example from §11.2. First, the following process interacts with the environment with two replicated names.

$$Q' \stackrel{\text{def}}{=} \overline{x_1}(z)z(n).\overline{x_2}(z')z'(n').\overline{w}\langle n+n'\rangle.$$

Our purpose is to prove:

$$Q \vdash \{C(0)\}\,\mathbf{rely}\,\mathsf{T}\,\mathbf{guar}\,w = (3)^{\uparrow}\,\{C(2)\}$$

where we set, with $B' \stackrel{\text{def}}{=} B[x_1/x] \wedge B[x_2/x]$ (B is the predicate used above):

$$C(i) \stackrel{\text{def}}{=} \mathsf{open}\,(\boldsymbol{\nu}y)x_1x_2\,\mathsf{in}\,(B' \wedge !y = i)$$

Thus we infer the behaviour of Q' assuming x_1 and x_2 share the same local state. The inference follows, where we start from the opened formulae in the rely/pre-post conditions which we later close. Below we let $Q' \stackrel{\text{def}}{=} \overline{x_2}(z')z'(n').\overline{w}\langle n+n'\rangle$ and $C' \stackrel{\text{def}}{=} \mathsf{open}\,(\boldsymbol{\nu}\varepsilon)x_1x_2\,\mathsf{in}\,(B' \wedge !y = i)$ for brevity.

1.	$\overline{w}\langle n{+}n'\rangle \vdash \{\mathsf{T}\}\,\textbf{rely}\,n{=}1 \wedge n'{=}2 \wedge B'\,\textbf{guar}\,w{=}(3)^{\uparrow}\,\{\mathsf{T}\}$	(LOut-n)
2.	$z'(n').\overline{w}\langle n{+}n'\rangle \vdash \{\mathsf{T}\}\,\textbf{rely}\,n{=}1 \wedge z'{=}(2)^{\uparrow} \wedge B'\,\textbf{guar}\,w{=}(3)^{\uparrow}\,\{\mathsf{T}\}$	(LIn-n)
3.	$Q_0' \vdash \{!y{=}1\}\,\textbf{rely}\,n{=}1 \wedge B'\,\textbf{guar}\,w\,=\,(3)^{\uparrow}\,\{!y{=}2\}$	(Weak-$?_{rw}$, Out)
4.	$z(n).Q_0' \vdash \{!y{=}1\}\,\textbf{rely}\,z{=}(1)^{\uparrow} \wedge B'\,\textbf{guar}\,w{=}(3)^{\uparrow}\,\{!y{=}2\}$	(LIn-n)
5.	$\overline{x_1}(z)z(n).Q_0 \vdash \{!y{=}0\}\,\textbf{rely}\,B'\,\textbf{guar}\,w{=}(3)^{\uparrow}\,\{!y{=}2\}$	(Conseq, Out)
6.	$Q' \vdash \{C'(0)\}\,\textbf{rely}\,\mathsf{T}\,\textbf{guar}\,w{=}(3)^{\uparrow}\,\{C'(2)\}$	(Shift-Out, $\boldsymbol{\nu}$-scope, Conseq)
7.	$Q' \vdash \{C(0)\}\,\textbf{rely}\,\mathsf{T}\,\textbf{guar}\,w{=}(3)^{\uparrow}\,\{C(2)\}$	(Co-Hide-thread)

The derivation is essentially identical with the previous one except for the introduction of the empty opening formula for both x_1 and x_2. This step is a logical consequence of the typing of x_1 and x_2, left implicit above, which we list below.

$$x_1 : ((\mathbb{N})^{\downarrow})^{?}_{y:\overline{ref}(\mathbb{N})},\ x_2 : ((\mathbb{N})^{\downarrow})^{?}_{y:\overline{ref}(\mathbb{N})}$$

Thus if we ever wish to hide y, we need to hide both names. From a different viewpoint, consider the following process, again from §11.2, which satisfies $C(0)$:

$$S \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,y)(R[x_1/x] \mid R[x_2/x] \mid \mathsf{Ref}\langle y0\rangle)$$

The derivation is easy and omitted. Note S does *not* satisfy $C''(0)$, where $C''(i)$ is the following simplified version of $C(0)$.

$$C''(i) \stackrel{\text{def}}{=} \mathsf{open}\,(\boldsymbol{\nu}y')x_1\,\mathsf{in}\,(\{!y' = i\}x_1 \bullet \varepsilon \rightsquigarrow i\{!y' = i+1\} \wedge !y' = i)$$

where we did an alpha-conversion for clarity. The sequent $S \models \textbf{rely}\,\mathsf{T}\,\textbf{guar}\,C''(0)$ is then *not* well-typed — the name y' carried in the type of x_1 is bound, so the typings do not match.

We now turn to the reasoning for the generation of opaque behaviour, using the following process from (11.19).

$$O \stackrel{\text{def}}{=} !u(nz).\overline{z}(x)(\boldsymbol{\nu}\,y)(R \mid \mathsf{Ref}\langle yn\rangle)$$

For this agent we wish to derive:

$$O \vdash \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ \forall i \in \mathbb{N}.\,[u \bullet i/z]E(z,i)$$

where we set:

$$E(z,i) \stackrel{\text{def}}{=} \mathsf{open}\,(\boldsymbol{\nu}y)(\boldsymbol{\nu}x)\,\mathsf{in}\,(B \wedge !y{=}i)$$

If we use (Sel$^\uparrow$), we can start from the judgement for $(\boldsymbol{\nu}\, y)(R \mid \mathsf{Ref}\langle yn\rangle)$ (which is derived precisely as $(\boldsymbol{\nu}\, y)(R \mid \mathsf{Ref}\langle y0\rangle)$ above).

$$\dfrac{\dfrac{1.\quad (\boldsymbol{\nu}\, y)(R \mid \mathsf{Ref}\langle yn\rangle) \vdash \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ (\boldsymbol{\nu} y)x\ \mathsf{in}\ (B \wedge !y = n) \quad (\ldots)}{2.\quad \overline{z}(x)(\boldsymbol{\nu}\, y)(R \mid !y = n) \vdash \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ (\boldsymbol{\nu} y)(\boldsymbol{\nu}\, x)\ \mathsf{in}\ (z = (x)^\uparrow \wedge B \wedge\ !y = n) \quad (\mathsf{Sel})}}{3.\quad O \vdash \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ \forall i.[u \bullet i/z](\boldsymbol{\nu} y)(\boldsymbol{\nu}\, x)\ \mathsf{in}\ (z = (x)^\uparrow \wedge B \wedge\ !y = n) \quad (\mathsf{In\text{-}n})}$$

as required. The same inference can also be done using the free output rule. For this purpose we use the following structural equality.

$$\overline{z}(x)(\boldsymbol{\nu}\, y)(R \mid \mathsf{Ref}\langle yn\rangle) \ \equiv\ (\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)(\overline{z}\langle x\rangle \mid R \mid \mathsf{Ref}\langle yn\rangle)$$

In general, because of the form of the opening rule, we should always infer the linear output rule after extending the ν-binders to the outermost (first opaque names, then reference names). The use of $\equiv$ may not be found satisfactory. One the one hand, we can easily fix this by making (Par) rules to include the opening rules as we did for bound output rules. On the other, as far as we work in such a normal form the use of $\equiv$ is not necessary; further this normal form is precisely what we always obtain when we do a natural encoding of program phrases into processes. We now infer as follows (we give a step-by-step inference to make the use of open names clear). Below we let $O' \stackrel{\text{def}}{=} \overline{z}\langle x\rangle \mid R \mid \mathsf{Ref}\langle yn\rangle$ for brevity.

1. $\overline{z}\langle x\rangle \vdash \{\mathsf{T}\}\ \textbf{rely}\ (x)^\uparrow = (x')^\uparrow \wedge B[x'/x]\ \textbf{guar}\ z = (x')^\uparrow \wedge B[x'/x]\ \{\mathsf{T}\}$	(FSel)
2. $\overline{z}\langle x\rangle \vdash \{\mathsf{T}\}\ \textbf{rely}\ x = x' \wedge B\ \textbf{guar}\ \exists x'.(z = (x')^\uparrow \wedge B[x'/x])\ \{\mathsf{T}\}$	(Conseq)
3. $\overline{z}\langle x\rangle \vdash \{\mathsf{T}\}\ \textbf{rely}\ \exists x'.(x = x' \wedge B)\ \textbf{guar}\ \exists x'.(z = (x')^\uparrow \wedge B[x'/x])\ \{\mathsf{T}\}$	(Aux-$\exists$)
4. $\overline{z}\langle x\rangle \vdash \{\mathsf{T}\}\ \textbf{rely}\ B\ \textbf{guar}\ \exists x'.(z = (x')^\uparrow \wedge B[x'/x])\ \{\mathsf{T}\}$	(Conseq)
5. $R \vdash \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ B$	(...)
6. $\overline{z}\langle x\rangle \mid R \vdash \{\mathsf{T}\}\ \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ \exists x'.(z = (x')^\uparrow \wedge B[x'/x])\ \wedge\ B\ \{\mathsf{T}\}$	(1,5,Par)
7. $\overline{z}\langle x\rangle \mid R \vdash \{\mathsf{T}\}\ \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ z = (x)^\uparrow\ \wedge\ B\ \{\mathsf{T}\}$	(Conseq)
8. $\overline{z}\langle x\rangle \mid R \mid \mathsf{Ref}\langle yn\rangle \vdash \{\mathsf{T}\}\ \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ z = (x)^\uparrow \wedge B \wedge\ !y = n\ \{\mathsf{T}\}$	(7,Ref-n,Par)
9. $O' \vdash \{\mathsf{T}\}\ \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ (\boldsymbol{\nu}\varepsilon)x\ \mathsf{in}\ (z = (x)^\uparrow\ \wedge\ B\ \wedge\ !y = n)\ \{\mathsf{T}\}$	(Open-thread)
10. $(\boldsymbol{\nu}\, y)O' \vdash \{\mathsf{T}\}\ \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ (\boldsymbol{\nu} y)x\ \mathsf{in}\ (z = (x)^\uparrow \wedge B \wedge\ !y = n)\ \{\mathsf{T}\}$	(Hide-thread)
11. $(\boldsymbol{\nu} xy)O' \vdash \{\mathsf{T}\}\ \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)\ \mathsf{in}\ (z = (x)^\uparrow \wedge B \wedge\ !y = n)\ \{\mathsf{T}\}$	(Hide-thread)
12. $O \stackrel{\text{def}}{=} !u(z).O' \vdash \textbf{rely}\ \mathsf{T}\ \textbf{guar}\ \forall n.([u \bullet n/x]E(z, n))$.	(In-n)

A somewhat roundabout series of inferences from Line 1 to Line 7 are to make the cancellation of B in the hypothesis possible while retaining it in the guarantee formula, and is not necessary if we used x in the guarantee formula (which, as noted in Remark 1, Page 35, and as used in §8.8, is a sound and convenient practice). This part (or its simpler counterpart) assumes names are open. Note we cannot hide x before y in Lines 10–11.

11.9 Distinction and Aliasing

We conclude this section by outlining how we can fully generalise the type structure for local state. We recall, since the introduction of statefulness in Section 9, we have been working with the set of types in which a type (either for a reference or for a non-reference) never carries a reference type. Thus we never have had such a type as $\mathit{ref}(\mathit{ref}(\tau))$. Since a reference behaviour can be encoded as a replicated agent, as we have shown in (9.3), Section 9.12, Page 125, this in fact does not lose any generality: we can fully encode a language which does send a reference via a reference (as in ML) in the calculus without it. Another observation is that, in the standard object-oriented languages (standard in the sense that the fundamental features of Simula are inherited), a direct reference to a variable never appears, at least in the usual programming practice.

But it is in fact easy, once we have a machinery of local state we have been discussing in this section, to specify and reason about processes in which types directly carry reference types. Further, at the level of encoding, this is the only way we can obtain a full abstraction for languages with the corresponding type structure (this is related to the so-called "good variable" problem). In the extended type structure, instead of using an indirect encoding as found in (9.3), we are directly representing the situation where a reference is carried in communication. This lack of indirection does not result in any difficulties in specifications, but it does lead to the use of an additional notion in semantics and proof rules.

We start from a simple example. Assume a reference x contains 0, and two variables y and z contain x. We can write the situation thus:

$$P \quad \stackrel{\text{def}}{=} \quad \mathsf{Ref}\langle yx\rangle \mid \mathsf{Ref}\langle zx\rangle \mid \mathsf{Ref}\langle x1\rangle \tag{11.40}$$

The process may as well be typed as:

$$\vdash P \triangleright x\!:\!\mathit{ref}(\mathbb{N}),\ y\!:\!\mathit{ref}(\mathit{ref}(\mathbb{N})),\ z\!:\!\mathit{ref}(\mathit{ref}(\mathbb{N})) \tag{11.41}$$

where we use the grammar of types in which any type can carry a reference type (regarding a reference type as a type with the extended replicated mode, similarly for its dual). An assertion for this typed agent may be given as follows.

$$A \quad \stackrel{\text{def}}{=} \quad !x = 1 \ \wedge\ !y = x \ \wedge\ !z = x. \tag{11.42}$$

The assertion is simple. If we hide x in P, then we also wish to hide x in the above assertion. Since a reference name is opaque, if we restrict it, we may as well use an opening formula, leading to:

$$\mathsf{open}\,(\boldsymbol{\nu}x)yz\,\mathsf{in}\,A \tag{11.43}$$

One may wonder why we do not simply use the notation $(\boldsymbol{\nu}\,x)A$. But consider

$$P' \quad \stackrel{\text{def}}{=} \quad P \mid\ !u(c).\overline{x}\mathsf{read}(c')c'(n).\overline{x}\mathsf{write}(n{+}1,c'')c''.\overline{c}\langle n\rangle. \tag{11.44}$$

and assume we should write a specification for $(\boldsymbol{\nu}\,x)P'$, instead of $(\boldsymbol{\nu}\,x)P$. Then we need to open u together with y and z, reaching:

$$\mathsf{open}\,(\boldsymbol{\nu} x)uyz\,\mathsf{in}\,(A \wedge \forall n.\{!x = n\}u \bullet \varepsilon \rightsquigarrow n\{!x = n{+}1\}) \tag{11.45}$$

so we do need to open reference names if they ever carry other hidden reference names, because of this critical problem with stateful agents, sharing. The significance of the use of opening formulae is a uniform and general treatment of issues associated with sharing, among others stateful, possibly higher-order, data structures which are shared in a complex way via references.

Consistent treatment of behaviour with higher-order stateful types, however, does not end with the introduction of opening formulae for references. Consider the following process.

$$Q \stackrel{\mathrm{def}}{=} u(yz).\overline{y}\mathsf{read}(c)c(n).\overline{z}\mathsf{read}(c')c'(n').\overline{w}\langle n + n'\rangle. \tag{11.46}$$

we can compose Q with the following agent:

$$R \stackrel{\mathrm{def}}{=} \overline{u}\langle aa\rangle|\mathsf{Ref}\langle a1\rangle \tag{11.47}$$

As a illustration $Q|R$ reduces as follows:

$$\begin{aligned} R|Q &\longrightarrow \mathsf{Ref}\langle a1\rangle \,|\, \overline{a}\mathsf{read}(c)c(n).\overline{a}\mathsf{read}(c')c'(n').\overline{w}\langle n + n'\rangle \\ &\longrightarrow \mathsf{Ref}\langle a1\rangle \,|\, \overline{a}\mathsf{read}(c')c'(n').\overline{w}\langle 1 + n'\rangle \\ &\longrightarrow \mathsf{Ref}\langle a1\rangle \,|\, \overline{w}\langle 2\rangle \end{aligned}$$

The question is how we can specify the property of Q, for which we should say:

> *if it interacts with an agent such as R, its y and z would collapse, and outputs* 2.

Clearly we cannot use the simplifying idea which has always worked so far, the identity distinction (which, as we recall, is the assumption that all reference/opaque names are always mutually distinct): we should now use a non-trivial distinction for specifying agents.

To proceed with the specification with distinction one by one, let us consider the body of Q, the process $Q' \stackrel{\mathrm{def}}{=} \overline{y}\mathsf{read}(c)c(n).\overline{z}\mathsf{read}(c')c'(n').\overline{w}\langle n{+}n'\rangle$. A natural specification would be:

$$B \quad \stackrel{\mathrm{def}}{=} \quad (y = z \wedge !y = 1) \;\supset\; w = (2)^{\uparrow} \tag{11.48}$$

Note we used an equation on reference names. How does Q conform to B? In Q, y and z looks distinct: if so, the assumption B may nod hold for Q', which however means B is valid for Q'. But this assumption does hold for $Q'[z/y]$. We may write this situation with the following sequent.

$$Q' \vdash^{\mathsf{T}} \{!y = 1 \wedge !z = 1 \wedge y = z\}\mathbf{rely}\,\mathsf{T}\;\;\mathbf{guar}\,w = (2)\,\mathbf{rely}\,\mathsf{T} \tag{11.49}$$

The mark "T" means we are considering the null distinction — in which all names can be collapsed into each other as far as well-typedness is considered. in this case, y and z can be collapsed: and the judgement asserts the assertion is true as far as a concrete distinction conforms to this, that is either y and z are coalesced, or if they are identical.

1. Assume y and z are distinct. In this case, we check:
$$Q' \vdash \{\mathsf{T}\}\mathbf{rely}\,!y=1 \wedge !z=1 \wedge y=z\,\mathbf{guar}\,w=(2)\,\mathbf{rely}\,\mathsf{T} \tag{11.50}$$
where we assume, as before, the identity distinction. Then the precondition is false, so the sequent is trivially valid.
2. Assume y and z are identical. Then we take a fresh v and check:
$$Q'[vv/yz] \vdash \{\mathsf{T}\}\mathbf{rely}\,!v=1 \wedge !v=1 \wedge v=v\,\mathbf{guar}\,w=(2)\,\mathbf{rely}\,\mathsf{T} \tag{11.51}$$
which, again taking it under the identity distinction, is indeed valid.

From (11.49) we may naturally infer:
$$u(yz).Q' \vdash \{\mathsf{T}\}\mathbf{rely}\,\exists yz.(u=(yz)^{\uparrow} \wedge y=z \wedge\ !y=1)\ \ \mathbf{guar}\,w=(2)^{\uparrow}\,\{\mathsf{T}\} \tag{11.52}$$
On the other hand we can check, just as above:
$$Q' \vdash^{y\neq z}\{!y{=}1 \wedge !z{=}2\}\mathbf{rely}\,\mathsf{T}\ \ \mathbf{guar}\,w=(3)\,\mathbf{rely}\,\mathsf{T} \tag{11.53}$$
Note we have stipulated a non-trivial distinction $y \neq z$. From (11.53) we obtain:
$$u(yz).Q' \vdash \{\mathsf{T}\}\mathbf{rely}\,\exists yz.(u=(yz)^{\uparrow} \wedge y\neq z \wedge\ !y=1 \wedge\ !z=2)\ \ \mathbf{guar}\,w=(3)^{\uparrow}\,\{\mathsf{T}\}$$
where we have added the distinction to the rely condition. Note the reasoning with a non-trivial distinction is rigorously stratified on the reasoning with an identity distinction. As can be seen, the same idea easily extends to the setting where a process are augmented with the construct for judging the equality of names (the so-called match construct).

We formally summarise the constructions. At the level of types, we allow types to carry any reference types (we may as well specify potential reference names to be instantiated, though we do not consider the restriction here). The typing rules stay as they are. The grammar of formulae also stays as in its original form, except we allow names of reference types to be opened in an opening formula. A *distinction* is a set of inequations over equi-typed names. We let $\mathcal{D}, \ldots$ range over distinctions. We write $\sigma \prec \mathcal{D}$ if a substitution σ conforms to $\mathcal{D}$ in the obvious sense. Then we set:
$$P \models^{\mathcal{D}}\{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}\ \equiv\ \forall\sigma\prec\mathcal{D}.\,P\sigma \models \{C\sigma\}\,\mathbf{rely}\,A\sigma\,\mathbf{guar}\,B\,\{C'\sigma\}$$
Note we have used all the machinery of the local (hence open, hence pure affine, hence linear) process logic: such stratification is the key to both clear semantics and reasoning principles in the present framework.

The proof rules are also stratified on the basis of the existing proof rules for local state using name substitutions conforming to substitutions,[17] except we

[17] We can in fact check conformance to distinction at each inference step in a single derivation (without working with several derivations with distinct distinctions), ranging the main co-replicated/read-write output subject over other possible names.

$$(\mathsf{In}^{\downarrow})\quad (A[(\vec{y})^{\uparrow}/x] \supset \mathcal{D}\restriction\vec{y})$$
$$\frac{P \vdash^{\mathcal{D}} \{C^{\text{-}\vec{y}}\}\,\mathbf{rely}\,A[(\vec{y})^{\uparrow}/x]\,\mathbf{guar}\,B\,\{C'^{\text{-}\vec{y}}\}}{x(\vec{y}).P \vdash^{\mathcal{D}/\vec{y}}\{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}}$$

$$(\mathsf{In}^{!})\quad (\forall\vec{y}.(\,B_1 \wedge \mathcal{D}\restriction\vec{y} \supset [\{C\}x\bullet\vec{y}\{C'\}/z]B_2\,) \supset B)$$
$$\frac{P \vdash^{\mathcal{D}}\{C\}\,\mathbf{rely}\ A^{\text{-}\vec{y}}\wedge B_1^{\vec{y}}\,\mathbf{guar}\ B_2\ \ \{C'\}}{!x(\vec{y}z).P \vdash^{\mathcal{D}/\vec{y}}\mathbf{rely}\ A\ \mathbf{guar}\ B}$$

Fig. 46. Proof Rules for Input with Distinction

change the input rules as given in Figure 46. First we consider the linear input rule. For simplicity, let us consider the case when a unary linear input only carries reference names (the general case is easily obtained). Then we reason:

$$(\mathsf{In}^{\downarrow})\ \frac{\begin{array}{c}P \vdash^{\mathcal{D}} \{C^{\text{-}\vec{y}}\}\,\mathbf{rely}\,A[(\vec{y})^{\uparrow}/x]\,\mathbf{guar}\,B\,\{C'^{\text{-}\vec{y}}\}\\ A[(\vec{y})^{\uparrow}/x] \supset \mathcal{D}\restriction\vec{y}\end{array}}{x(\vec{y}).P \vdash^{\mathcal{D}/\vec{y}}\{C\}\,\mathbf{rely}\,A\,\mathbf{guar}\,B\,\{C'\}}$$

In the rule, $\mathcal{D} \restriction \vec{y}$ denotes the formula which we obtain by restricting $\mathcal{D}$ to $\vec{y}$. $\mathcal{D}/\vec{y}$ is the result of taking off $\vec{y}$ from $\mathcal{D}$.The rule says that if the assumption is already indicating the distinction to be abstracted, then we can safely abstract names in that distinction. Similarly for replication we infer:

$$(\mathsf{In}^{!})\ \frac{\begin{array}{c}P \vdash^{\mathcal{D}}\{C\}\,\mathbf{rely}\ A^{\text{-}\vec{y}}\wedge B_1^{\vec{y}}\,\mathbf{guar}\ B_2\ \ \{C'\}\\ \forall\vec{y}.(\,\forall\vec{y}.(\,B_1 \wedge \mathcal{D}\restriction\vec{y} \supset [\{C\}x\bullet\vec{y}\{C'\}/z]B_2\,) \supset B\end{array}}{!x(\vec{y}z).P \vdash^{\mathcal{D}/\vec{y}}\mathbf{rely}\ A\ \mathbf{guar}\ B}$$

which again says that, if the conclusion to be obtained already assumes the distinction, then we can abstract them. We can easily check these rules conform to the preceding examples. These rules conform to the semantics of distinction stipulated above. Further discussions on systems with non-trivial distinction will be given in a later version of the present exposition.

12 Program Logics for Higher-Order Imperative Programs (2): Languages with Local State

12.1 A Simple Object-Oriented Language

Though one often forgets, object-oriented languages as we know now are direct descendants of procedural languages (in fact, Simula-67, introduced by Dahl and Nygarrd, is based on Algol-60). The essential aspect of these languages is that they add, to the idioms of procedural languages, a notion of user-defined data structures which encapsulate both code and state for flexible program organisation. This aspect is elucidated in several early expositions on the topic, for example [28]. Classes and objects are used for organising complex programs hierarchically, where a class is a template for a data structure which combines code and local state, and objects are instances of this data structure, created at will at arbitrary program points.

In the following we present a simple program logic for such a language. Our purpose is expository, so that we shall only cover bare minimum features of object-oriented languages. Nevertheless the language includes such notions as object generation, object sharing, scope extrusion for a locally created object, and object invocation with side effects, making its logical articulation challenging. We use the process encodings and the associated process logic for elucidating these features, leading to a simple program logic which indeed smoothly extends the logic for procedural programs discussed in §10.7. Our intention is solely to carry out a small theoretical experiment: for practical adaptation of the presented framework, various engineering concerns need be carefully examined.

We use the following syntax. First, expressions, ranged over by $e, \ldots$, are given by the grammar:

$$e \quad ::= \quad x \;\mid\; \mathsf{new}\ \mathsf{C}(\vec{e}) \;\mid\; e.l(\vec{e}) \;\mid\; \mathsf{this} \;\mid\; \mathsf{nil} \;\mid\; e+e' \;\mid\; e-e' \;\mid\; e=e' \;\mid\; \ldots$$

In object invocation $e.l(\vec{e})$, e is an object to be invoked, l is a *method name* and $\vec{e}$ is a vector of arguments. A new object creation $\mathsf{new}\ \mathsf{C}(\vec{e})$ creates an object of a class C with the initial values $\vec{e}$. this indicates the object itself (in the definition of a class for the object). nil is a fixed neutral reference value.

The grammar of programs, ranged over by $P, Q, \ldots$, are given as follows, together with class definitions, ranged over by $L, \ldots$.

$$\begin{array}{rcl} P & ::= & \mathtt{skip} \;\mid\; x := e \;\mid\; P;Q \;\mid\; \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \;\mid\; \mathtt{while}\ e\ \mathtt{do}\ P \;\mid \\ & & \mathsf{var}\,\alpha\,x := e\ \mathsf{in}\ P \;\mid\; \mathsf{return}\ e \;\mid\; \{L_i\}\ \mathsf{in}\ P \\ L & ::= & \mathsf{class}\ \mathsf{C}(\vec{\alpha}\ \vec{x})\{\beta_i\ \mathtt{l}_i(\vec{\gamma}_i\ \vec{x}_i)P_i\} \end{array}$$

All constructs come from the procedural language discussed in Section 10.7 except class definitions which we discuss below. $\{L_i\}\ \mathsf{in}\ P$ closes consists of a family $\{L_i\}$ of (possibly recursive) class definitions and a main program P which uses them. A class definition includes *method definitions*, each of the form $\beta\ \mathtt{l}(\vec{\gamma}\ \vec{x})P$. l is a method name, $\vec{x}$ is formal parameters typed $\vec{\gamma}$, β is a return type, and P

$$\text{(new)}\ (\mathit{\Gamma} \vdash \mathsf{C} : \vec{\alpha})\quad \frac{\mathit{\Gamma};\mathit{\Upsilon} \vdash e_i : \alpha_i}{\mathit{\Gamma};\mathit{\Upsilon} \vdash \mathsf{new}\ \mathsf{C}(\vec{e}) : \mathsf{C}} \qquad \text{(invoke)}\ (\mathit{\Gamma}(\mathsf{C}) \vdash l : \vec{\alpha} \overset{\mathit{\Upsilon}}{\Rightarrow} \beta)\quad \frac{\mathit{\Gamma};\mathit{\Upsilon},\mathit{\Upsilon}' \vdash e : \mathsf{C},\quad \mathit{\Gamma} \vdash \vec{e} : \vec{\alpha}_i}{\mathit{\Gamma};\mathit{\Upsilon}' \vdash e.\mathtt{l}(\vec{e}) : \beta} \qquad \text{(this)}\quad \frac{-}{\mathit{\Gamma}, \mathsf{this} : \mathsf{C}\ ;\ \mathit{\Upsilon} \vdash \mathsf{this} : \mathsf{C}}$$

$$\text{(skip)}\ \frac{-}{\mathit{\Gamma};\mathit{\Upsilon} \vdash \mathtt{skip}} \qquad \text{(assign)}\ \frac{\mathit{\Gamma};\mathit{\Upsilon} \vdash e : \mathit{\Upsilon}(x)}{\mathit{\Gamma};\mathit{\Upsilon} \vdash x := e} \qquad \text{(seq)}\ \frac{\mathit{\Gamma};\mathit{\Upsilon} \vdash P_1,\quad \mathit{\Gamma};\mathit{\Upsilon} \vdash P_2}{\mathit{\Gamma};\mathit{\Upsilon} \vdash P_1; P_2} \qquad \text{(if)}\ \frac{\mathit{\Gamma};\mathit{\Upsilon} \vdash e : \mathbb{B},\quad \mathit{\Gamma};\mathit{\Upsilon} \vdash P_{1,2}}{\mathit{\Gamma};\mathit{\Upsilon} \vdash \mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2}$$

$$\text{(while)}\ \frac{\mathit{\Gamma};\mathit{\Upsilon} \vdash e : \mathbb{B},\quad \mathit{\Gamma};\mathit{\Upsilon} \vdash P}{\mathit{\Gamma};\mathit{\Upsilon} \vdash \mathtt{while}\ e\ \mathtt{do}\ P} \qquad \text{(var-dec)}\ \frac{\mathit{\Gamma};\mathit{\Upsilon}, x : \alpha \vdash P,\quad \mathit{\Gamma};\mathit{\Upsilon} \vdash e : \alpha}{\mathit{\Gamma};\mathit{\Upsilon} \vdash \mathsf{var}\ \alpha\ x := e\ \mathsf{in}\ P} \qquad \text{(return)}\ \frac{\mathit{\Gamma};\mathit{\Upsilon} \vdash e : \alpha}{\mathit{\Gamma};\mathit{\Upsilon} \vdash \mathsf{return}\ e : \alpha}$$

$$\text{(class def)}\ (\mathcal{T} = \langle \vec{\alpha},\ \{l_i : \vec{\beta}_i \overset{\mathit{\Upsilon}_i}{\Rightarrow} \gamma_i\}\rangle)\quad \frac{\forall i.\ \ \mathit{\Gamma}, \mathsf{C} : \mathcal{T}, \mathsf{this} : \mathsf{C}\ ;\ \mathit{\Upsilon}, \vec{x} : \vec{\alpha}, \vec{y}_i : \vec{\beta}_i, \vdash P_i : \gamma_i}{\mathit{\Gamma} \vdash \mathsf{C}(\vec{\alpha}\ \vec{x})\{\gamma_i\ \mathtt{l}_i(\vec{\beta}_i\ \vec{y}_i)P_i\} : \mathsf{C}} \qquad \text{(class dec)}\ (\mathsf{dom}(\mathit{\Delta}) = \{L_i\})\quad \frac{\mathit{\Gamma}, \mathit{\Delta} \vdash L_i : \mathsf{C_i},\quad \mathit{\Gamma}, \mathit{\Delta}\ ;\ \mathit{\Upsilon} \vdash P}{\mathit{\Gamma};\mathit{\Upsilon} \vdash \{L_i\}\ \mathsf{in}\ P}$$

Fig. 47. Typing Rules for Simple Objects

is a method body. In $\mathsf{C}(\vec{\alpha}\ \vec{x})\{M_i\}$, $\{M_i\}$ is a finite number of method definitions for distinct method names; while $\vec{\alpha}\ \vec{x}$ declares instance variables $\vec{x}$ of types $\vec{\alpha}$. Instead of having code for initialisation, the arguments of a "new" expression become their initial values. This simplification is practically unsatisfactory (in particular it exposes signature of instance variables) but gives the simplest setting for presentation on whose basis further ramifications are easily possible.

For types, we use the following grammar.

$$\alpha\ ::=\ \mathtt{int} \mid \mathtt{bool} \mid \mathsf{C} \qquad\qquad \mathcal{T}\ ::=\ \langle \vec{\alpha}, \{\mathtt{l_i} : \vec{\beta}_i \overset{\mathit{\Upsilon}_i}{\Rightarrow} \gamma_i\}\rangle$$

$\alpha, \beta, \ldots$ are types given to variables and expressions, which are either atomic types or class names. $\mathcal{T}, \ldots$ are types given to classes, called *class types*. In $\langle \vec{\alpha}, \{\mathtt{l_i} : \vec{\beta}_i \overset{\mathit{\Upsilon}_i}{\Rightarrow} \gamma_i\}\rangle$, $\vec{\alpha}$ is the types of instance variables, while $\{\mathtt{l_i} : \vec{\beta}_i \overset{\mathit{\Upsilon}_i}{\Rightarrow} \gamma_i\}$ gives the types of methods. In $\vec{\alpha} \overset{\mathit{\Upsilon}}{\Rightarrow} \beta$, $\vec{\alpha}$ are the types of arguments, β is the type of a return value, and $\mathit{\Upsilon}$ is a finite map from variables to types, called *effect* of *effect typing* (which can in fact be inferred from other information and is only needed for typing assertions, cf. §10.7). We write $\mathcal{T} \vdash l : \vec{\alpha} \overset{\mathit{\Upsilon}}{\Rightarrow} \beta$ if $l : \vec{\alpha} \overset{\mathit{\Upsilon}}{\Rightarrow} \beta$ is in $\mathcal{T}$. The typing uses three kinds of sequents: $\mathit{\Gamma};\mathit{\Upsilon} \vdash e : \alpha$ for expressions, $\mathit{\Gamma};\mathit{\Upsilon} \vdash P$ for programs, and $\mathit{\Gamma} \vdash L : \mathsf{C}$ for class definitions. In these sequents $\mathit{\Gamma}$ maps finite class names to class types as well as, as needed this to a class name, while $\mathit{\Upsilon}$ maps variables to their types. The typing rules are given in Figure 47. We assume the base (either of the form Γ or $\Gamma; \Upsilon$) in each sequent is well-formed in that all occurring class names are in the domain of Γ and each variable a unique type including in its effect typing.

12.2 Assertions for Simple Objects

In the following we present a few programming examples and illustrate how we can specify their properties guided by the process logic via encoding.

As a simple example of a program, we give a class definition of Inc with a method do. Below and henceforth we freely insert braces for clarity.

```
class Inc(int x)
{
    int do()
    {
        x := x + 1;
        return x
    }
}
```

For example, we can generate an object of this class by a command:

$$y := \mathsf{new}\ \mathsf{Inc}(1) \tag{12.1}$$

which generates an object of class Inc with the initial parameter 1 and stores the resulting object to the variable y.

How can we write down a specification for the class Inc? Or the new object generation in (12.1)? Let us take a look at a process encoding of the class definition.

$$P \stackrel{\text{def}}{=} \ !\,\mathsf{Inc}(nc).\,(\boldsymbol{\nu} ox)(\,!o(c').\overline{x}\mathsf{read}(e)e(m).(\boldsymbol{\nu} f)\overline{x}\mathsf{write}\langle m{+}1, f\rangle f.\overline{c'}\langle m{+}1\rangle \mid \mathsf{Ref}\langle xn\rangle \mid \overline{c}\langle o\rangle\,) \tag{12.2}$$

where we regard the class name Inc as a channel. In this encoding, we find two nested replications, one for an object generation and, inside it, another for an object itself. The outermost replication, when invoked at a channel Inc with a parameter n and a return channel, generates a new object (which is another replication) and a local reference storing n, while passing back the generated name to the invoker via the return channel. for the internal replication, when invoked at o with a return channel, it will read a number from x, writes back its increment to x, as well as returning that number to the invoker.

The encoding (12.2) suggests we may regard "classes" and "objects" uniformly as replicated behaviours; and that we can use assertions for replication in the process logic for specifying their properties. With simplification due to the fixed shape of interactions, we arrive at the following specification B for the class definition above.

$$B \stackrel{\text{def}}{=} \forall i \in \mathbb{N}.\ \langle\!\langle \mathsf{Inc} \bullet \mathsf{new}(i)/(\nu x)o \rangle\!\rangle\,(A(o,i) \wedge x = i) \tag{12.3}$$

$$A(o,x) \stackrel{\text{def}}{=} \forall n \in \mathbb{N}.\ \{x = n\}\langle\!\langle o \bullet \mathsf{do}()/z \rangle\!\rangle(z = n{+}1)\{x = n{+}1\} \tag{12.4}$$

Let us start from $A(o,x)$, which uses the formula

$$\{x=n\}\langle\!\langle o \bullet \mathsf{do}()/z\rangle\!\rangle(z=n+1)\{x=n+1\}$$

in which z in "$\langle\!\langle .../z\rangle\!\rangle$" binds its free occurrences in the subsequent formula. "$\{x=n\}$" is called *pre-condition*, "$o \bullet \mathsf{do}()$" *invocation*, "$z=n+1$" *main formula*, and "$\{x=n+1\}$" *post-condition*. The formula as a whole is called *invocation formula*. The constructions closely follow treatment of replicated behaviour in the process logic. The assertion $A(o,x)$ is then read as:

> *For each n, invoking an object o with a method* do *with null arguments at the initial state $x=n$ results in the final state $x=n+1$ and a return value z, the latter satisfying $z=n+1$.*

which describes the behaviour of the "code" part of an object belonging to Inc.

Now an object may have not only code but also local state. In the present case, x is a local reference inside an object (as is clearly seen from the encoding in (12.2), where x is ν-bound). Thus, when the class Inc generates an object, the latter's local state may as well be generated. In the main assertion B in (12.3), the formula $\langle\!\langle \mathsf{Inc} \bullet \mathsf{new}(i)/(\boldsymbol{\nu}\, x)o\rangle\!\rangle(A(o,x) \wedge x=i)$ precisely describes a new object creation. In the formula, both $(\boldsymbol{\nu}\, x)$ and o in "$\langle\!\langle .../(\boldsymbol{\nu}\, x)o\rangle\!\rangle$" are binders, binding their occurrences in "$A(o,x) \wedge x=i$" which follows them. In the formula, the class name Inc is in the position of the invoked, which is naturally understood from the process encoding (12.2) where the class name is used as a replicated channel name. Thus B as a whole may be read as:

> *For each i, if we invoke a class* Inc *for generation of an object with an argument i, then it results in a new object o satisfying $A(o,i)$.*

Thus we have reached the specification of the class Inc, guided by the process logic[18]. The uniform treatment of these "replicated behaviours" easily extends to treatment static methods in general, including those with non-trivial side effects.

Let us now consider how we can express the command in (12.1) which generates a new instance of Inc and assigns it to y, assuming the presented code for Inc. The net effect of the command is a write on y: so its behaviour may naturally be specified by the state of y after its execution. y stores an object which is newly generated, whose behaviour is given by its code and a local state. We now give a specification for the command, using a judgement which is close to the judgement in the Hoare logic. Below we use B as an assumption, which is the assertion for Inc treated above.

$$\{\mathsf{T}\}\ B\ \vdash\ y := \mathsf{new}\ \mathsf{Inc}(1)\ \{\,\mathsf{open}\,(\boldsymbol{\nu} x)y\,\mathsf{in}\,(\,A(y,x)\ \wedge\ x=1)\,\} \tag{12.5}$$

[18] For reference we list the corresponding assertions for P.

$$A_p(o,x) \stackrel{\text{def}}{=} \forall n.\{!x=n\}\langle\!\langle o \bullet \varepsilon/z\rangle\!\rangle(z=(n+1)^{\uparrow})\{!x=n+1\}$$

$$B_p(u) \stackrel{\text{def}}{=} \forall i \in \mathbb{N}.\ \langle\!\langle u \bullet i/(\nu x)o\rangle\!\rangle(\,A_p(o,x)\ \wedge\ !x=i\ ).$$

Note there is no distinction between object invocation and a new object generation in these formulae: they are both instances of invocation of replicated agents.

The formula in the post condition:

$$\mathsf{open}\,(\boldsymbol{\nu}x)y\,\mathsf{in}\,(\,A(y,x)\ \wedge\ x=1)$$

is called an *opening formula*, where y is the *opened name* and x is the (introduced) *local state*. It says that:

> *The variable y stores an object such that (1) it has the code specified by $A(y,x)$, and (2) its (instance) variable x has the initial state 1, and (3) x is local to the behaviour.*

Thus the assertion in (12.5) as a whole says that, at whatever initial state we start, the execution of the given command leads to the state where y refers to an object satisfying the above description. The freshness of the created object is expressed by the ν-binding of x in the opening formula, by which we know its internal state is distinct from any other state, local or global.

In addition to object creation, one of the prominent features of object-orientation is the capability to represent sharing of information via object references, allowing the construction of complex data structures. Let us consider how we can assert for such sharing. We use the following class which represents a list whose "car" (head) stores an instance of Inc we have just defined.

```
class List(Inc p,  List l)
{
    Inc getCar() {return p}
    List getCdr() {return l}
    Inc setCar(Inc p') {p := p';  return p'}
    List setCdr(List l') {l := l';  return l'}
}
```

If we had the void type, `setCar` and `setCdr` do not have to return values: here we let them return the new local state. Following what we did for Inc class, we arrive at the following specifications for List. Below we set $C(p',l')$ to be $(p=p'\ \wedge\ l=l')$.

$$\begin{aligned}
E_1(o,p,l) &\stackrel{\text{def}}{=} \forall p',l'.\ (\{C(p',l')\}\langle\!\langle o\bullet \mathtt{getCar}()/z\rangle\!\rangle(z=p')\{C(p',l')\})\\
E_2(o,p,l) &\stackrel{\text{def}}{=} \forall p',l'.\ (\,\{C(p',l')\}\langle\!\langle o\bullet \mathtt{setCar}(p'')/z\rangle\!\rangle(z=p')\{C(p'',l')\}\,)\\
E_3(o,p,l) &\stackrel{\text{def}}{=} \forall p',l'.\ (\{C(p',l')\}\langle\!\langle o\bullet \mathtt{getCdr}()/z\rangle\!\rangle(z=l')\{C(p',l')\})\\
E_4(o,p,l) &\stackrel{\text{def}}{=} \forall p',l'.\ (\,\{C(p',l')\}\langle\!\langle o\bullet \mathtt{setCdr}(l'')/z\rangle\!\rangle(z=p')\{C(p',l'')\}\,)\\
E(o,p,l) &\stackrel{\text{def}}{=} E_1\wedge E_2\wedge E_3\wedge E_4\\
G &\stackrel{\text{def}}{=} \forall pl.\,(\,\langle\!\langle \mathsf{List}\bullet \mathsf{new}(p'l')/(\boldsymbol{\nu}\,pl)o\rangle\!\rangle\,(\,E(o,p,l)\wedge C(p',l')\,)
\end{aligned}$$

Above each E_i gives a specification of each method, E collects all of them, and finally G gives the specification for the class List, saying it would generate a new object with two local references which initially store the parameters the class receives when it is invoked. The specifications follow the same pattern as the one for Inc and should be read easily.

Given the two classes we have defined above, let us consider the following programs, which together demonstrate a simple, but non-trivial, sharing.

$$R_1 \stackrel{\text{def}}{=} x := \mathsf{new\ Inc}(1)\ ;\ y := \mathsf{new\ List}(x, \mathsf{nil})\ ;\ z := \mathsf{new\ List}(x, y)$$
$$R_2 \stackrel{\text{def}}{=} u := y.\mathtt{getCar}()\ ;\ v := z.\mathtt{getCar}()\ ;\ w := (u.\mathsf{do}()) + (v.\mathsf{do}())$$

R_1 creates a new incrementor (let it be i) with the initial state 1, stores it in x, then, using i, creates two lists, one with the value i and the tail nil, another with the same value i and the tail being the list just created. R_2 then asks two objects stored in y and z (which in fact is a single object) to return numbers, and stores their sum to w. Our task is to precisely describe the behaviour of $R_1; R_2$ using a logical language.

We start from a description of the state immediately after executing R_1, which already demonstrates a non-trivial sharing of objects. How can we represent the fact that an object stored in x is shared by the two objects stored in y and z, in the shape of the latter's "car" values? This means two objects, denoted by y and z, share their local state: since local state of an object is specified using an opening formula, this suggests we should open several names at the same time in an opening formula. Using this idea, we can write down the postcondition of R_1 in the following way.

$$\begin{aligned} H \quad \stackrel{\text{def}}{=} \quad & \mathsf{open}\,(\boldsymbol{\nu}\, lpl'p'r)\, xyz \ \mathsf{in}\ (\, E(y,l,p) \wedge E(p',l,p) \wedge E(z,l',p') \wedge \\ & \qquad A(x,r) \wedge A(l,r) \wedge A(l',r) \wedge p{=}\mathsf{nil} \wedge r{=}1\,) \end{aligned}$$

We can equivalently open yz and x successively in a nested fashion; anyway opening all three names are indeed necessary for representing the proper sharing of state r and behaviour $A(x,r)$. In this way the predicate directly represents sharing, reflecting the sharing induced by programs. Similarly, assuming the precondition H, if R_2 is executed, we shall reach:

$$\begin{aligned} H' \stackrel{\text{def}}{=} \ & \mathsf{open}\,(\boldsymbol{\nu}\, lpl'p'r)\, xyzuv \ \mathsf{in}\ (\, E(y,l,p) \wedge E(p',l,p) \wedge E(z,l',p') \wedge \\ & A(x,r) \wedge A(l,r) \wedge A(l',r) \wedge A(u,r) \wedge A(v,r) \wedge p{=}\mathsf{nil} \wedge r{=}3\,) \wedge w{=}5\,. \end{aligned}$$

Note that, for the numerical calculation of R_2 — $(u.\mathsf{do}()) + (v.\mathsf{do}())$ – to be carried out, it relies on the states $A(u,r)$ and $A(v,r)$ (implied by the above formula), which in fact come from the preceding assignments by R_2 itself. In H', r is now incremented to be 3, recording two invocations by R_2. We can now write down the specification of each of $R_{1,2}$ as follows.

$$\{\mathsf{T}\}\ B \wedge G \vdash R_1\ \{H\}, \qquad \{H\}\ B \wedge G \vdash R_2\ \{H'\}, \tag{12.6}$$

One can now use the standard proof rule for sequential composition to conclude:

$$\{\mathsf{T}\}\ B \wedge G \vdash R_1; R_2\ \{H'\}$$

which records not only the result of numerical calculation by this program, but also shared data structure it constructed.

The reader may hopefully be at this point persuaded that the opening formulae do a reasonable job for representing locality and sharing. But one may still wonder how in the world we can *infer* such assertions, especially given the state r, which plays a crucial role, is a bound occurrence: in fact, it is not even clear whether r in H and r in H' denote the same reference, given both occurrences are bound. But if locality of these variables is a problem, we may as well try to reason about them *by making them non-local.* Let us set:

$$\begin{array}{lcl} H_0 & \stackrel{\text{def}}{=} & E(y,l,p) \wedge E(p',l,p) \wedge E(z,l',p') \wedge \\ & & A(x,r) \wedge A(l,r) \wedge A(l',r) \wedge p = \mathsf{nil} \wedge r = 1 \\ H'_0 & \stackrel{\text{def}}{=} & E(y,l,p) \wedge E(p',l,p) \wedge E(z,l',p') \wedge \\ & & A(x,r) \wedge A(l,r) \wedge A(l',r) \wedge A(u,r) \wedge A(v,r) \wedge p = \mathsf{nil} \wedge r = 3 \wedge w = 5\,. \end{array}$$

These are the "opened" versions of H and H' where (among others) r, l, p, l', p' are now free. Further the predicates E and A now describe how invocation of their methods may change these global states. This allows us to reason about $R_1; R_2$ just as in the standard Hoare logic, reaching:

$$\{H_0\}\,\mathsf{T} \vdash R_2\,\{H'_0\}$$

Then we "close" the pre/post conditions simultaneously, obtaining:

$$\{\,\mathsf{open}\,(\boldsymbol{\nu} lpl'p'r)yzx\,\mathsf{in}\,H_0\,\}\,\mathsf{T} \vdash R_2\,\{\,\mathsf{open}\,(\boldsymbol{\nu} lpl'p'r)yzx\,\mathsf{in}\,H'_0\,\},$$

which is precisely what we specified in (12.6). Later we shall see how this inference can be done using concrete proof rules.

As a final example we show assertions for behaviour which involves "this" and a new object generation. First, consider the following simple code.

```
class Test()
{
   Inc make(int i) { return new Inc(i) }
   Inc make2() { return this.make(2)  }
}
```

where `make2` calls `make` to achieve its function. The class is encoded as:

$$\begin{array}{l} !\,\mathsf{Test}(c).\overline{c}(o)\,!o[\;\; \mathtt{make}(y,i).\overline{\mathsf{Inc}}(i,f)f(x).\overline{y}\langle x\rangle \;\& \\ \qquad\qquad\qquad\quad \mathtt{make2}(y).\overline{o}(2,f)f(x).\overline{y}\langle x\rangle\;] \end{array}$$

where Test and Inc are regarded as channel names. Note "this" is turned into o, the name of the object itself [96]. This suggests "this" may simply be regarded as a name/variable, reaching the following specification for the body of `make2`.

$$\{\mathsf{T}\}\; B \wedge J(\mathsf{this}) \vdash \mathsf{return\ this.}\mathtt{make}(2) \;:^{\nu x}_{u}\; (A(u,x) \wedge !x = 2)\;\{\mathsf{T}\} \qquad (12.7)$$

Above "u" is the *anchor*, naming the value to be returned from this statement. "νx" indicates a fresh local state this expression has created. B and $A(o,i)$ appeared in (12.5) and (12.3) of Page 194. $J(o)$ is given as:

$$J(o) \quad \stackrel{\text{def}}{=} \quad \forall i.\langle\!\langle o \bullet \mathtt{make}(i)/(\boldsymbol{\nu}\, x)z\rangle\!\rangle(A(z,x) \wedge x = 2)$$

which gives the specification of the code (object) part of Test.

The code for the class Test as a whole, which we write L below, would be given the following specification, which uses the assumption B for Inc,

$$B \vdash L : \langle\!\langle \mathsf{Inc} \bullet \mathsf{new}(i)/o\rangle\!\rangle J(o) \tag{12.8}$$

The specification (12.8) now cancels the assumption $J(\mathsf{this})$, which follows the typing ($\mathsf{class\ def}$). Note, however, the encoding of the class (12.2) suggests the assumption on "this" may be cancelled not when the class (external replication) is inferred but when the behaviour of its instance (internal replication together with a recursion) is inferred. We shall see how this observation indeed guides us to a clean proof rule by which we can reach (12.8) starting from (12.7).

Further examples of assertions will be presented later.

Remark 14. (notations for invocations) One may write down invocation formulae using different notations. For example, the formula

$$\{x = n\}\langle\!\langle o \bullet \mathsf{do}()/z\rangle\!\rangle(z = n+1)\{x = n+1\} \tag{12.9}$$

may be written as:

$$\{x = n\} o \bullet \mathsf{do}() \rightsquigarrow n+1\{x = n+1\} \tag{12.10}$$

This notation is however strictly less expressive than the original one because we cannot have a more vague specification such as:

$$\{x = n\}\langle\!\langle o \bullet \mathsf{do}()/z\rangle\!\rangle Even(z)\{x = n+1\} \tag{12.11}$$

where $Even(z)$ says z is even. We can however consider a more general notation along the line of (12.10), which is in fact simply a different (and less terse, but perhaps easier to grasp) way of arranging prefixed substitutions, writing the equivalent of (12.11) in the following way.

$$\{x = n\}\ o \bullet \mathsf{do}() \rightsquigarrow z \text{ WHERE } Even(z)\ \{x = n+1\} \tag{12.12}$$

where z before "$\rightsquigarrow$" binds the occurrences of z in the formula after WHERE. Similarly a new object generation:

$$\langle\!\langle \mathsf{Inc} \bullet \mathsf{new}(i)/(\boldsymbol{\nu}\, x)o\rangle\!\rangle\ (A(o,x) \wedge x = i) \tag{12.13}$$

can be written as:

$$\mathsf{Inc} \bullet \mathsf{new}(i) \rightsquigarrow (\boldsymbol{\nu}\, x)o \text{ WHERE } (\ A(o,x) \wedge x = i\ ) \tag{12.14}$$

which may add pre/post conditions if an object creation involves side effects on, e.g. static variables. As before, occurrences of o in $A'(o,i)$ are bound. The original notations — which we shall use in the following discussions — are more terse and may fit better for mathematical reasoning: while these notations would be easier to grasp their meaning, and may suit practical specifications.

12.3 A Program Logic for Simple Objects: Syntax and Typing

Syntax. In the following we briefly outline the formal syntax of assertions and their typing. We start from the grammar of terms and formulae. Below $e, e', \ldots$ range over terms while $A, A', \ldots$ range over formulae. Below each name (variable) is typed with a *term type* (ranged over by $\tau, \ldots$) discussed later.

$$\begin{array}{lll} e & ::= & i^{\tau} \;\mid\; x^{\tau} \;\mid\; e+e' \;\mid\; e-e' \;\mid\; e=e' \;\mid\; \mathsf{this}^{\tau} \;\mid\; \ldots \\ A & ::= & e_1 = e_2 \;\mid\; \neg A \;\mid\; A_1 \star A_2 \;\mid\; \mathcal{Q}i.A \;\mid\; \mathcal{Q}x.A \;\mid\; \mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}^{\vec{\mathsf{C}}}\,\mathsf{in}\,A \\ & & \{C\}\langle\!\langle e_1 \bullet \mathtt{l}(\vec{e}_2)/(\nu\vec{y})z\rangle\!\rangle A\{C'\} \;\mid\; \langle\!\langle \mathsf{C} \bullet \mathsf{new}(\vec{e})/(\nu\vec{y})z\rangle\!\rangle A \end{array}$$

We distinguish imperative (or program) variables (ranged over by $x, y, \ldots$) and logical variables (ranged over by $i, j, n, \ldots$ etc.) which is semantically important, even though we may not be strict about symbols in examples. Formulae are those of the first-order logic with equality except for the last three predicates. In $\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,A$, $\nu\vec{y}$ are binders by program variables, binding their free occurrences in A. $(\boldsymbol{\nu}\,\vec{y})$ (program variables) and z (a logical variable) of $\langle\!\langle ../(\boldsymbol{\nu}\,\vec{y})z\rangle\!\rangle$ are binders in $\{C\}\langle\!\langle e_1 \bullet \mathtt{l}(\vec{e}_2)/(\nu\vec{y})z\rangle\!\rangle A\{C'\}$ (fresh in C, with scope in A and C') and in $\langle\!\langle \mathsf{C} \bullet \mathsf{new}(\vec{e})/(\nu\vec{y})z\rangle\!\rangle A$ (fresh in C, with scope in A). $(\nu\vec{y})$ may be omitted when $\vec{y}$ is empty. We assume the standard binding convention. If x occurs free in A without being opened (by an opening formula), then we say x occurs *open*.

We use the following judgements, which follow those of the process logic.

1. $\{C\}\, A \vdash e^{\Gamma;\Upsilon;\alpha} :_u^{\nu\vec{w}:\vec{\alpha}} B\,\{C'\}$ for expressions, which reads:

   At the initial state C and relying on the specification for classes as well as this *as given in A, if e is evaluated, then, with fresh local states $\vec{w}$, the state changes into C' and the resulting value, if we name it u, satisfies B.*

   The sequent assumes $\Gamma;\Upsilon \vdash e : \alpha$ and u is fresh from $\Gamma;\Upsilon$. This sequent is also used for programs with return commands. We write $\{C\}\, A \vdash e^{\Gamma;\Upsilon;\alpha} :_u B\,\{C'\}$ for $\{C\}\, A \vdash e^{\Gamma;\Upsilon;\alpha} :_u^{\nu\varepsilon} B\,\{C'\}$.

2. $\{C\}\, A \vdash P^{\Gamma;\Upsilon}\,\{C'\}$ for programs, which reads:

   At the initial state C and relying on the specification for classes as well as this *as given in A, if P runs, then the state changes into C'.*

   The sequent assumes we have $\Gamma;\Upsilon \vdash P$.

3. $A \vdash L^{\Gamma;\mathsf{C}} : B$ for class definitions, which reads:

   If we rely on the specification for classes as given in A, then a class definition L satisfies the property B.

   The sequent assumes we have $\Gamma \vdash L : \mathsf{C}$.

In these sequents, the formulae $\{C\}$, $\{C'\}$, A and B are respectively called *pre-condition*, *post-conditions*, *rely condition* and *guarantee condition*.

Typing. The notion of well-typedness of formulae is an indispensable element of the present logic, needed, among others, for having consistent semantics of assertions: once stipulated, however, we can mostly forget it in specifications. Below we provide the basic ideas for interested readers. First, terms are typed with the following collection of *term types.*

$$\tau \quad ::= \quad \mathbb{B} \quad | \quad \mathbb{N} \quad | \quad \{\mathtt{l_i} : \vec{\beta}_i \stackrel{\Upsilon_i}{\Rightarrow} \gamma_i\}$$

We later show why we need to use $\{\mathtt{l_i} : \vec{\beta}_i \stackrel{\Upsilon_i}{\Rightarrow} \gamma_i\}$. All names in terms are annotated with term types. For formulae (for which we assume the standard binding convention), each formula should be well-typed under some $\Gamma;\Upsilon$ (where Γ is extended with types for logical variables). Rather than giving inductive definition, we give one concrete example. We give a typing to the formula $A'(o,i)$ (cf. (12.5), Page 194), whose formal (fully annotated) version is:

$$\mathsf{open}\,(\boldsymbol{\nu}x)o^{\mathsf{Inc}}\ \mathsf{in}\ (\forall n^{\mathbb{N}}.\ \langle\!\langle\{x^{\mathbb{N}}\!=\!n^{\mathbb{N}}\}\,o^{\tau}\bullet\mathtt{do}()\,\{x^{\mathbb{N}}\!=\!n^{\mathbb{N}}\!+\!1\}/z^{\mathbb{N}}\rangle\!\rangle(z\!=\!n\!+\!1)\wedge x^{\mathbb{N}}\!=\!i)$$

where $\tau \stackrel{\text{def}}{=} \{\mathtt{do} : \varepsilon \stackrel{x:\mathbb{N}}{\Rightarrow} \mathbb{N}\}$ (which is the signature of the code of class Inc, being a singleton of a method with name $\mathtt{do}$, no argument, the return type $\mathbb{N}$, and an effect at x). The formula is well-typed under the following typing:

$$\mathsf{Inc}\!:\!\langle\mathbb{N},\ \{\mathtt{do} : \varepsilon\!\Rightarrow\!\mathbb{N}\}\rangle,\ i\!:\!\mathbb{N}\ ;\ o\!:\!\mathsf{Inc}$$

Note the effect x on τ is eliminated in the type for Inc in this typing, which conforms to the binding of x in the formula. The term type τ gives the type of an object o after getting opened. The type τ is in turn used to type the invocation $\{x^{\mathbb{N}}\!=n^{\mathbb{N}}\}\,\langle\!\langle o^{\tau}\bullet\mathtt{do}()/z\rangle\!\rangle(z=n\!+\!1)\,\{x^{\mathbb{N}}\!=n^{\mathbb{N}}\!+\!1\}$, where we demand, among others, the pre/post conditions are typed with $x\!:\!\mathbb{N}$.

In $\{C\}\,A\ \vdash\ e^{\Gamma;\Upsilon;\alpha}\ :^{\nu\vec{w}:\vec{\alpha}}_{u}\ B\,\{C'\}$, names (including class names) in $\mathsf{dom}(\Gamma)$, $\mathsf{dom}(\Upsilon)$ and u as well as $\vec{w}$ are *primary.* All other names in C, A, B, C' are *auxiliary.* We assume auxiliary names do not contain class names, and that there is a common typing on auxiliary names, say Θ. Then we set:

- Names in C should be typed under $\Upsilon\cdot\Theta$, while those in C' should be typed under $\Upsilon\cdot\Theta\cdot\vec{w}\!:\!\vec{\alpha}$,
- Names in A (including "this") should be typed under Γ,Θ.
- Names in B should be typed under $u\!:\!\alpha,\Theta,\vec{w}\!:\!\vec{\alpha}$.

Similarly for $\{C\}\,A\ \vdash\ P^{\Gamma;\Upsilon}\,\{C'\}$ and for $A\ \vdash\ L^{\Gamma;\mathsf{C}}\ :\ B$. *Hereafter we only consider well-typed sequents.* Under this assumption, we shall often omit type annotations in the sequent, both in programs and in names in formulae. The following convention is used throughout examples, which is convenient while does not let us lose precision. Below an opening in $\mathsf{open}\,(\boldsymbol{\nu}\vec{y})\vec{x}\,\mathsf{in}\,A$ is *trivial* if $\vec{y}$ is the empty string (i.e. $\vec{x}$ hide no local state).

Convention 6. *A trivial opening is usually omitted, i.e. we often simply write* A *for* $\mathsf{open}\,(\boldsymbol{\nu}\varepsilon)\vec{x}\,\mathsf{in}\,A$.

Semantics. We omit detailed discussions on the semantics of assertions in the present logic: as we have done in the preceding sections, we can either give semantics of terms and formulae directly based on operational semantics of the given programming language; or we can use the semantics of the corresponding process logic via encoding. In either case, the semantics of assertions may be considered in terms of partial correctness, total correctness, or a generalisation of both, as discussed in §5. We can give a proof system for each one of them, of which we shall consider a system for partial correctness in the present section. Thus, if e diverges, the formula $\{C\}\,A \vdash e :_u B\,\{C'\}$ always becomes valid; similarly for $\{C\}\,A \vdash P\,\{C'\}$. Invocation $\{C\}\langle\!\langle e_1 \bullet \mathtt{l}(\vec{e}_2)/z\rangle\!\rangle A\{C'\}$ is interpreted accordingly. Since the partial invocation talks about liveness if it is used contravariantly (for example if one negates it), it becomes necessary to preclude contravariant occurrences of invocations on primary names from partial correctness formulae. This well-formedness is called *admissibility*, on which detailed discussions are given in §5.3. Naturally defined formulae which do not talk about termination satisfy admissibility, including all the formulae discussed in the present section.

12.4 A Program Logic for Simple Objects: Proof Rules

We now present the proof rules for partial correctness for our simple object-oriented language. The rules closely follow the standard proof rules for procedural languages, except for refinement needed to treat expressions involving object invocation and generation, as well as local state in objects. We directly treat expressions with side effects in the proof rules: having such expressions is a standard practice in programming, and if we regard proof rules as a tool for analysing behaviours of programs as they are given, it would be worthwhile to try to treat them as a legitimate target of logical analysis. In doing so we take care that this does not lead to proof rules with too much dependency on implementation details (cf. [45]), namely order of evaluation.

We start from proof rules for programs, summarised in Figure 50. All rules are standard except refinement to treat arbitrary expressions in the present language. In each rule, we assume all processes and sequents (including those in conclusions) are well-typed. (skip) is standard. In (assign), the rule models the situation where e is evaluated first, which may result in not only a value but also side effects. The use of existential in these cases goes back to [50]. If the assigned expression contains neither object invocation nor new object generation (which we call *simple*), we can use the standard rule for assignment:

$$\text{(assign-simple)}\ \frac{e \text{ simple}}{\{C[e/x]\}\,A \vdash x := e\ \ \{C\}}$$

which gives much higher-level of abstraction than (assign). The (assign) rule however is more general (which subsumes the simpler version above) and can be used for general cases. The use of existential quantifier has subtlety on which we shall discuss when we apply the rule to reasoning examples.

$$\text{(skip)}\ \frac{-}{\{C\}A \vdash \texttt{skip}\{C\}} \qquad \text{(assign)}\ \frac{\{C\}A \vdash e :_v B\{D^{\text{-}x}\} \quad B[x/v] \wedge D \supset C'}{\{C\}\, A \vdash x := e\ \ \{C'\}} \qquad \text{(seq)}\ \frac{\{C\}\, A \vdash P_1 \{E\} \quad \{E\}\, A \vdash P_2 \{C'\}}{\{C\}\, A \vdash P_1; P_2 \{C'\}}$$

$$\text{(if)}\ \frac{\{C\}\, A \vdash e :_v B\, \{E\} \quad \{E \wedge B[\mathsf{T}/v]\}\, A \vdash P\, \{C'\} \quad \{E \wedge B[\mathsf{F}/v]\}\, A \vdash Q\, \{C'\}}{\{C\}\, A \vdash \texttt{if}\, e\, \texttt{then}\, P\, \texttt{else}\, Q\, \{C'\}} \qquad \text{(while)}\ \frac{\{C\}\, A \vdash e :_v B\, \{C'\} \quad \{C' \wedge B[\mathsf{T}/v]\}\, A \vdash P\, \{C\}}{\{C\}A \vdash \texttt{while}\ e\ \texttt{do}\ P\, \{C' \wedge B[\mathsf{F}/v]\}}$$

$$\text{(var-dec)}\ \frac{\{C\}\, A \vdash P\, \{C'^{\text{-}x}\}}{\{C[\texttt{init}(\alpha)/x]\}\, A \vdash \mathsf{var}\, \alpha\, x\ \mathsf{in}\ P\, \{C'\}} \qquad \text{(class dec)}\ \ (\wedge_i B_i \supset B)\ \frac{A^{\text{-}\vec{\mathsf{C}}} \wedge B \vdash L_i^{\mathsf{C}_i} : B_i \quad \{C\}\, A \wedge B \vdash P\, \{C'\}}{\{C\}\, A \vdash \{L_i\}\ \mathsf{in}\ P\, \{C'\}}$$

Fig. 48. Proof Rules for Simple Objects (program)

(seq) is standard. (if) treats an boolean expression in its guard which may contain, among others, method invocation and object generation. The rule assumes auxiliary names in formulae in the antecedent never collide with primary names in other formulae: the same applies to other rules. Again we may as well use the following rule when we can use a simple expression as a guard.

$$\text{(if-simple)}\ \frac{\{E \wedge e\}\, A \vdash P\, \{C'\} \quad \{E \wedge \neg e\}\, A \vdash Q\, \{C'\} \quad e \text{ simple}}{\{C\}\, A \vdash \texttt{if}\, e\, \texttt{then}\, P\, \texttt{else}\, Q\, \{C'\}}$$

Similarly (while) has its standard variant:

$$\text{(while-simple)}\ \frac{\{C \wedge e\}\, A \vdash P\, \{C\} \quad e \text{ simple}}{\{C\}A \vdash \texttt{while}\ e\ \texttt{do}\ P\, \{C \wedge \neg e\}}$$

(var-dec) is the standard rule for variable declaration. $\texttt{init}(\alpha)$ is a default initial value for a type α. (class-dec) is for class declaration, which says that if:

1. The class definitions ensure properties whose conjunction, $\wedge_i B_i$, is strong enough to ensure what they assume as their own properties, B; and
2. Under the same assumption P satisfies some specification;

then the result of combining P with these class definitions still satisfies the same specification as P even after erasing the assumption B. In the rule $A^{\text{-}\vec{\mathsf{C}}}$ says none

$$\frac{\begin{array}{l}\{C_i^{\Upsilon_i} \wedge I^{\Upsilon_0}\}\, A \vdash e_i :^{\nu \vec{y_i}}_{u_i} B_i\, \{{C'}_i^{\Upsilon_i} \wedge I^{\Upsilon_0}\} \quad (1 \leq i \leq n) \\ \mathsf{dom}(\Upsilon_i) \cap \mathsf{dom}(\Upsilon_j) = \emptyset \quad (0 \leq i, j \leq n,\ i \neq j)\end{array}}{\{\wedge_i C_i \wedge I\}\, A \vdash e_1..e_n :^{\nu \vec{y_1}..\vec{y_n}}_{u_1..u_n} \wedge_i B_i\, \{\wedge_i C'_i \wedge I\}}\ (\textsf{eval-eoi})$$

Fig. 49. A Rule for Evaluation-Order Independence

of C_i occurs in A, while $L_i^{\mathsf{C}_i}$ says L_i is about C_i. Its variation for total correctness is similarly defined.

We next move to the proof rules for expressions. We use the following auxiliary predicate:

$$\{C\}\, A \vdash e_1..e_n :^{\nu \vec{w}}_{u_1..u_n} B\{C'\} \quad (n \geq 0), \tag{12.15}$$

which intuitively says:

> *In whichever order $e_1, \ldots, e_n$ are evaluated, if we start from C under A, the results (if any) together satisfy (named $u_1..u_n$ respectively) B with new local states $\vec{w}$, and the state would change to C'.*

If $n = 0$ then $\{C\}A \vdash \vec{e} :^{\nu \vec{w}}_{\vec{u}} B\{C'\}$ is understood to denote $C \equiv C'$ (the logical equivalence of C and C'). Since expressions can have side effects including object generation, the property of an expression in general depends on the order of evaluations of sub-expressions, which is not desirable. By using this judgement, we obtain assertions which involve evaluation of multiple expressions and which are still independent from possible variations in evaluation orders. The judgement $\{C\}\, A \vdash \vec{e} :^{\nu \vec{w}}_{\vec{u}} B\{C'\}$ itself is inferred from the assertions for individual expressions, by the rule in Figure 49. In the rule, C^{Υ} indicates the primary names (in a given sequent) in C are from Υ, and we assume $\vec{w}_{i,j}$ are disjoint if $i \neq j$. The rule reads:

> *Under the assumption of A, if each e_i converges (if ever) to satisfy B_i, named u_i, and each such evaluation turns C_i to C'_i, both typed under Υ_i, while leaving I invariant, typed under Υ_0; and, moreover, all these sets of involved primary names are mutually disjoint, then we can start from the conjunction of each C_i as well as I to reach the conjunction of C'_i as well as I, while ensuring, under the same assumption A, that the resulting values (if any) satisfy the conjunction of B_i.*

The disjointness $\mathsf{dom}(\Upsilon_i) \cap \mathsf{dom}(\Upsilon_j) = \emptyset$ is related with the separation principle put forward in [95, 104]. Notice however our assumption is much weaker (hence the reasoning principle is more general) since it only demands the exposed free names to be disjoint, not the disjointness of the whole “memory regions”. But it is as strong for deriving various properties. For further discussions, see Remark 11.7, Page 182.

$$\frac{(\mathsf{var})\quad (C \supset B[x/u]) \qquad u \text{ open in } B}{\{C\}\, A \vdash x :_u B\, \{C\}}
\qquad
\frac{(\mathsf{num})\quad (A \supset B[\mathsf{n}/u]) \qquad -}{\{C\}\, A \vdash \mathsf{n} :_u B\, \{C\}}
\qquad
\frac{(\mathsf{this}) \qquad -}{\{C\}\, A[\mathsf{this}/u] \vdash \mathsf{this} :_u A\, \{C'\}}$$

$$\frac{(\mathsf{new})\quad (A \wedge E \supset \langle\!\langle \mathsf{C} \bullet \mathsf{new}(\vec{v})/(\nu\vec{z})u\rangle\!\rangle B) \qquad \{C\}\, A \vdash \vec{e} :_{\vec{v}}^{\nu\vec{y}} E\, \{C'\}}{\{C\}\, A \vdash \mathsf{new}\ \mathsf{C}(\vec{e}) :_u^{\nu\vec{y}\vec{z}} B\, \{C'\}}$$

$$\frac{(\mathsf{invoke})\quad (\, E \supset \{D\}\langle\!\langle x \bullet \mathtt{l}(\vec{y})/(\nu\vec{z})u\rangle\!\rangle\, B\, \{C'\}\,) \qquad \{C\}\, A \vdash e\vec{e'} :_{x\vec{y}}^{\nu\vec{w}} E\, \{D\}}{\{C\}\, A \vdash e.\mathtt{l}(\vec{e'}) :_u^{\nu\vec{w}\vec{z}} B\, \{C'\}}$$

Fig. 50. Proof Rules for Simple Objects (expression)

The rules for expressions are given in Figure 50. All rules use the judgement $\{C\}A \vdash e :_u^{\nu\vec{w}} B\{C'\}$. (var) says x named as u has a property B if we assume for x in C a stronger property. The condition that u should be open (which comes from the process logic) intuitively says that x cannot have a state local to the environment. (num) is about identity of its name, u, and n itself.

The proof rule for object generation, (new), says that: *if*

1. A specifies, among others, what kind of an object C would generate, together with fresh local state;
2. Under A, the evaluation of expressions $\vec{e}$ changes state C to state C' and resulting values (if any), named $\vec{v}$, satisfy E; and
3. A (for the class) and E (for the arguments) together imply that a generation of an instance of C with the arguments $\vec{v}$ results in u which satisfies B,

then creating a new instance of C with arguments $\vec{e}$ would generate an object (named u) satisfying B and would change the state from C to C'.

The proof rule for object invocation, (invoke), is read as:

1. The evaluation of e (an object to be invoked) and $\vec{e'}$ (arguments for the invocation) under the condition A results in: (1) fresh local state $\vec{y}$, (2) change of state from C to D, and (3) objects $o\vec{v}$ of a property E; and
2. E entails the result of invoking o at the method l with arguments $\vec{v}$ results, if ever, (1) fresh local state $\vec{w}$; (2) state change from D to C'; and (3) an object (named u) for which B holds.

then $e.\mathtt{l}(\vec{e'})$ named as u at C would results in fresh local state $\vec{y}\vec{z}$, an object satisfying B, as well as a state change to C'.

(return)

$$\frac{\{C\}A \vdash e :_v^{\nu\vec{y}} B\{C'\}}{\{C\}A \vdash \mathsf{return}\ e :_v^{\nu\vec{y}} B\{C'\}}$$

(seq-return)

$$\frac{\{C\}A \vdash P_1\{E\} \quad \{E\}A \vdash P_2 :_v^{\nu\vec{y}} B\{C'\}}{\{C\}A \vdash P_1; P_2 :_v^{\nu\vec{y}} B\{C'\}}$$

(if-return)

$$\frac{\{C\}A \vdash e :_v^{\nu\vec{y}} B\{E\} \quad \{E \wedge B[\mathsf{T}/v]\}A \vdash P :_v^{\nu\vec{z}} B\{C'\} \quad \{E \wedge B[\mathsf{F}/v]\}A \vdash Q :_v^{\nu\vec{z}} B\{C'\}}{\{C\}A \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q :_v^{\nu\vec{y}\vec{z}} B\{C'\}}$$

(class body) $(\bigwedge_i \forall \vec{x}_i.(E_i \wedge B_i \supset \{C_i\}[u \bullet \mathtt{l_i}(\vec{x}_i)/(\nu\vec{w}_i)m_i]C'_i\{B_i\}) \supset B)$

$$\frac{\forall i.\ \{C_i^{\text{-}\vec{x}_i} \wedge E_i^{\vec{x}_i}\}A \wedge B[\mathsf{this}/u] \vdash P_i :_{m_i}^{\nu\vec{w}_i} B_i\{C_i'^{\text{-}\vec{x}_i}\}}{A \vdash \{\gamma_i\ \mathtt{l}_i(\beta_i\ x_i)P_i\} :_u B}$$

(class def) $(\forall \vec{y}.(\langle\langle \mathsf{C} \bullet \mathsf{new}(\vec{y})/(\nu\vec{w})u\rangle\rangle E) \supset B)$

$$\frac{A \vdash \{\gamma_i\ \mathtt{l}_i(\beta_i\ x)P_i\} :_u E}{A \vdash\ \mathsf{class}\,\mathsf{C}(\alpha\,\vec{y})\{\gamma_i\ \mathtt{l}_i(\beta_i\ \vec{x}_i)P_i\} : B}$$

Fig. 51. Proof Rules for Simple Objetcs (return command and class)

To infer properties of classes, we first treat method bodies, given by the first three rules in Figure 51. Except (return), they precisely follow Figure 48 (we can also add the return occurring in the body of while).

We can then use the last two rules of Figure 51 for inferring about classes. (class-body) is for (the code part of) an object, while the second rule (class-dec) is for a class. Recall that the process encoding of a class is given by two nested replications: the first rule corresponds to the inner replication (with a treatment of a recursion), while the second one for the outer replication. (class-body) is directly based on the structural correspondence between (1) the antecedent, which collects assertions for method bodies for all methods in a class; and (2) the side condition which says that the assumption for "this" (which also becomes the guarantee condition in the conclusion) should be weaker than the conjunction of the properties of method bodies.

In (class-def), an assertion E is for the code of the class, which would mention instance variables $\vec{y}$ of the class, which, in the present simplified language, coincide with the parameters to the new command. The side condition simply regards the assertion for the code of the class, B, as the result of a "new" invocation with $\vec{y}$ being its arguments. This rule does *not* treat a circular usage of properties of classes, which is done in (class-dec) (cf. Page 202). The rule is easily refined to the case which involves a more non-trivial initialisation is done in object creation, or when general static methods are involved.

$$\frac{\{C\boldsymbol{\sigma}\}\,A \vdash P\,\{C'\boldsymbol{\sigma}\}}{\{C\}\,A \vdash P\,\{C'\}}\ (\text{co-hide-prg})\quad (\boldsymbol{\sigma} = [\boldsymbol{\nu}\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\vec{y}_i x\vec{z}_i]_i)$$

$$\frac{\{C\boldsymbol{\sigma}\}\,A \vdash e :_v^{\nu\vec{w}}\ \{C'\boldsymbol{\sigma}\}}{\{C\}\,A \vdash e :_v^{\nu\vec{w}}\ \{C'\}}\ (\text{co-hide-exp})\quad (\boldsymbol{\sigma} = [\boldsymbol{\nu}\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\vec{y}_i x\vec{z}_i]_i)$$

$$\frac{\{C\}\ A\ \vdash e :_v^{\nu\vec{u}\vec{w}}\ (B^{\text{-}\vec{w}} \wedge G^{\vec{w}})\,\{C_0'\}}{\{C\}\ A\ \vdash e :_v^{\nu\vec{u}}\ B\,\{C'\}}\ (\text{shift-right})\quad (\,G \wedge C_0' \supset\ C'[\boldsymbol{\nu}\,\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\,\vec{y}_i\vec{w}\vec{z}_i]_i\,)$$

$$\frac{\{C\boldsymbol{\sigma}\}\ A\ \vdash e :_v^{\nu\vec{u}}\ B\,\{C'^{\text{-}\vec{w}} \wedge E^{\vec{w}}\}}{\{C\}\ A\ \vdash e :_v^{\nu\vec{u}\vec{w}}\ B \wedge E\,\{C'\}}\ (\text{shift-left})\quad (\boldsymbol{\sigma} = [\boldsymbol{\nu}\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\vec{y}_i\vec{w}\vec{z}_i]_i)$$

$$\frac{\{C_0\}\,A_0 \vdash e :_u^{\nu\vec{z}}\ \{C_0'\} \quad \vec{z} \overset{\mathsf{prm}}{\equiv} \vec{y}}{\{C\}\,A \vdash e :_u^{\nu\vec{y}}\ \{C'\}}\ (\text{consequence})\quad (A \wedge C \supset A_0 \wedge C_0,\ B_0 \wedge C_0' \supset C')$$

Fig. 52. Proof Rules for Simple Objects (structure rules)

Finally we list the main structural rules in Figure 52. We use the following notation (cf. §11.6, Page 178).

$$A[\boldsymbol{\nu}\,\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\,\vec{y}_i\vec{x}\vec{z}_i]_i,$$

The notation indicates multiple substitutions, replacing each occurrence of a binder of shape $\boldsymbol{\nu}\,\vec{y}_i\vec{x}\vec{z}_i$ with $\boldsymbol{\nu}\,\vec{y}_i\vec{z}_i$. As an example,

$$(\mathsf{open}\,(\boldsymbol{\nu} wx)u \,\mathsf{in}\, A \wedge \mathsf{open}\,(\boldsymbol{\nu} wxw')uv \,\mathsf{in}\, B)[\boldsymbol{\nu}\,\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\,\vec{y}_i x\vec{z}_i]_i$$

yields the following formula, assuming no binding by x occurs in A.

$$\mathsf{open}\,(\boldsymbol{\nu} w)u \,\mathsf{in}\, A \wedge \mathsf{open}\,(\boldsymbol{\nu} ww')uv \,\mathsf{in}\, B.$$

There are three hiding rules.

1. (co-hide-prg) and (co-hide-exp) add a hiding in the pre/post conditions, thus allowing us to transform assertions derived from the reasoning based on unhidden variables to those on local state.
2. (shift-right) internalises the hiding into the post condition by moving the local state from the guarantee formula, while (shift-left) is its precise dual, which together transfer the hiding in assertions and the hiding in opening formulae.

For consequence rules, we have one consequence rule for each form of the judgement. We only list one for expressions above: other consequence rules are its simpler variants. In the rule, $\overset{\mathsf{prm}}{\equiv}$ denotes the equality of non-duplicating strings up to permutation (e.g. $xyz \overset{\mathsf{prm}}{\equiv} yxz$).

(weaken-program)

$$\frac{\{C^{\text{-}\vec{x}}\}\, A \vdash P^{\Upsilon;\Gamma}\, \{C'^{\text{-}\vec{x}}\}}{\{C \wedge E^{\vec{x}}\}\, A \vdash P^{\Upsilon\cdot\vec{x}:\vec{\alpha}\,;\,\Gamma}\, \{C' \wedge E^{\vec{x}}\}}$$

(weaken-exp)

$$\frac{\{C^{\text{-}\vec{x}}\}\, A \vdash e^{\Upsilon;\Gamma;\alpha} :_u^{\nu\vec{w}}\ \{C'^{\text{-}\vec{x}}\}}{\{C \wedge E^{\vec{x}}\}\, A \vdash e^{\Upsilon\cdot\vec{x}:\vec{\alpha};\Gamma;\alpha} :_u^{\nu\vec{w}}\ \{C' \wedge E^{\vec{x}}\}}$$

Fig. 53. Weakening Rules

If we assume validity of these formulae is given by the underlying universe of processes in Section 11 (with translations of formulae and assertions), then the entailment mentioned in (consequence) talks about not only number-theoretic facts but also satisfiability of (translated) formulae by processes. While this in general involves calculating inclusions of inhabitants of each formula using, for example, labelled transition, some aspects are amenable to syntactic reasoning. In the following we list those rules related with opening formulae from the process logic. Below $\star$ ranges over $\{\wedge, \vee, \supset\}$, while $\mathcal{Q}$ ranges over $\{\forall, \exists\}$.

(open-$\boldsymbol{\nu}$-elim-1)	$\mathsf{open}\ (\boldsymbol{\nu}\vec{y})\vec{x}\ \mathsf{in}\ A^{\text{-}\vec{x}\vec{y}}$	$\equiv$	A
(open-$\boldsymbol{\nu}$-elim-2)	$\mathsf{open}\ (\boldsymbol{\nu}\varepsilon)\vec{x}\ \mathsf{in}\ A$	$\equiv$	A
(open-$\boldsymbol{\nu}$-perm-1)	$\mathsf{open}\ (\boldsymbol{\nu}\vec{z}_1 y_1 y_2 \vec{z}_2)\vec{x}\ \mathsf{in}\ A$	$\equiv$	$\mathsf{open}\ (\boldsymbol{\nu}\vec{z}_1 y_2 y_1 \vec{z}_2)\vec{x}\ \mathsf{in}\ A$
(open-$\boldsymbol{\nu}$-perm-2)	$\mathsf{open}\ (\boldsymbol{\nu}\vec{z})\vec{x}_1 y_1 y_2 \vec{x}_2\ \mathsf{in}\ A$	$\equiv$	$\mathsf{open}\ (\boldsymbol{\nu}\vec{z})\vec{x}_1 y_2 y_1 \vec{x}_2\ \mathsf{in}\ A$
(open-$\boldsymbol{\nu}$-dist-1)	$\mathsf{open}\ (\boldsymbol{\nu}\vec{y})\vec{x}\ \mathsf{in}\ (A \star B)$	$\supset$	$(\mathsf{open}\ (\boldsymbol{\nu}\vec{y})\vec{x}\ \mathsf{in}\ A)\star$ $(\mathsf{open}\ (\boldsymbol{\nu}\vec{y})\vec{x}\ \mathsf{in}\ B)$
(open-$\boldsymbol{\nu}$-dist-2)	$\mathsf{open}\ (\boldsymbol{\nu}\vec{y})\vec{x}\ \mathsf{in}\ (\mathcal{Q}z.A)$	$\equiv$	$\mathcal{Q}z.\,(\mathsf{open}\ (\boldsymbol{\nu}\vec{y})\vec{x}\ \mathsf{in}\ A)$
(open-$\boldsymbol{\nu}$-scope)	$(\mathsf{open}\ (\boldsymbol{\nu}\vec{y})\vec{x}\ \mathsf{in}\ A) \star B^{\text{-}\vec{x}\vec{y}}$	$\equiv$	$\mathsf{open}\ (\boldsymbol{\nu}\vec{y})\vec{x}\ \mathsf{in}\ (A \star B)$
(open-$\boldsymbol{\nu}$-vacuous)	$\mathsf{open}\ (\boldsymbol{\nu}\vec{y}_1\vec{y}_2)\vec{x}\ \mathsf{in}\ A^{\text{-}z}$	$\supset$	$\mathsf{open}\ (\boldsymbol{\nu}\vec{y}_1 z \vec{y}_2)\vec{x}\ \mathsf{in}\ A$
(open-$\boldsymbol{\nu}$-concat)	$\mathsf{open}\ (\boldsymbol{\nu}\vec{y})\vec{x}\ \mathsf{in}\ (\mathsf{open}\ (\boldsymbol{\nu}\vec{v})\vec{u}\ \mathsf{in}\ A)$	$\supset$	$\mathsf{open}\ (\boldsymbol{\nu}\vec{y}\vec{v})\vec{x}\vec{u}\ \mathsf{in}\ A$

There are other structural rules which are useful in reasoning. In Figure 53, we list the weakening rules which crucially make use of the type information. The rule uses a weak form of separation, requiring only the surface (primary names) of two predicates are disjoint, which may as well allow sharing of structures at names not mentioned in these predicates. This is still safe because the sharing information is strictly described by opening formulae. For detailed discussions on the relationship between the separation principle by Reynolds, O'Hearn and others [16, 95, 104, 105], on the one hand, and reasoning principles derivable from these weakening rules, see Remark 11.7, Page 182.

12.5 Reasoning Examples

Recall the specification for the class Inc in §12.3, which we reproduce below:

$$A(u,x) \stackrel{\text{def}}{=} \forall n.\ (\ \langle\!\langle \{x=n\}\, u \bullet \mathsf{do}()\, \{x=n+1\}/z \rangle\!\rangle z=n+1\ ) \qquad (12.16)$$

$$B \stackrel{\text{def}}{=} \forall i \in \mathbb{N}.\ \langle\!\langle \mathsf{C} \bullet \mathsf{new}(i)/(\boldsymbol{\nu}\, x)u \rangle\!\rangle\, (A(u,x) \wedge x=i). \qquad (12.17)$$

We first infer B for the class Inc. Below for brevity we let $P \stackrel{\text{def}}{=} x:=x+1; \mathsf{return}\ x$.

1.	$\{x=i\}\, \mathsf{T} \vdash x :_j j=i\, \{x=i\}$	(var)
2.	$\{x=i\}\, \mathsf{T} \vdash 1 :_k k=1\, \{x=i\}$	(num)
3.	$\{x=i\}\, \mathsf{T} \vdash x+1 :_v v=i+1\, \{x=i\}$	(1, 2, add)
4.	$x=i \supset \exists v.(v=i+1 \wedge v=i+1)$	(PRED)
5.	$\{x=i\}\, \mathsf{T} \vdash x := x+1\, \{x=i+1\}$	(3, 4, assign)
6.	$\{x=i+1\}\, \mathsf{T} \vdash x :_v v=i+1\, \{x=i+1\}$	(var)
7.	$\{x=i+1\}\, \mathsf{T} \vdash \mathsf{return}\ x :_v v=i+1\, \{x=i+1\}$	(return)
8.	$\{x=i\}\, \mathsf{T} \vdash x := x+1\ ;\ \mathsf{return}\ x :_v v=i+1\, \{x=i+1\}$	(5,7,seq-ret)
9.	$\{x=i\}\, A[\mathsf{this}/u] \vdash x := x+1\ ;\ \mathsf{return}\ x :_v v=i+1\, \{x=i+1\}$	(8, conseq)
10.	$\mathsf{T} \vdash \{\mathtt{int\ do}()\{P\}\} :_u A(u,x)$	(8, c-body)
11.	$\mathsf{T} \vdash \mathsf{class\ Inc}(\mathtt{int}\ x)\{\mathtt{int\ do}()\{P\}\} : B$	(10, c-def)

Above (add) is the obvious rule for addition. (PRED) means a theorem from the standard first-order predicate calculus. We deliberately applied the original rules in the first eight lines, where we could have used the standard rules for side-effect free commands. In fact, if we use (assign-simple) combined with:

$$(\mathsf{return\text{-}simple})\ \frac{e \text{ simple}}{\{C[e/v]\}A \vdash \mathsf{return}\ e :_v C\{C[e/v]\}}$$

then we can condense the initial eight steps into mere three steps. Another rule which is sometimes convenient is:

$$(\mathsf{assign}_d)\ \frac{\{C^{\text{-}x}\}A \vdash e :_u B\{C'^{\text{-}x}\}}{\{C\}\, A \vdash x := e\, \{C' \wedge B[x/u]\}}$$

which is derivable from (assign) and (consequence). This rule will be used later.

We next infer for an assignment of a new object for the class Inc, showing, with the predicate B given above:

$$\{\mathsf{T}\}\ B \vdash y := \mathsf{new\ Inc}(1)\ \{A'(y,1)\} \qquad (12.18)$$

where $A'(o,i) \stackrel{\text{def}}{=} \mathsf{open}\,(\boldsymbol{\nu} x)o\,\mathsf{in}\,(A(u,x) \wedge x = i)$. The inference follows.

1.	$\{\mathsf{T}\}\, B \vdash 1 :_i i = 1\, \{\mathsf{T}\}$	(num)
2.	$B \wedge i = 1 \supset \langle\!\langle \mathsf{C} \bullet \mathsf{new}(i)/(\boldsymbol{\nu}\, x)u \rangle\!\rangle\, (A(u,x) \wedge x = 1)$	(EQ)
3.	$\{\mathsf{T}\}\, B \vdash \mathsf{new\ Inc}(1) :_u^{\nu x} (\ A(u,x) \wedge x = 1\ )\, \{\mathsf{T}\}$	(1, 2, new)
4.	$\{\mathsf{T}\}\, B \vdash \mathsf{new\ Inc}(1) :_u \mathsf{open}\,(\boldsymbol{\nu} x)u\,\mathsf{in}\,(\ A(u,x) \wedge x = 1\ )\, \{\mathsf{T}\}$	(shift-r)
5.	$\mathsf{T} \supset \exists u.(A'(u,1) \wedge A'(u,1))$	(UNDEF)
6.	$\{\mathsf{T}\}\, B \vdash y := \mathsf{new\ Inc}(1)\, \{A'(y,1)\}$	(assign$_d$)

Above (EQ) indicates a theorem in the predicate calculus with equality; (UNDEF) indicates satisfiability of a formula of the form $\exists x.A$ (with x denoting an object) by assigning the least defined element to x (which immediately diverges at each method invocation). Such a behaviour is always realisable, and an existential formula can be satisfied by such a behaviour in many practical examples (including the above case) as far as we are working with partial correctness. There are other useful techniques, which will be discussed elsewhere.

Next we take an example which demonstrate non-trivial sharing of object references, again from §12.2. Below we reproduce their code[19].

$$R_1 \stackrel{\text{def}}{=} x := \mathsf{new\ Inc}(1)\ ;\ y := \mathsf{new\ List}(x, \mathsf{nil})\ ;\ z := \mathsf{new\ List}(x, y)$$
$$R_2 \stackrel{\text{def}}{=} u := y.\mathtt{getCar}()\,;\ v := z.\mathtt{getCar}()\,;\ w' := u.\mathsf{do}()\,;\ w := v.\mathsf{do}() + w'$$

We wish to show:

$$\{\mathsf{T}\}\ B \wedge G \vdash R_1; R_2\ \{H'\} \tag{12.19}$$

where B is the assertion for Inc we have seen above, while G is the assertion for the class List which is given as follows, with $C(p',l') \stackrel{\text{def}}{=} (p = p' \ \wedge\ l = l')$:

$$\begin{aligned}
G &\stackrel{\text{def}}{=} \forall pl.\,(\,\langle\!\langle \mathsf{List} \bullet \mathsf{new}(p'l')/(\boldsymbol{\nu}\, pl)o \rangle\!\rangle\,(\,E(o,p,l) \wedge C(p',l')\,)\\
E_1(o,p,l) &\stackrel{\text{def}}{=} \forall p', l'.\ (\{C(p',l')\}\langle\!\langle o \bullet \mathtt{getCar}()/z \rangle\!\rangle(z = p')\{C(p',l')\})\\
E_2(o,p,l) &\stackrel{\text{def}}{=} \forall p', l'.\ (\,\{C(p',l')\}\langle\!\langle o \bullet \mathtt{setCar}(p'')/z \rangle\!\rangle(z = p')\{C(p'',l')\}\,)\\
E_3(o,p,l) &\stackrel{\text{def}}{=} \forall p', l'.\ (\{C(p',l')\}\langle\!\langle o \bullet \mathtt{getCdr}()/z \rangle\!\rangle(z = l')\{C(p',l')\})\\
E_4(o,p,l) &\stackrel{\text{def}}{=} \forall p', l'.\ (\,\{C(p',l')\}\langle\!\langle o \bullet \mathtt{setCdr}(l'')/z \rangle\!\rangle(z = p')\{C(p',l'')\}\,)\\
E(o,p,l) &\stackrel{\text{def}}{=} E_1 \wedge E_2 \wedge E_3 \wedge E_4
\end{aligned}$$

and H' is given by, using the predicate $A(x,r)$ treated above:

$$\begin{aligned}
H' \stackrel{\text{def}}{=}\ & \mathsf{open}\,(\boldsymbol{\nu}\, lpl'p'r)\, xyzuv\ \mathsf{in}\ (\ E(y,p,l)\ \wedge\ E(z,p',l')\ \wedge\ E(l',p,l)\ \wedge\\
& A(x,r) \wedge A(p,r) \wedge A(p',r) \wedge A(u,r) \wedge A(v,r) \wedge l = \mathsf{nil} \wedge r = 3\,)\ \wedge\ w = 5\,.
\end{aligned}$$

[19] R_2 is altered slightly so that it is immediately evaluation-order independent: the original code is in fact semantically evaluation-order independent, but its shape does not conform to (eval-eoi).

The derivation of the assertion in (12.19) is done by inferring the following two assertions.

$$\{\mathsf{T}\}\, G \vdash R_1\, \{H\} \tag{12.20}$$

$$\{H\}\, G \vdash R_2\, \{H'\} \tag{12.21}$$

were we set:

$$H \quad \stackrel{\text{def}}{=} \quad \mathsf{open}\,(\boldsymbol{\nu}\, lpl'p'r)\, xyz \;\mathsf{in}\; (\, E(y,p,l) \wedge E(z,p',l') \wedge E(l',p,l) \wedge A(x,r) \wedge A(p,r) \wedge A(p',r) \wedge l = \mathsf{nil} \wedge r = 1\,)$$

For inferences, it is convenient to use the following derived rule.

$$(\mathsf{seq}\text{-}\nu)\ \frac{\{C\}\ A\ \vdash P_1\ \{E\} \quad \{E\boldsymbol{\sigma}\}\ A\ \vdash P_2\ \{C'\boldsymbol{\sigma}\} \quad (\boldsymbol{\sigma} = [\boldsymbol{\nu}\vec{y}_i\vec{z}_i/\boldsymbol{\nu}\vec{y}_i\vec{x}\vec{z}_i]_i)}{\{C\}\ A\ \vdash P_1; P_2\ \{C'\}}$$

By applying (co-hide-prg) to the second sequent in the antecedent, the rule is easily admissible. For inferring for (12.20), we also use the following abbreviations.

$$J_1(o,p,l,r) \stackrel{\text{def}}{=} (E(o,p,l) \wedge A(p,r) \wedge l = \mathsf{nil})$$

$$J_2(o,p',l',p,l,r) \stackrel{\text{def}}{=} (E(o,p',l') \wedge A(p',r) \wedge E(l',p,l)).$$

The inference follows. We freely use the consequence rule for programs, in particular scope extension/concatenation of opening formulae (cf. Page 207).

1. $\{\mathsf{T}\}\ B \wedge G \vdash x := \mathsf{new\ Inc}(1)\ \{A'(x,1)\}$ (as before)

---

2. $B \wedge G \wedge A(v,r) \wedge w = \mathsf{nil}\ \supset\ \langle\!\langle \mathsf{List} \bullet \mathsf{new}(vw)/(\boldsymbol{\nu}\, pl)o \rangle\!\rangle\ J_1(o,p,l.r)$ (PRED)

---

3. $\{A(x,r) \wedge r = 1\}\ B \wedge G \vdash \mathsf{new\ List}(x,\mathsf{nil}) :_u$ $\mathsf{open}\ (\boldsymbol{\nu}\, pl)u\ \mathsf{in}\ J(u,x,\mathsf{nil})\ \{A(x,r) \wedge r = 1\}$ (new, shift-r)

---

4. $\{A(x,r) \wedge r = 1\}\ B \wedge G \vdash\ y := \mathsf{new\ List}(x,\mathsf{nil})$ $\{\mathsf{open}\ (\boldsymbol{\nu}\, pl)xy\ \mathsf{in}\ (A(x,r) \wedge J_1(y,p,l,\mathsf{nil}) \wedge r = 1)\}$ (assign$_d$)

---

5. $\{A(x,r) \wedge J_1(y,p,l,\mathsf{nil}) \wedge r = 1\}\ B \wedge G \vdash \mathsf{new\ List}(x,y) :_u$ $\mathsf{open}\ (\boldsymbol{\nu}\, p'l')u\ \mathsf{in}\ J_2(u,p',l',p,l,r)$ (new, shift-r)

---

6. $\{A(x,r) \wedge J_1(y,p,l,\mathsf{nil}) \wedge r = 1\}\ B \wedge G\ \vdash\ z := \mathsf{new\ List}(x,y)$ $\{\mathsf{open}\ (\boldsymbol{\nu}\, p'l')xyz\ \mathsf{in}\ (A(x,r) \wedge J_1(y,p,l,\mathsf{nil}) \wedge p'r'\, J_2(z,p',r',p,l,r) \wedge r = 1)\}$ (assign$_d$)

---

7. $\{A(x,r) \wedge r = 1\}\ B \wedge G\ \vdash\ y := \mathsf{new\ List}(x,\mathsf{nil}); z := \mathsf{new\ List}(x,y)$ $\{\mathsf{open}\ (\boldsymbol{\nu}\, plp'l')xyz\ \mathsf{in}\ (A(x,r) \wedge J_1(y,p,l,\mathsf{nil}) \wedge p'r'\, J_2(z,p',r',p,l,r) \wedge r = 1)\}$ (5,7,seq-ν)

---

8. $\{\mathsf{T}\}\, B \wedge G\ \vdash\ x := \mathsf{new\ Inc}(1)\,;\, y := \mathsf{new\ List}(x,\mathsf{nil})\,;\, z := \mathsf{new\ List}(x,y)$ $\{\mathsf{open}\ (\boldsymbol{\nu}\, plp'l'r)xyz\ \mathsf{in}\ (A(x,r) \wedge J_1(y,p,l,r) \wedge p'r'\, J_2(z,p',l',p,l,r) \wedge r = 1)\}$ (1,8,seq-ν)

hence as required. Note how the sharing information is accumulated by a series of sequential compositions.

We now turn to (12.20). As we mentioned in §12.2, we use intermediate assertions which are about open names.

$$
\begin{aligned}
H_0 &\stackrel{\text{def}}{=} A(x,r) \ \wedge\ J_1(y,p,l,r) \ \wedge\ J_2(z,p',l',p,l,r) \ \wedge\ r=1 \\
I &\stackrel{\text{def}}{=} A(x,r) \ \wedge\ J_1(y,p,l,r) \ \wedge\ J_2(z,p',l',p,l,r) \ \wedge\ r=2 \ \wedge\ A(u,r) \ \wedge\ A(v,r) \\
I' &\stackrel{\text{def}}{=} A(x,r) \ \wedge\ J_1(y,p,l,r) \ \wedge\ J_2(z,p',l',p,l,r) \ \wedge\ r=3 \ \wedge\ A(u,r) \ \wedge\ A(v,r) \\
H_0' &\stackrel{\text{def}}{=} I' \ \wedge\ w=5
\end{aligned}
$$

In the following inference for the assertion (12.20), we use the following entailment (discussed in Proposition 6, §9.6, Page 135).

$$
\text{(weaken-p)}\ (\{C\}[a \bullet \vec{b}/z]A\{C'\} \wedge (E \supset E_0)) \ \supset\ \{C \wedge E\}[a \bullet \vec{b}/z](A \wedge E_0)\{C' \wedge E\}
$$

where, respectively, e and $\vec{e'}$ are typed by $\vec{\alpha} \stackrel{\Upsilon}{\Rightarrow} \beta$ and $\vec{\alpha}$, both C and C' are typed under[20] $\Upsilon \cdot \Gamma$, while E and E_0 are typed under $\Upsilon' \cdot \Gamma$ and Γ respectively, where $\mathsf{dom}(\Upsilon') \cap \mathsf{dom}(\Upsilon) = \emptyset$. The inference follows. Below we remember

$$
E_1(o,p,l) \stackrel{\text{def}}{=} \forall p', l'.\ (\{C(p',l')\}\langle\!\langle o \bullet \mathtt{getCar}()/z \rangle\!\rangle(z=p')\{C(p',l')\}
$$

[20] Here we use distinction between program variables (which can be ν-bound) and logical ones: e.g. $uvxyzrlp$ are program variables while others are logical.

where $C(p',l') \stackrel{\text{def}}{=} p = p' \wedge l = l'$,

1.	$\{H_0\}\,\mathsf{T} \vdash y :_u E_1(u,p,l)\{H_0\}$	(var)
2.	$E_1(u,p,l) \supset \{H_0\}\langle\!\langle u \bullet \mathtt{getCar}()/z\rangle\!\rangle A(z,r)\,\{H_0\}$	(weaken-p, EQ)
3.	$\{H_0\}\,\mathsf{T} \vdash y.\mathtt{getCar}() :_f A(f,r)\,\{H_0\}$	(2,invoke)
4.	$\{H_0\}\,\mathsf{T} \vdash u := y.\mathtt{getCar}()\,\{H_0 \wedge A(u,r)\}$	(assign$_d$)
5.	$\{H_0 \wedge A(u,r)\}\,\mathsf{T} \vdash z.\mathtt{getCar}() :_g A(g,r)\,\{H_0 \wedge A(u,r)\}$	(as above)
6.	$\{H_0 \wedge A(u,r)\}\mathsf{T} \vdash v := z.\mathtt{getCar}()\,\{H_0 \wedge A(u,r) \wedge A(v,r)\}$	(assign$_d$)
7.	$\{H_0 \wedge A(u,r) \wedge A(v,r)\}\,\mathsf{T} \vdash u.\mathsf{do}() :_a a=2\{I\}$	(var, invoke)
8.	$\{H_0 \wedge A(u,r) \wedge A(v,r)\}\,\mathsf{T} \vdash w' := u.\mathsf{do}()\,\{I \wedge w'=2\}$	(assign$_d$)
9.	$\{I\}\mathsf{T} \vdash v.\mathsf{do}() :_b b=3\{I'\}$	(var, invoke)
10.	$\{w'=2\}\mathsf{T} \vdash w :_c c=3\{w'=2\}$	(var, invoke)
11.	$\{I \wedge w'=2\}v.\mathsf{do}(),\ w' :_{bc} (b=3 \wedge c=3)\{I' \wedge w'=2\}$	(9, 10, eval-eoi)
12.	$\{I \wedge w'=2\}v.\mathsf{do}() + w' :_h h=5\{I' \wedge w'=2\}$	(11, add)
13.	$\{I' \wedge w'=2\}\,\mathsf{T} \vdash w := (v.\mathsf{do}()+w')\,\{I' \wedge w=5\}$	(assign$_d$)
14.	$\{H_0\}\mathsf{T} \vdash R_2\,\{H_0'\}$	(4,6,Seq,8,Seq,12,Seq)
15.	$\{H\}\mathsf{T} \vdash R_2\,\{H'\}$	(co-hide-prg)

We can now complete our inference by combining the conclusions of the preceding two sub-inferences, which we call $(*)$ and $(**)$.

1.	$\{\mathsf{T}\}\,B \wedge G \vdash R_1\,\{H\}$	$(*)$
2.	$\{H\}\,B \wedge G \vdash R_2\,\{H'\}$	$(**)$
3.	$\{\mathsf{T}\}\,B \wedge G \vdash R_1; R_2\,\{H'\}$	(seq)

as required.

Leaving other examples to later occasions, we shall conclude our present exploration with a simple example which illustrates the need of a binding in assertions for expressions and method bodies. Assume given a code fragment:

$$P_0 \stackrel{\text{def}}{=} x := \mathsf{new\ Inc}(1)\,;\ \mathsf{return}\ x \tag{12.22}$$

where we assume x is an instance variable. For P_0, we can infer:

1. $\{\mathsf{T}\}\, B \vdash x := \mathsf{new\ Inc}(1)\, \{A'(x,1)\}$	$(\mathsf{new},\ \mathsf{assign}_d)$
2. $\{A(x,r) \wedge r\!=\!1\}B \vdash x\ :_u A(u,r)\{A(x,r) \wedge r\!=\!1\}$	(var)
3. $\{A(x,r) \wedge r\!=\!1\}B \vdash \mathsf{return}\ x\ :_u A(u,r)\{A(x,r) \wedge r\!=\!1\}$	(return)
4. $\{A'(x,1)\}B \vdash \mathsf{return}\ x\ :_u^{\nu r}\ (A(u,r) \wedge r\!=\!1)\{A(x,r)\}$	$(\mathsf{shift\text{-}I})$
5. $\{\mathsf{T}\}B \vdash x := \mathsf{new\ Inc}(1)\, ;\, \mathsf{return}\ x\ :_u^{\nu r}\ (A(u,r) \wedge r\!=\!1)\{A(x,r)\}$	$(\mathsf{seq\text{-}return})$

In the conclusion in Line 5, r occurs in both the guarantee condition and the post-condition, while it is freshly generated: thus it is necessary to have a binding which goes across two conditions. One may also notice the above inference (as well as preceding ones) becomes simpler if we allow a similar binder for programs, in the shape of, for example, $\{C\}A \vdash P\,(\boldsymbol{\nu}\,\vec{w})\{C'\}$, which in fact follows more closely to the process logic in Section 11. We have chosen the present form to have the shape of assertions closer to the known program logics (at least for programs). It is worth exploring whether the additional notation for programs adds non-trivial expressivenes (or convenience) in our logic for objects, both formally and in its practical usage.

Our investigation into logics for object-oriented languages should stop here, at least for the time being. For extending the presented program logic to a fully-fledged object-oriented language in use, such as Java [31, 65] and C♯ [89], several significant additional language features should be considered, including, among others: *inheritance*, *exceptions*, *interface*, *concurrency primitives*, and *reflections*. Logical articulation of some of these ideas — for example inheritance and exceptions – may better be studied starting from purely functional setting, cf. [63], exploiting the stratification of logical theories in the present framework. We believe incorporation of these extensions into our framework is a feasible endeavour, directly built on the basis of the present development, even though some of them do seem to pose significant technical challenges, presenting fascinating future research topics.

⋆ ⋆ ⋆ ⋆ ⋆

At this point we tentatively close our investigation into logics for sequential processes and their applications to program logics. We hope our investigation in the present note, a modest trial in a limited realm to pursue the directions and insights presented in, among others, [49, 82], based on the fundamental contributions by the scientific community in semantics and logics of computation, makes it clear that, philosophically, the computational structures induced by diverse programming languages do have a basic order which can be cleanly articulated by a logical means; and that, scientifically, it is indeed possible to have articulation of programs' behaviour which closely suits individual programming languages while being based on a unifying mathematical theory of interacting processes.

As discussed in Introduction, and beyond the topics mentioned in that section, there are many theoretical and practical challenges which are awaiting to be tackled, if the framework introduced in this note is ever to develop into a truly general scientific and engineering framework. It would be a pleasure beyond our expectations if the endeavour to meet these challenges can be shared with any of the readers of the present note, which, combined with other activities, would in the end lead to a general understanding of the universe of computation with clarity and precision which only science can give us.

End, March 15, 2004.
Revised, April 11, 2004.
Revised, May 2, 2004.

References

1. Abadi, M., $\top\top$-closed relations and admissibility, MSCS 10(3) (June 2000), pp. 313-320.
2. Abadi, M., Cardelli, L., Curien, P.-L., Formal Parametric Polymorphism, *TCS* 121, 1-2, (Dec), pp. 9-58. Elsevier, 1993.
3. Abadi, M, Leino, R. A logic for object-oriented programs. Technical Report SRC-161, Compaq SRC, 1998.
4. Abramsky, S., Domain Theory in Logical Form, Annuals of Pure and Applied Logic, 51:1-77, 1991.
5. Abramsky, S., Honda, K. and McCusker, G., Fully Abstract Game Semantics for General References, *LICS'98*, 334–344, IEEE, 1998.
6. Abramsky, S., Jagadeesan, R. and Malacaria, P., Full Abstraction for PCF, 1994. *Info. & Comp.* 163 (2000), 409-470.
7. Agha, G., Actors: A Model of Concurrent Computing in Distributed Systems, MIT Press, 1986.
8. Apt, K.R. Ten Years of Hoare Logic: a survey. *TOPLAS*, Vol. 3:431-483, 1981.
9. Barbanera, F., Dezani-Ciancaglini, M. and de'Liguoro, U., Intersection and Union Types: syntax and semantics. *Inf. and Comp.*, 119:202–230, 1995.
10. Boudol, G., Asynchrony and the pi-calculus, INRIA Research Report 1702, 1992.
11. Broda, K., et. al., Reasoned Programming, Prentice Hall, 1994.
12. Baeten, J.C.M. and Weijland, W.P. *Process Algebra.* Cambridge University Press, 1990.
13. Berger, M., Honda, K. and Yoshida, N., Sequentiality and the π-Calculus, *TLCA01*, LNCS 2044, 29–45, Springer, 2001.
14. Berger, M., Honda, K. and Yoshida, N., Genericity and the π-Calculus, *FoSSaCs'03*, LNCS 2620, 103–119, Springer, 2003.
15. Berger, M., Honda, K. and Yoshida, N., Genericity and the π-Calculus (1). The full versiuon of [14], submitted for publication, February 2004.
16. Bornat, R., Proving pointer programs in Hoare Logic. *Mathematics and Program Construction*, 2000.
17. Caires, L. and Cardelli, L., A Spatial Logic for Concurrency (Part I). *Info. & Comp.*, Academic Press.
18. Caml Home Page, http://caml.inria.fr/.
19. Cardelli, L. Type Systems. In: pp 2208-2236, *The Computer Science and Engineering Handbook.* CRC Press, 1997.
20. Cardelli, L. and Wegner, P. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
21. Church, A. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.
22. Clarke, E. M. Jr. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *POPL'79*, pp.129–147, ACM.
23. Clint, M. and Hoare C.A.R. Program proving: jumps and functions. Acta Inf. 1 (1971) 214–224.
24. Cousot, P. Methods and logics for proving programs. *Handbook of Theoretical Computer Science*, volume B, 843-993. Elsevier, 1999.
25. Curien, P. L., Sequentiality and full abstraction. Proc. of *Application of Categories in Computer Science*, LNM 177, 86–94, Cambridge Press, 1995.
26. Curry, H. and Feys, R. *Combinatory Logic, volume 1.* North Holland, 1958,
27. Howard, W.A. The formulae-as-types notion of construction. In *To H.B.Curry: Essays on Combinatory Logic, lambda calculus and formalism* (ed. J.P. Seldin & J.R. Hindley), pp.479–490, London: Academic Press.

28. Dahl, O-J., and Hoare, C.A.R. Hierarchical Program Structure. In: *Structured Programming*, Academic Press 1972.
29. Dam, M. Proof Systems for π-Calculus Logics. *Logic for Concurrency and Synchronization.* Studies in Logic and Computing, Oxford University Press, 2003.
30. Dunfield, J. and Pfenning, F., Type Assignment for Intersections and Unions in Call-by-Value, *FoSSaCs'03*, LNCS 2620, 250-266, Springer, 2003.
31. Drossopoulou, S. Eisenbach, S. amd Khurshid, S., Is the Java Type System Sound?. *Theory and Practice of Object Systems*, 5(1), pp.3-24, 1999.
32. Gunter, C., *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992.
33. Gordon, M., Milner, A. and Wadsworth, C., Edinburgh LCF, LNCS 78, Springer, 1979.
34. Fiore, M., *Axiomatic Domain Theory in Category of Partial Maps.* Distinguished Dissertations Series, Cambridge University Press, 1996.
35. Fiore, M. and Honda, K. Recursive Types in Games: Axiomatics and Process Representation. LICS 1998, 345-356, IEEE, 1998.
36. Floyd, W. Assigning meaning to programs. *Prc. Symp. in Applied Mathematics.* Vol 19. pp.19-32, 1967.
37. Francez, N. Pnueli, A. A proof method for cyclic programs. *Acta Inf.* 9, pp.133-157, 1978.
38. GIRARD, J.-Y. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur.* PhD thesis, Universite de Paris VII, 1972.
39. Girard, J.-Y., Linear Logic, *TCS*, Vol. 50, 1–102, 1987.
40. Grossman, D., et al. Region-Based Memory Management in Cyclone, *PLDI'02*, ACM, 2002.
41. Hasegawa, M., Models of Sharing Graphs: A Categorical Semantics of let and letrec, Distinguished Dissertation Series, Springer-Verlag (1999).
42. The Haskell home page, http://haskell.org.
43. Hennessy, M. and Milner, R. Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM*, 32:1, pp.137-161, 1985.
44. Carl Hewitt, Peter Bishop, Richard Steiger, A Universal Modular ACTOR Formalism for Artificial Intelligence, 235-245, IJCAI 1973.
45. Hoare, C.A.R. An axiomatic basis of computer programming. *CACM*, 12:576-580, 1969.
46. Hoare, C.A.R. Procedures and Parameters: an axiomatic approach. Lecture Notes in Mathematics 188, 102–116, Semantics of Algorithmic Languages, Springer, 1971.
47. Hoare, C.A.R. *Communicating Sequential Processes*, Comm. ACM 21, 666-677. 1978.
48. Hoare, C.A.R. *Communicating Sequential Processes*, Prentice Hall, 1985.
49. Hoare, C.A.R. and Jifeng, H. *Unifying Theories of Programming.* Prentice Hall, 1993.
50. Hoare, C.A.R. and Wirth, N., An axiomatic definition of the programming language PASCAL. Acta Inf. 2 (1973), 335–355.
51. Honda, K., Composing Processes, *POPL'96*, 344-357, ACM, 1996.
52. Honda, K. Elementary Structures for Process Theory (1): Sets with Renaming. MSCS, 2001.
53. Honda, K. *Sequential Process Logics: Soundness Proofs.* Typescript, 50pp. November 2003. Corrected and Extended in January 2004. Available at: www.dcs.qmul.ac.uk/~kohei.
54. Honda, K. *Process Logic and Duality: Part (1) Sequential Processes.* Typescript, 150pp. January 2004. Available at: www.dcs.qmul.ac.uk/~kohei.

55. Honda, K., Yoshida, N. and Berger, M., Control in the π-Calculus, *Proc. CW'04*, ACM, 2004.
56. Honda, K. and Tokoro, M. An object calculus for asynchronous communication. ECOOP'91, LNCS 512, 133–147, 1991.
57. Honda, K. and Yoshida, N. On Reduction-Based Process Semantics. *TCS*. 151, 437-486, 1995.
58. Honda, K. and Yoshida, N. Game-theoretic analysis of call-by-value computation. *TCS*, 221 (1999), 393–456.
59. Honda, K. and Yoshida, N., A Uniform Type Structure for Secure Information Flow, *POPL'02*, 81–92, ACM, 2002.
60. Honda, K. and Yoshida, N., Noninterference proofs from Flow Analysis. Available at: www.doc.ic.ac.uk/˜yoshida.
61. Huisman, M. and Jacobs, B. Java program verification via a Hoare Logic with Abrupt Termination. *FASE'00*, SLNCS 1783, pp.284-303, 2000.
62. Hyland, M. and Ong, L., "On Full Abstraction for PCF": I, II and III. *Info. & Comp.* 163 (2000), 285-408.
63. Igarashi, A. Pierce, B. and Wadler, P., *Featherweight Java.* TOPLAS.
64. Jacobs, B. et al. Reasoning about Java Classes (Preliminary Report). *OOPSAL'98*, ACM, 1998.
65. Java home page. Sun Microsystems Inc., http://www.javasoft.com/, 1995.
66. Jones, C.B. A Software Development: A Rigorous Approach. Prentice Hall, 1980.
67. Jones, C.B. Specification and Design of (Parallel) Programs. *Proc. IFIP 9th World Computer Congress.* North Holland, pp321-332, 1983.
68. Kobayashi, N., Pierce, B., and Turner, D., Linear types and the π-calculus, *TOPLAS*, 21(5):914–947, 1999.
69. Kowaltowsky, T. Axiomatic approach to side effects and general jumps. *Acta Informatica 7, pp. 357-360.*
70. Landin, P., A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. *Comm. ACM*, 8, 2 (1965), 1965.
71. Laurent, O., Polarized games, LICS 2002, 265-274, IEEE, 2002.
72. Longley, J. and Plotkin, G. Logical Full Abstraction and PCF. *Tbilisi Symposium on Logic, Language and Information.* pp.333-352, Stanford, CSLI, 1998.
73. Mendelson, E., Introduction to Mathematical Logic (third edition). Wadsworth Inc., 1987.
74. Pitts, A.M., Existential Types: Logical Relations and Operational Equivalence, Proceedings ICALP'98, LNCS 1443, 309–326, Springer, 1998.
75. Pitts, A.M., Parametric Polymorphism and Operational Equivalence, Mathematical Structures in Computer Science, 2000, 10:321–359.
76. Milner, R., Processes: a mathematical model of computing agents. *Logic Colloquium '73*, North Holland, 1973.
77. Milner, R., Fully Abstract Models of Typed Lambda-Calculi, Theoretical Computer Science, volume 4 (1977), 1–22, 1997.
78. Milner, R., A Calculus of Communicating Systems, LNCS 92, Springer, Berlin, 1980.
79. Milner, R., Communication and Concurrency, Prentice Hall, 1989.
80. Milner, R., Functions as Processes, *MSCS*. 2(2):119–141, 1992,
81. Milner, R., Polyadic π-Calculus: a tutorial. *Proceedings of the International Summer School on Logic Algebra of Specification*, Marktoberdorf, 1992.
82. Milner, R., Speech by Robin Milner on receiving an Honorary Degree from the University of Bologna, ICALP'97, 1997, http://www.cs.unibo.it/icalp/Lauree_milner.html.

83. Milner, R., Tofte, M. and Harper, R.W, The Definition of Standard ML, MIT Press, 1990.
84. Milner, R., Parrow, J. and Walker, D., A Calculus of Mobile Processes, *Info. & Comp.* 100(1):1–77, 1992.
85. Milner, R., Parrow, J.G. and Walker, D.J., Modal logics for mobile processes, *TCS*, Vol 114, pp.149–171, 1993.
86. Milner, R., What's in a name?, in a volume of papers written in honour of Roger Needham, available at: http://www.cl.cam.ac.uk/users/rm135/.
87. Morrisett, G., Walker, D. G. Crary, K, and Glew, W./ From System F to Typed Assembly Language. In ACM TOPLAS, 21(3):528-569, May 1999.
88. Morrisett, G. et al. Cyclone: A Safe Dialect of C. Usenix Annual Technical Conference, Monterey, CA, June 2002.
89. Microsoft Corporation, .NET Framework Developer's Guide, `http://msdn.microsoft.com`, 2003.
90. Moggi, E., Notions of Computations and Monads. *Information and Computation*, 93(1):55–92, 1991.
91. Nauer, P. et al. Reviseed Report on the Algorithmic Language Algol 60. *Comm. ACM*, 6, 1 (1963).
92. Nauer, P. Proof of algorithms by general snapshots. *BIT 6*, pp.310–316, 1966.
93. Nickau, M., Hereditarily Sequential Functionals, LNCS 813, pp.253–264, Springer-Verlag, 1994.
94. Oheimb, D.V. Hoare Logic for Java in Isabelle/HOL. *Concurrency: Practice and Experience.* John Wiley, 2002.
95. O'Hearn, P., Yang, H. and Reynolds, J., Separation and Information Hiding, POPL'04, 268–280, 2004.
96. Pierce, B.C., *Types and Programming Languages*, MIT, 2002.
97. Pierce, B.C. and Sangiorgi. D, Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.
98. Plotkin, G., Call-by-name, Call-by-value and the lambda-calculus, *TCS*, 1:125–159, 1975.
99. Plotkin, G. and Abadi, M., A Logic for Parametric Polymorphism, *LICS'98*, 42–53, IEEE Press, 1998.
100. Poetzsh-Heffter, A. and Muller, P. A programming logic for sequential Java. ESOP'99, LNCS 1576, pp.162-176, Springer, 1999,
101. Pratt, V.R., Six Lectures on Dynamic Logic, Foundations of Computer Science III, Part 2, 53-82, Mathematical Centre Tracts 109, Amsterdam, 1980.
102. Whitehead, A. N. and Russell, B., *Principia Mathematica*, I–III, 1910–1913.
103. Reynolds, J. C., Types, abstraction and parametric polymorphism. In *Information Processing 83* (1983), R. E. A. Mason, Ed.
104. Reynolds,J.C., Intuitionistic Reasoning about Shared Mutable Data Structure, Millennial Perspectives in Computer Science, Oxford, 1999.
105. Reynolds, J.C., Separation logic: a logic for shared mutable data structures. Invited Paper, LICS, 2002.
106. Sangiorgi, D., π-calculus, internal mobility, and agent-passing calculi. *TCS*, 167(2):235–271, 1996.
107. Kleist, J. and Sangiorgi, D., Imperative Objects and Mobile Processes, , Journal of Science of Computer Programming, Volume 44, Number 3, September 2002, 293-342.
108. Scott, D. and Strachey, C., Towords a mathematical semantics for computer languages, Proceeding of 21st Symposium on Computers and Automata, 19–46, Polytechinc Institute of Brooklyn, 1971.

109. Smith, G. and Volpano, D., Secure information flow in a multi-threaded imperative language, pp.355–364, *POPL'98*, ACM, 1998.
110. Talpin, J-P, Jouvelot, P. The type and effect discipline. *LICS'92*, pp.162-173, 1992.
111. Turner, David N. *The Polymorphic Pi-Calculus: Theory and Implementation.* Ph.D. thesis, University of Edinburgh, 1996.
112. Walker, D., Objects in the π-calculus. *Info. Comp.*, 116(2), pp.253-271, 1995.
113. Yoshida, N., Graph Types for Monadic Mobile Processes. Proc. of 16th FST/TCS, LNCS 1180, 1996.
114. Wadler, P. *Theorems for Free.* Functional Programming and Computer Architecture, Addison Wesley, 1989.
115. Winskel, G. *The Formal Semantics of Programming Languages.* MIT Press, 1993.
116. Yoshida, N., Berger, M. and Honda, K., Strong Normalisation in the π-Calculus, *LICS'01*, 311–322, IEEE, 2001. Full version to appear in Journal of *Info. & Comp.*.
117. Yoshida, N., Honda, K. and Berger, M. Linearity and Bisimulation, FoSSaCs 2002, LNCS 2303, pp.417–433, Springer, 2002.
118. Woodcock, J. and Davies, J. *Using Z: specification, refinement and proof.* Prentice-Hall International, 1996.

A Syntax of Processes

A.1 Structural Rules and Reduction

The relation $\equiv$ is the least congruence generated by $\equiv_\alpha$ and the following equations.

$$\begin{array}{llll} P|\mathbf{0} \equiv P & P|Q \equiv Q|P & (P|Q)|R \equiv P|(Q|R) & \\ (\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0} & (\boldsymbol{\nu}\, xy)P \equiv (\boldsymbol{\nu}\, yx)P & ((\boldsymbol{\nu}\, x)P)|Q \equiv (\boldsymbol{\nu}\, x)(P|Q) & (x \notin \mathsf{fn}(Q)) \end{array}$$

The reduction relation $\longrightarrow$ is generated by the following rules.

$$\begin{array}{rl} (\text{BRA}) & x[\&_i(\vec{y}_i).P_i] \mid \overline{x}\mathtt{in}_i(\vec{y}_i)Q \longrightarrow (\boldsymbol{\nu}\, \vec{y}_i)(P_i \mid Q) \\ (\text{COM}_!) & !\, x(\vec{y}).P \mid \overline{x}(\vec{y})Q \longrightarrow !\, x(\vec{y}).P|(\boldsymbol{\nu}\, \vec{y})(P|Q) \\ (\text{RES}) & P \longrightarrow Q \implies (\boldsymbol{\nu}\, x)P \longrightarrow (\boldsymbol{\nu}\, x)Q \\ (\text{PAR}) & P \longrightarrow P' \implies P|Q \longrightarrow P'|Q \\ (\text{CONG}) & P \equiv P' \longrightarrow Q' \equiv Q \implies P \longrightarrow Q \end{array}$$

We also list the reduction of reference agents.

$$\begin{array}{rll} (\text{READ}) & \mathsf{Ref}\langle xy\rangle|\overline{x}\mathsf{read}(c)P & \longrightarrow \mathsf{Ref}\langle xy\rangle|(\boldsymbol{\nu}\, c)(\overline{c}\langle y\rangle|P) \\ (\text{WRITE}) & \mathsf{Ref}\langle xy\rangle|\overline{x}\mathsf{write}(y'c)P & \longrightarrow (\boldsymbol{\nu}\, y')(\mathsf{Ref}\langle xy'\rangle|\overline{c}|P) \end{array}$$

A.2 Computability Assumption in Process Syntax

For a sharper definability result necessary for logical full abstraction, we assume that indices of selections under $\mathbb{N}$-indexed branches are specified by a computable total function. For example, in $x[\&_{i\in\mathbb{N}}\overline{y}\mathtt{in}_{F(i)}]$, we demand $F(i)$ to be computable. In the following we formally stipulate this restriction, after a brief discussion of its background. Similar construction in the setting of game semantics is given in [62, Part III].

In standard computational calculi, including both functional and process calculi, terms are finite objects (for example they are essentially finite abstract syntax trees). The syntax of processes we use in the present paper uses infinite branching, which may look strange from the viewpoint of the tradition. While we can in fact use finite branching (which, in combination with recursion and linear typing, can precisely represent infinite objects such as natural numbers precisely), we find it more convenient to use infinite branching, because of its ability to precisely and concisely represent decomposition of computational behaviours into communication behaviours. A drawback is that, given infinite branches, it becomes natural to have uncomputable behaviours (say uncomputable functions) as processes. It should be noted that this point has no bearing in equational full abstraction results. However in logics, we may as well assert an existence of a program which computes, for example, the halting problem. The validity of such arguments relies on what class of functions we can realise in a given universe of behaviours, so that it becomes necessary to delineate the class of processes which only represent computational (effectively realisable) behaviours. In the following, we illustrate two ways to achieve this.

Simple Method. There is a simple way to restrict processes to computable ones, which is to mimic generation of all and only computable functions in standard programming languages. In this method, we simply generate processes starting from a (process representing a) successor, a predecessor, and a if-zero primitive (this is precisely as done in PCF), Combined with recursion, this directly leads to the class of computable processes. This simple construction is enough for the technical development in the present paper: the following construction however has the merit in that processes that calculate addition, multiplication, etc. can be directly represented. If we assume all processes we treat are generated in this way, and that examples and such which do not conform to such syntax are simply for the convenience of illustration, we do not have to go into any further complexity.

General Method. As a more comprehensive method for defining computable processes, we use encodings of processes into finite (effective) objects. Let $P \stackrel{\text{def}}{=} x[\&_{i\in\mathbb{N}}P_i]$ be a typable process in one of the type disciplines treated in the present work. Then, by the constraint on types, it has one free linear output port, which we call u. Given an occurrence $\overline{u}\mathtt{in}_j$, the *index* of u is j. By affine typing we know each P_i has the form:

$$\text{either} \quad \overline{y}(\vec{z}w)P'_i \quad \text{or} \quad \overline{u}\mathtt{in}_j(\vec{w})P'_i.$$

Since P is finite in depth (in that, as an abstract syntax tree, its height is finite even though it may contain a subtree of infinite branch), we can assume each occurrence of u is in a finite number of (nested) linear branches. For example, take the following process:

$$x[\&_{i\in\mathbb{N}}\overline{y}(z)z[\&_{j\in\mathbb{N}}\overline{u}\mathtt{in}_{i\times j}]].$$

Then we can see each linear output at u is prefixed by two linear branchings. In general, different occurrences of u may have different branchings.

Now starting from the zero depth, we can encode each subprocess into a computable object. Since a computable object can be represented by a finite construction, we can consider a branching as a map from indices to a set of finite objects as far as each of its branches is encoded thus. We now define the encoding $[\![P]\!]$ together with the notion of computable processes inductively as follows.

1. $x[\&_{i\in\mathbb{N}}\overline{u}\mathtt{in}_{f(i)}]$ is computable if f is a computable total function. In this case, we set its encoding as f.
2. $x[\&_{i\in\mathbb{N}}P_i]$ is computable if the map $i \mapsto [\![P_i]\!]$ is given by a computable total function, where $[\![P_i]\!]$ is given by induction. The resulting function is its encoding.
3. $\overline{x}(\vec{y}z)(\vec{R}^{\vec{y}}|P^z)$ is computable if each R_i and P are. The encoding is a sequence of the encodings of the components.
4. $!x(\vec{y}z).P$ is computable if P is. Its encoding is the same as P.
5. $P|Q$ is computable if both are. Its encoding is a sequence of the encodings of its components.
6. $(\boldsymbol{\nu}\, x)P$ is computable if P is. Its encoding is the same as P.

Note the computability from $\mathbb{N}$ to computable functions in Clause 2 can be determined by any preferred uniform encoding. Finally we note:

- As far as processes are computable in the above sense, affine processes of suitable shape and types represent precisely computable functions.
- Characterisation can also be given based on traces (cf. [62, Part III]).
- As we briefly illustrated at the beginning, we can make the conditions as given above stronger by, for example, assuming each component of branching is essentially finite in their shape (by the use of recursion, this leads to the same set of behaviours up to weak bisimilarity). The above definition is however enough for our present inquiries.

A.3 Recursion

We give the operational semantics of recursion as an operator. We present the semantics first as transition and second as reduction. For transition:

$$\text{(Rec-Unfold)}\quad \frac{P \xrightarrow{\overline{y}(\vec{w})} \xrightarrow{x(\vec{w}')} P'}{\mu y\!=\!x.P \xrightarrow{\tau} \mu y\!=\!x.(\boldsymbol{\nu}\,\vec{w})(P'[\vec{w}/\vec{w}'])} \qquad \text{(Rec-Comp)}\quad \frac{P \xrightarrow{l} P' \quad y \notin \mathsf{fn}(l)}{\mu y\!=\!x.P \xrightarrow{l} \mu y\!=\!x.P'}$$

For reduction, we first add the structural rule.

$$\mu y = x.(P|Q) \equiv (\mu y = x.P)|Q \qquad (x \notin \mathsf{fn}(P), y \notin \mathsf{fn}(Q)),$$

Then we add the following reduction rule.

$$\mu y = x.(!x(\vec{w}z).P|\overline{y}(\vec{w}z)Q|R) \longrightarrow \mu y = x.(!x(\vec{w}z).P|(\boldsymbol{\nu}\, y\vec{w}z)(P|Q)|R)$$

In either presentation, we can check that, in typed syntax, the recursion as given above gives the same behaviour as the recursion in the encoded form, up to, for example, the weak bisimilarity. In fact, the above transition and reduction are precisely the same as the result of substitution $P[x/y]$ (to be precise we need to allow the replicated prefix which prefixes its subject, e.g. $!x(c).\overline{x}(e)e.\overline{c}$). Thus, semantically speaking, it neither adds nor takes off anything to/from the original affine system.

B Alternative Construction of Models

In this appendix, we present an alternative construction of models for the linear and affine process logics. These models use behaviour constants, the starting point of the present logic (cf. Section 2) for clear articulation of semantics of the logic, as well as domain-theoretic constructions. The constructions connect the sequential typed process logic to one of its origins, the semantics of sequential higher-order functions, while inducing the models which are equivalent to those given in Sections 3 and 4 (in the sense that no difference in validity comes about).

One drawback of using the domain theory is that the corresponding construction becomes more complex when statefulness and other elements are incorporated, or worse, we cannot find a satisfactory treatment in the foregoing study of domain theory (e.g. polymorphism). This does not mean the following constructions are less worthy than the more direct constructions in the main sections: if we can have a similar model for these features, that would result in new elucidation of the underlying behavioural universe and possibly its new articulation which can even be engineeringly useful.

The following development only pertains to the constructions of ξ and the relation $P^\Gamma \models \xi \triangleright^\Gamma$. Once they are determined, the rest of the technical development remains identical with those given in the main sections (except equality of two behaviour expressions use the equivalence $\simeq_{\mathsf{b}}$). In the remainder, the model construction for linear processes is given in §B.1. This is followed by the model construction for affine processes (augmenting the one for linear processes) in §B.2, which uses some ideas from domain theory. Finally we give the corresponding construction for the total PCF$_v$-logic in §B.3.

B.1 A Model for Linear Process Logic

The grammar of behavioural constants are given as follows.

$$\alpha^{!,?} ::= \wedge_{i\in I}(\vec{\alpha}_i\beta_i)^{!} \mid \vee_{i\in I}\,(\vec{\alpha}_i\beta_i)^{?} \qquad \alpha^{\downarrow,\uparrow} ::= \mathtt{in}^{\rho}_{\mathtt{c}}(\vec{\alpha})^{\uparrow} \mid \mathtt{in}^{\rho}_{\mathtt{c}}(\vec{\alpha})^{\downarrow}.$$

Note linear input/output behavioural constants are annotated by types, just as the corresponding terms are. The well-typedness of the behaviour constants is generated as follows. The clause 1-(2) below, by which behavioural constants effectively become higher-order total functions expressed in a dualised form, becomes important when we consider interpretation of terms.

1. $\wedge_{i\in I}(\vec{\alpha}_i\beta_i)^{!}$ has type $(\vec{\tau}\rho)^{!}$ iff: (1) $\vec{\alpha}_i\beta_i$ has type $\vec{\tau}\rho$ for each $i \in I$ (where $\gamma_1..\gamma_n$ has type $\sigma_1..\sigma_n$ when each γ_i has type σ_i); and (2) $\vec{\alpha}_i \mapsto \beta_i$ defines a total function from the set of $\vec{\tau}$-typed constants to the set of ρ-typed constants. Dually for $\vee_{i\in I}(\vec{\alpha}_i\beta_i)^{?}$.
2. $\mathtt{in}^{\rho}_{\mathtt{c}}(\vec{\alpha})^{\uparrow}$ has type $\rho = [\oplus_{i\in I}\vec{\tau}_i]^{\uparrow}$ iff $\mathtt{c} \in I$ and $\vec{\alpha}$ has type $\vec{\tau}_{\mathtt{c}}$. Dually for $\mathtt{in}^{\rho}_{\mathtt{c}}(\vec{\alpha})^{\downarrow}$.

Note each well-typed behaviour constant is given a unique channel type thanks to, as in terms, the type annotation ρ in $\mathtt{in}^{\rho}_{\mathtt{c}}(\vec{\alpha})^{\uparrow}$ (which is again usually omitted as in terms). *We hereafter only consider well-typed behavioural constants.*

Model. A *pre-model* ξ is a finite map from names to positive behavioural constants (a behavioural constant is *positive* if its type is positive). ξ has type Γ iff $\xi = x_1 : \alpha_1, .., x_n : \alpha_n$ and $\Gamma = x_1 : \tau_1, .., x_n : \tau_n$ such that α_i has type τ_i $(1 \leq i \leq n)$. ξ^Γ indicates ξ which has type Γ. A *model* is simply a pre-model which indeed represent a typed process behaviour, in the following sense.

Definition 12. (model) *We say* P^Γ defines ξ^Γ*, written* $P^\Gamma \models_{\mathsf{b}} \xi$*, iff the following inductive conditions hold (below* $\mathbf{!}\xi$ *indicates* ξ *only uses* $\mathbf{!}$*-typed constants).*

1. $P^{\Gamma\cdot x:\rho} \models_{\mathsf{b}} \xi \cdot x : \mathtt{in}_i(\vec{\alpha})^\uparrow$ *iff* $P \stackrel{\overline{x}\mathtt{in}_i(\vec{y})}{\Longrightarrow} Q$ *such that* $Q^{\Gamma\cdot\vec{y}:\rho_i} \models_{\mathsf{b}} \xi \cdot \vec{y} : \vec{\alpha}$.
2. $P \models_{\mathsf{b}} \mathbf{!}\xi$ *iff, whenever* $\xi = \xi' \cdot x : \wedge_{i\in I}(\vec{\alpha}_i\beta_i)^{\mathbf{!}}$ *we have* $P \stackrel{x(\vec{y}z)}{\longrightarrow} P'$ *such that, for each* $i \in I$*,* $R \models_{\mathsf{b}} \vec{y}:\vec{\alpha}_i$ *implies* $(\boldsymbol{\nu}\, x\vec{y})(P'|R) \models_{\mathsf{b}} \xi/x \cdot z : \beta_i$.

A model ξ *is a pre-model such that* $P^\Gamma \models_{\mathsf{b}} \xi$ *for some* P^Γ*. Given two models* $\xi^\Gamma_{1,2}$*,* $\xi_1 \simeq_{\mathsf{b}} \xi_2$ *iff* $\xi_{1,2}$ *define the same set of linearly typed processes. We also write* $\alpha_1 \simeq_{\mathsf{b}} \alpha_2$ *for* $x:\alpha_1 \simeq_{\mathsf{b}} x:\alpha_2$ *for some* x*.*

Let us say a behaviour constant α is *definable* iff $x:\alpha$ is a model for some x. The following convention demands that the whole technical development is carried out entirely within the universe of behaviours represented by typed processes.

Convention 7. *Henceforth we assume all behaviour constants considered are well-typed and definable.*

The proof of the following is given in detail in [53].

Lemma 4. *1. Let* $P^\Gamma \models_{\mathsf{b}} \xi$*. Then* $P^\Gamma \cong Q^\Gamma$ *iff* $Q^\Gamma \models \xi$*.*
2. $\xi^\Gamma_1 \simeq_{\mathsf{b}} \xi^\Gamma_2$ *if and only if* $P^\Gamma \models \xi_{1,2}$ *for some* P*.*

We omit interpretation of formulae. From the results above, we can easily show validity coincides between this model and the one in Section 3.

B.2 A Model for Affine Process Logic

For having the model for the affine process logic, we augment the behavioural constants with the constant ω^ρ at each linear output type ρ. The well-formedness of behaviour constants of $\mathbf{!}$-$\mathbf{?}$ types is redefined so that, instead of $\wedge_{i\in I}(\vec{\alpha}_i\beta_i)^{\mathbf{!}}$ and $\vee_{i\in I}(\vec{\alpha}_i\beta_i)^{\mathbf{?}}$ being total functions, we require them to be total continuous functions, regarding ω as the bottom element (continuity is not essential but makes the construction leaner). This gives a notion of *pre-models*, again ranged over by $\xi, \xi', \ldots$. Using these pre-models, the notion of models (cf. Definition 12) is defined through the relation $\models_{\mathsf{b}}$, adding one clause to those in Definition 12. For reference we again give the whole definition: the clause 0 is the added one.

0. $P \models_{\mathsf{b}} \mathbf{!}\xi, x : \omega$ iff $P \Uparrow$. $\xi, \xi', \ldots$ range over the resulting set of models.
1. $P^{\Gamma\cdot x:\rho} \models_{\mathsf{b}} \xi \cdot x : \mathtt{in}_i(\vec{\alpha})^\uparrow$ iff $P \stackrel{\overline{x}\mathtt{in}_i(\vec{y})}{\Longrightarrow} Q$ such that $Q^{\Gamma\cdot\vec{y}:\rho_i} \models_{\mathsf{b}} \xi \cdot \vec{y} : \vec{\alpha}$.

2. $P \models_{\mathsf{b}} \mathbf{!}\xi$ iff, whenever $\xi = \xi' \cdot x : \wedge_{i \in I}(\vec{\alpha}_i \beta_i)^{\mathbf{!}}$ we have $P \xrightarrow{x(\vec{y}z)} P'$ such that, for each $i \in I$, $R \models_{\mathsf{b}} \vec{y} : \vec{\alpha}_i$ implies $(\boldsymbol{\nu}\, x\vec{y})(P'|R) \models_{\mathsf{b}} \xi/x \cdot z : \beta_i$.

Then, as before, a model is a (well-typed) definable premodel, that is ξ such that $\xi \models_{\mathsf{b}} P$ for some affine process P (note definability means such behaviours as Parallel Or are excluded in our subsequent discussions). We again obtain the result corresponding to Lemma 4, using which the equivalence of this model and the one in Section 4 is again established.

The construction extends to stateful sequential processes, even though some investigation would be necessary to set up a similar framework when we incorporate, for example, polymorphism.

B.3 A Model for Total PCF$_v$-Logic

While we can equally use a syntactic universe, here we work with the familiar universe of partial CPOs, which is closer to our construction in Section 4. $\mathbb{B}$ and $\mathbb{N}$ are interpreted as the standard bottomless flat CPOs (whose elements are identified with constants in PCF$_v$); $\alpha \Rightarrow \beta$ becomes the set of partial continuous functions from α to β. We call these elements together with $\perp$ (the undefined computation) *behaviours*, which are together ranged over by $\lambda, \kappa, \ldots$. Then the two relations, $\models_{\mathsf{b}}$ and $\models_{\mathsf{b}}^{\mathrm{val}}$, are generated by the following clauses. Let M and V be closed.

1. $M \models_{\mathsf{b}} \perp$ iff $M \Uparrow$.
2. $M \models_{\mathsf{b}} \lambda \neq \perp$ iff $M \Downarrow V$ and $V \models_{\mathsf{b}}^{\mathrm{val}} \lambda$.
3. $V^{\mathbb{N}} \models_{\mathsf{b}}^{\mathrm{val}} \mathbf{n}$ iff $V = \mathbf{n}$.
4. $V^{\mathbb{B}} \models_{\mathsf{b}}^{\mathrm{val}} \mathbf{b}$ iff $V = \mathbf{b}$.
5. $V^{\alpha \Rightarrow \beta} \models_{\mathsf{b}}^{\mathrm{val}} \lambda$ iff, whenever $W^{\alpha} \models_{\mathsf{b}}^{\mathrm{val}} \kappa$, we have $VW \Downarrow$ iff $\lambda(\kappa)$ (application of κ to λ) is defined and, if so, $VW \models_{\mathsf{b}}^{\mathrm{val}} \lambda(\kappa)$.

The two relations $\models_{\mathsf{b}}$ and $\models_{\mathsf{b}}^{\mathrm{val}}$ for interpretation are best illustrated using their translations into the affine process logic. First let $\langle\!\langle \lambda^{\alpha} \rangle\!\rangle^{\mathrm{val}}$ be the corresponding behavioural constant in α°, defined inductively as, e.g.:

$$\langle\!\langle 1 \rangle\!\rangle^{\mathrm{val}} \stackrel{\mathrm{def}}{=} \wedge(\ \varepsilon\ \mathtt{in}_1()^{\uparrow})^{\mathbf{!}}, \qquad \langle\!\langle n \mapsto 2n \rangle\!\rangle^{\mathrm{val}} \stackrel{\mathrm{def}}{=} \wedge_n(\ \langle\!\langle n \rangle\!\rangle^{\mathrm{val}}\ \mathtt{in}_{2n}()^{\uparrow})^{\mathbf{!}}.$$

Then we set $\langle\!\langle \lambda^{\alpha} \rangle\!\rangle$ to be $\omega^{(\alpha^{\circ})^{\uparrow}}$ when $\lambda = \perp$, and $(\langle\!\langle \lambda \rangle\!\rangle^{\mathrm{val}})^{\uparrow}$ if not. We can now encode $M \models_{\mathsf{b}} \lambda$ as $\langle\!\langle M \rangle\!\rangle_u \models_{\mathsf{b}} u : \langle\!\langle \lambda \rangle\!\rangle$, while $V \models_{\mathsf{b}}^{\mathrm{val}} \lambda$ is encoded as $\langle\!\langle V \rangle\!\rangle_u \models_{\mathsf{b}} u : \langle\!\langle \lambda \rangle\!\rangle^{\mathrm{val}}$. Thus having two relations in PCF$_v$-logic compensates for not having $\uparrow$-typed behaviours.

We say λ *is definable* iff $M \models_{\mathsf{b}} \lambda$ for some M. λ is *total* if it is not $\perp$. $\lambda \simeq_{\mathsf{b}} \lambda'$ if they define the same set of closed terms. A *model* ξ maps names to total, definable behaviours. The rest precisely follows §B.3 above.

C Refined Typing Rules for Stateful Processes

Below we present the refined typing rules for parallel composition for the π-calculus with open state.

$$\textsf{(Par-}\downarrow\uparrow\textsf{)}\quad \frac{\vdash P_1 \triangleright \Gamma_1 \,;\, x : \rho^{\uparrow} \qquad \vdash P_2 \triangleright \Gamma_2, x : \overline{\rho}^{\downarrow} \,;\, y : \tau^{\uparrow}}{\vdash P_1 | P_2 \triangleright \Gamma_1 \cup \Gamma_2,\, x : \updownarrow \,;\; y : \tau}$$

$$\textsf{(Par-}\mathbf{!?}\textsf{)}\quad \frac{\vdash P_1 \triangleright \Gamma_1 \,;\, \mathbf{!!}_{rw}\Delta_1, \mathbf{!}\Theta \qquad \vdash P_2 \triangleright \Gamma_2, \mathbf{?}\overline{\Theta} \,;\, \mathbf{!!}_{rw}\Delta_2}{\vdash P_1 | P_2 \triangleright \Gamma_1 \cup \Gamma_2 \,;\, \Delta_1, \Delta_2, \Theta}$$

$$\textsf{(Par-}\mathbf{!?}\textsf{-thread)}\quad \frac{\vdash P_1 \triangleright \Gamma_1 \,;\, \mathbf{!!}_{rw}\Delta, \mathbf{!!}_{rw}\Theta \qquad \vdash P_2 \triangleright \Gamma_2, \mathbf{??}_{rw}\overline{\Theta} \,;\, x : \tau^{\uparrow}}{\vdash P_1 | P_2 \triangleright \Gamma_1 \cup \Gamma_2 \,;\, \Delta, \Theta, x : \tau}$$

This refined typing is based on distinct ways threads/nonthreads compose. such distinction already existed in the stateless linear/affine typings we treated in Sections 3 and 4: however we never need the corresponding differentiations at the level of proof rules.

First, we need a classification of threads (recall a *thread* is a process which has a free linear output channel). A thread is *pure* if it does not contain a free replicated channel; a thread is *active* if it does not contain a free linear input (we can prove an active thread either contains a reduction, or, if not, has an active free output: the latter means it is in a terminating state). Now we illustrate these rules.

1. (Par-$\downarrow\uparrow$) is for composing two pure threads to create another pure thread. It is possible, indeed, to create a trivial process (with no positive channels) from two pure processes, by a circular composition. This case is omitted since such a process is semantically negligible (e.g. it is equal to $\mathbf{0}$ up to strong bisimilarity).
2. (Par-$\mathbf{!?}$) is for composing two non-threads to create another non-thread (without circular composition).
3. (Par-$\mathbf{!!}_{rw}\mathbf{??}_{rw}$) is for composing a pure thread and a non-thread (thus automatically without circular composition). Since composition of two non-pure threads (or one pure thread and one non-pure thread) is reducible to composition of two pure threads followed by composition of the resulting process and a non-thread, this rule combined with the above two rules is enough for generating processes.

One may observe that a pure thread P_1 in (Par-$\mathbf{!!}_{rw}\mathbf{??}_{rw}$) above does not contain a free linear input. Since the proof rule of the same name in §9.6 also uses the same typing, the proof system in fact does not treat such a case either. While we can easily treat this case, we do need a different proof rule, and in fact a different sequent, for treating a non-pure, non-active thread. Since this case seems of little practical interest the present version dispenses this case. As far as we are interested in active threads (either pure or non-pure), we can always construct them from composition of pure active/non-active threads together with replicated processes, so this does not lead to loss of generality.