

# *From Process Logic to Program Logic*

Kohei Honda

*Queen Mary, London*

*Department of Computer Science*

*Queen Mary, London*

---

## **Abstract**

We present a process logic for the  $\pi$ -calculus with the linear/affine type discipline (Berger et al. 2001; Berger et al. 2003; Honda and Yoshida 2002; Honda and Yoshida 2003; Honda et al. 2004; Yoshida et al. 2001; Yoshida et al. 2002). 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 behaviour of an environment as the dual of that of a process in an assertion, which is crucial for obtaining compositional proof systems. From the process logic we can derive compositional program logics for various higher-order programming languages, whose soundness is proved via their embeddings into the process logic. In this paper, the key technical framework of the process logic and its applications is presented focussing on pure functional behaviour and a prototypical call-by-value functional language, leaving the full technical development to (Honda 2004a; Honda 2004b).

---

## **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 feedback 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 (Hoare 69), Hoare Logic, developed on the basis of earlier work by Floyd (Floyd 67) and Naur (Naur 66). 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 it 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 (Apt 81; Cousot 99)). 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, cf. (Oheimb 2002; Jacobs et al. 98; Poetzsf-Heffter and Muller 99; Leavens and Baker 99).

Compositional program logics make it possible to reason about observable properties of software starting from their constituent 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  $\pi$ -calculus, introduced by Milner, Parrow and Walker (Milner 92) — has a simple operation, *communication of names*, and a small set of algebraic operators, including *parallel composition*. The  $\pi$ -calculus is a calculus of concurrency, developed along the line of such process calculi as CCS (Milner 89), CSP (Hoare 85) and ACP (Baeten and Weijland 90). But, as the originators noted (Milner 92), 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 higher-order sequential computation (Milner 97). In fact, if we constrain the composition of processes using a class of type annotations (which come from Linear Logic (Girard 87; Laurent 02), Game Semantics (Abramsky et al. 00; Honda and Yoshida 99; Hyland and Ong 2000) and study of types for the  $\pi$ -calculus (Milner 92; Pierce and Sangiorgi 96; Kobayashi et al. 99; Honda 96)), we can embed a wide class of programming languages into name passing processes without

losing essential semantic properties (Berger et al. 2001; Yoshida et al. 2001; Yoshida et al. 2002; Honda and Yoshida 2002; Berger et al. 2003; Honda and Yoshida 2003; Honda et al. 2004). 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  $\pi$ -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  $\pi$ -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:

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

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 for processes offer us — the moulds by which we can shape fine-grained process behaviour. In fact, one of the distinguishing features of the  $\pi$ -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 logic starts from typed processes. Following Hennessy-Milner Logic (Hennessy and Milner 85), 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, 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  $\pi$ -calculus. The present paper illustrates these key technical elements in the simplest setting, focussing on purely functional sequential processes and a basic call-by-value higher-order language. The experiment with a large class of sequential behaviour and different kinds of higher-order programming languages (including call-by-name, two forms of polymorphisms, data structure with destructive update, objects and global and local state) is reported in (Honda 2004a). It also reports how

the original Hoare Logic is recoverable from a stateful version of the process logic, suggesting fruitful connections to the foregoing studies on program logics.

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 (some of the remaining topics are also summarised in §1.3 later).

### *1.1 Summary of Technical Contributions*

Apart from the general direction to use the typed logics for the  $\pi$ -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) leads to a highly structured form of Hennessy-Milner logic for the  $\pi$ -calculus. The representation of the behaviour of an environment in assertions yields simple compositional proof rules.
2. The use of fully abstract embeddings of programming languages into typed  $\pi$ -calculus to derive, analyse and verify program logics for these languages. This extends various semantic embeddings of programming languages in processes (Milner 90; Walker 95; Berger et al. 2001; Yoshida et al. 2001) 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, among others, higher-order functions and procedures, polymorphisms, and complex data structures.

Regarding the third point, a uniform treatment of higher-order features of programming languages in compositional program logics may not have been known so far, in spite of many studies on program logics in the past. The present work offers a general technical solution to this problem, using decomposability of language constructs into fine-grained dynamics of name passing processes and the precise logical analysis of the resulting representation through the typed process logics.

### *1.2 Related work*

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 in these three strands.

**(1) Program Logics.** Among extensions of Hoare's axiomatic method, the proof rules for procedures studied by Hoare and Clint (Hoare 71; Clint and Hoare 71) predict the logical treatment of replicated processes in the present work. Kowaltowsky (Kowaltowsky 77) proposes a way to use names to express return values of procedures. More recently, Abadi and Leino (Abadi and Leino 97) 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 (Jones 83) 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 (Francez and Pnueli 78)). 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 and their colleagues (Reynolds 99; Reynolds 2002; O'Hearn et al. 2004; Bornat 2000) study extensions of Hoare's 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. Another, and related, aspect of their approach is its focus on resource-sensitive (rather than observational) behaviours of programs. Because of these differences, direct comparisons are difficult. We believe different approaches to program logics will contribute to the integrated understanding of a diverse aspects of language semantics.

In a tradition different from Hoare Logic, equational logics for the  $\lambda$ -calculi have been studied since the classical work by Curry (Curry and Feys 58) and Church (Church 41). LCF (Gordon 79) augments the standard equational theory of the  $\lambda$ -calculus with Scott's fixed point induction. Other prominent logics along this line include those for polymorphic  $\lambda$ -calculi (Plotkin and Abadi 98; Abadi and Curien 93). 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, (Gordon 79; Plotkin and Abadi 98; Abadi and Curien 93) (cf. (Wadler 89; Pitts 1443; Pitts 2000; Abadi 2000; Berger et al. 2003)) on the other.

The intersection type disciplines (Barbanera et al. 95; Dunfield and Pfenning 2003) offer one of the most general ways to specify behaviours of programs among various type systems for functions. Apart from the fact that they have mainly been studied for untyped calculi, there are two main differences from the presented framework. First, specifications in intersection types are based on conjunction and entailment of a specific kind, whereas the logical language in the present framework allows the full use of standard logical connectives. Second, the properties expressible in intersection types are certain closed sets in the corresponding CPOs, whereas formulae in the presented logic essentially encompass arbitrary observable properties of programs.

A work on program logics closely related with intersection type disciplines is Abramsky's domain logic (Abramsky 91), 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. Domains offer clean and rich notions of algebras for a wide class of computational behaviours. At the same time, the representation using domains may not extend easily to several significant classes of behaviours such as polymorphism and local state. Name passing processes are more versatile in representing computation, even though capturing specific classes of behaviours and their algebra from this large universe, necessary for clean logical articulation, can be non-trivial. As in intersection types, Abramsky’s domain logic considers certain upper-closed subsets of domains (or equivalently open sets in the corresponding topologies) as specifiable properties.

**(2) Theories of Processes.** In (Milner 73), 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 (Milner 80) and its behavioural theories, and, one decade later, to the  $\pi$ -calculus (Milner 92). It is also notable that Hoare’s early exposition of CSP (Hoare 78) 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 algebras (Milner 89; Hoare 85; Baeten and Weijland 90) form the basis of the present work, including behavioural equivalences, consistent syntactic treatment of various process operators, and logics for processes.

The embeddings of various computational calculi and programming languages in the  $\pi$ -calculus have been studied by many researchers, starting from Milner (Milner 90) (for higher-order functions) and Walker (Walker 95) (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 (Longley 98)). This semantic preservation crucially relies on the so-called definability property of the embedding, which is the idea originating in denotational semantics (Plotkin 75; Milner 77).

As already noted, Hennessy-Milner Logic (Hennessy and Milner 85), 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 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. The generality of logical representation in Hennessy-Milner logics is a fundamental technical underpinning, offering a detailed and 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  $\pi$ -calculus include (Milner 93) (for characterising different

bisimilarities using Hennessy-Milner logics) and (Caires and Cardelli 02) (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  $\pi$ -calculus becomes quite complex, as detailed by Dam (Dam 2003). This poses an interesting question: does the present framework — explicit representation of the environment — also help in the untyped (or weakly typed) setting for a simple compositional logic?

The embeddability of a program logic in a process logic is first demonstrated by Milner (Milner 89), 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 (Milner 89), 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 (Whitehead and Russell 1910). 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 (Pierce 2002) started with the use of typed  $\lambda$ -calculi as the basis of theories of programming language, whose origins can be found in the early work by Landin (Landin 65) as well as Scott-Strachey's denotational semantics (Scott and Strachey 71) (which, in effect, models programs as certain classes of typed higher-order functions). Typed  $\lambda$ -calculi are also closely related with proof theory by so-called Curry-Howard correspondence (Howard).

ML (Milner 90; CAML), developed by Milner and his colleagues (originally a meta-language for implementing LCF (Gordon 79)), positions types as a central organising principle of programming in its language design, combining higher-order functions with imperative features. Through the studies of ML, Haskell (Haskell) 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 (Java) and C# (Microsoft 2003); and into the low-level code as studied by Morrisett and others (Morrisett 99; Grossman 2002).

In the present study, types play the role 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  $\lambda$ -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  $\pi$ -calculus known as *sorting* (Milner

92) and its refinements (Pierce and Sangiorgi 96; Kobayashi et al. 99; Honda 96; Yoshida 96); and, on the other, decomposition of types for higher-order functions into interaction originating in Linear Logic (Girard 87) (introduced in the context of studies of the correspondence between typed  $\lambda$ -calculi and proof theories noted before), which later led to type structures in game semantics (Abramsky et al. 00; Hyland and Ong 2000; Nichau 94). 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 (Laurent 02). 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.

### *1.3 Further Topics*

The present work is but a modest initial step towards language-independent compositional logics for software behaviour. Its extensions to logics to various sequential behaviours are studied in (Honda 2004a; Honda and Yoshida 2004; Honda et al. 2004). Some of the remaining issues include: studies of models for the presented logics, both in theoretical and practical aspects, which would shed light on the methods of calculation of validity in the present logics; extensions of the framework to the realm of behaviours beyond those treated so far, including exceptions, input/output, concurrency and distribution; Experiment with 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  $\pi$ -calculus logic; Search for natural specification methods for language-independent correctness of software, including experiments with practically feasible notations; incorporation of the present theory into a general specification framework such as VDM (Jones 80) and Z (Woodcock and Davies 96); 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*

In the remainder, Section 2 reviews the syntax of the  $\pi$ -calculus and informally discusses key ideas of the present work using simple examples. Section 3 formally introduces the typed process logic, including assertions, their semantics, and proof rules. Section 4 briefly reviews the syntax of the call-by-value PCF and its process encoding. Section 5 derives a compositional program logic for the call-by-value PCF from the process logic, shows it is fully abstractly embeddable into the process logic, demonstrates how a fully abstract logical embedding leads to a simple proof of soundness of the proof rules, and concludes with simple inference examples.

### 1.5 Notation

Logical connectives are used with their standard precedence and association: 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 denotes logical equivalence. 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 program/process logics are based) is Mendelson's textbook (Mendelson 87).

## 2 Preview of Main Ideas

### 2.1 Processes with Affine Sequential Types

The  $\pi$ -calculus used in this paper is a typed variant of the standard asynchronous  $\pi$ -calculus (Honda and Tokoro 91; Boudol 92). The following gives its reduction rule.

$$x(\vec{y}).P \mid \bar{x}(\vec{v}) \longrightarrow P[\vec{v}/\vec{y}] \quad (1)$$

Here  $\vec{y}$  denotes a potentially empty vector  $y_1 \dots y_n$  of names,  $\mid$  denotes parallel composition,  $x(\vec{y}).P$  is an input, and  $\bar{x}(\vec{v})$  is an asynchronous output. Operationally, this reduction represents the consumption of an asynchronous message by a receptor. The idea extends to a receptor with replication,  $!x(\vec{y}).P$ :

$$!x(\vec{y}).P \mid \bar{x}(\vec{v}) \longrightarrow !x(\vec{y}).P \mid P[\vec{v}/\vec{y}]. \quad (2)$$

where the replicated process remains in the configuration after reduction.

Types for processes prescribe usage of names. To be able to do this with precision, it is important to control dynamic sharing of names. For this purpose, it is useful to restrict name passing to *bound (private) name passing*, where only bound names are passed in communication (Sangiorgi 96). Using bound name passing leads to simple proof rules without sacrificing expressiveness. Syntactically we restrict outputs to the form  $(\nu \vec{y})(\bar{x}(\vec{y}) \mid P)$  (where  $(\nu \vec{y})$  indicate name hiding: names in  $\vec{y}$  should be pairwise distinct), which we henceforth write  $\bar{x}(\vec{y})P$  (observe the lack of “.” in this prefix, indicating the lack of synchronisation). The restriction of syntax to the bound output gives the following simpler form of dynamics.

$$\begin{aligned} x(\vec{y}).P \mid \bar{x}(\vec{y})Q &\longrightarrow (\nu \vec{y})(P \mid Q) \\ !x(\vec{y}).P \mid \bar{x}(\vec{y})Q &\longrightarrow !x(\vec{y}).P \mid (\nu \vec{y})(P \mid Q) \end{aligned}$$

“ $\bar{x}(\vec{y})Q$ ” indicates that  $\bar{x}(\vec{y})$  is an asynchronous output exporting  $\vec{y}$  which are originally local to  $Q$ . After communication,  $\vec{y}$  are shared between  $P$  and  $Q$ .

We also use *branching*  $x[\&_{i \in I}(\vec{y}_i).P_i]$ , where  $I$  is a finite or countable indexing set, and *selection*  $\bar{x}\text{in}_i(\vec{w})$ , the latter written  $\bar{x}\text{in}_i(\vec{w})P$  in the bound output notation. These constructs are used for representing base values, conditionals and the case construct (Berger et al. 2001; Yoshida et al. 2001; Girard 87). They play a basic role in the typed setting, just as the case/injection constructs in the typed  $\lambda$ -calculus with sums. The associated reduction involves selection of one branch, discarding remaining ones, combined with name passing.

$$x[\&_{i \in I}(\vec{y}_i).P_i] \mid \bar{x}\text{in}_j(\vec{y}_j)P \longrightarrow (\nu \vec{y}_j)(P_j \mid P)$$

We can now summarise the formal grammar of the calculus. Below and henceforth  $x, y, \dots$  range over a countable set of names.

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

Each  $\bar{y}_i$  is a binder with scope  $P_i$  in  $x[\&_{i \in I}(\bar{y}_i).P_i]$ ;  $\bar{y}$  is a binder with scope  $P$  in  $!x(\bar{y}).P$  and  $\bar{x}(\bar{y})P$ ;  $x$  is a binder with scope  $P$  in  $(\nu x)P$ . We assume the standard bound name convention, and identify processes up to the  $\alpha$ -equality. We often omit  $I$  in  $x[\&_{i \in I}(\bar{y}_i).P_i]$ . In the present paper we only consider  $\{*\}$  (a distinguished singleton),  $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$ , and  $\mathbb{N} = \{0, 1, \dots\}$  as the indexing sets for branching.<sup>1</sup> When the indexing set is  $\{*\}$ , the branching is written  $x(\bar{y}).P$ , dually for output.  $P|Q$  is parallel composition of  $P$  and  $Q$ ,  $(\nu x)P$  is the result of hiding  $x$  in  $P$ , and  $\mathbf{0}$  is the inaction. We usually omit the empty vector and  $\mathbf{0}$ 's, writing, for example,  $\bar{x}(\bar{y})\mathbf{0}$ ,  $x.P$  for  $x().P$ ,  $x[\&_i.P_i]$  for  $x[\&_i().P_i]$ , and  $\bar{x}\text{in}_i$  for  $\bar{x}\text{in}_i().\mathbf{0}$ . The structural congruence  $\equiv$  (which includes the  $\alpha$ -equality) is standard (Yoshida et al. 2001). The reduction relation  $\longrightarrow$  is generated starting from the rules already given, closing it under contexts except input prefixes, taking processes modulo  $\equiv$ .

As untyped processes, the above syntax can represent a large class of sequential and concurrent behaviours. We use the *sequential affine type discipline* (Berger et al. 2001) for constraining their behaviour to purely functional ones. We use the following *action modes* (Berger et al. 2001; Yoshida et al. 2001), which represent different modes of communication actions at channels.

$$\begin{array}{cc} \downarrow & \text{Affine input} & \uparrow & \text{Affine output} \\ ! & \text{Replicated server} & ? & \text{Client request to !} \end{array}$$

We also use  $\updownarrow$  which stands for uncomposability.  $\downarrow$  indicates a non-replicated input which receives an output at most once.  $\uparrow$  indicates an output which is done at most once. Thus  $\downarrow$  and  $\uparrow$  are dual to each other. When we compose  $\downarrow$  and  $\uparrow$  at a shared channel, we obtain  $\updownarrow$ , indicating no further composition is possible at that channel.  $!$  indicates a replicated input, while  $?$  indicates an output to  $!$  (these symbols are from (Girard 87)). Thus  $?$  and  $!$  are dual to each other. When they are composed, we obtain  $!$ , so that a replicated channel is available to an arbitrary number of dual outputs.

Using the action modes, the grammar of *channel types*, ranged over by  $\tau, \tau', \rho, \dots$ , is given as follows ( $\vec{\tau}$  denotes a vector).

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

$(\vec{\tau})^!$  and  $[\&_{i \in I} \vec{\tau}_i]^\downarrow$  (resp.  $(\vec{\tau})^?$  and  $[\oplus_{i \in I} \vec{\tau}_i]^\uparrow$ ) are sometimes collectively called *input types* (resp. *output types*). We assume an input type only carries output types (e.g. in  $(\tau)^!$ ,  $\tau$  is an output type) and vice versa.  $(\vec{\tau})^!$  indicates a replicated input which receives channels of type  $\vec{\tau}$ .  $(\vec{\tau})^?$  is its dual.  $[\&_{i \in I} \vec{\tau}_i]^\downarrow$  indicates a branching input which has  $I$ -branches and which, at each  $i$ -th branch, receives channels of type  $\vec{\tau}_i$ .  $[\oplus_{i \in I} \vec{\tau}_i]^\uparrow$  is its dual. The symbol  $\updownarrow$  is also used as a type, again denoting uncomposability.

<sup>1</sup> The use of infinitary branching is not essential (since, for example, natural numbers can be represented using the sum type and the recursive type), but is convenient for our technical development. To avoid anomalies, we assume suitable restrictions on the summands of branching as well as on the use of indices of selections so that processes are of finite size and represent only computable functions, both discussed in Appendix B

The *mode* of  $\tau$  is its outermost mode (if  $\tau \neq \downarrow$ ) or  $\downarrow$  (if  $\tau = \downarrow$ ). We often write  $\rho^p$  if the mode of  $\rho$  is  $p$ . Given  $\tau \neq \downarrow$ , the *dual* of  $\tau$ , written  $\bar{\tau}$ , is the result of exchanging, in  $\tau$ ,  $!$  and  $?$ ,  $\downarrow$  and  $\uparrow$ , as well as  $\&$  and  $\oplus$ . An *action type*  $(\Gamma, \Delta, \dots)$  is a finite map from names to channel types.  $\text{fn}(\Gamma)$  is its domain. An action type is *sequential* if it contains at most one  $\uparrow$ -typed channel (operationally this condition guarantees the existence of at most one thread in a typable process). In this paper we only use sequential action types.

A classification of channel types called *polarities* (Honda and Yoshida 2003; Laurent 02) plays a basic role in the process logic. A channel type, or a name given that type, is *positive* (resp. *negative*) if its mode is one of  $!, \uparrow$  (resp.  $?, \downarrow$ ).  $\downarrow$  is *neutral*. Intuitively, a type or a typed channel which emits, or generates, information is positive, while one which receives, or consumes, information is negative (a behavioural characterisation of this idea is given in (Honda and Yoshida 2003); note polarities are *not* directly related to the distinction between input and output).

A typed term is written  $\vdash P \triangleright \Gamma$  (read:  $P$  has type  $\Gamma$ , or  $P$  is typed under  $\Gamma$ ). The typing rules are presented in Section 3: the following discussion should be understood without details of the typing rules. A few examples of typed processes follow.

**Example 1** (typed processes)

1.  $\bar{w}\text{in}_3$  simply selects the third branch at  $w$ , for which we have  $\vdash \bar{w}\text{in}_3 \triangleright w : [\oplus_{i \in \mathbb{N}}]^\uparrow$ .
2. Let  $[[3]]_x \stackrel{\text{def}}{=} !x(w).\bar{w}\text{in}_3$ . Then  $\vdash [[3]]_x \triangleright x : \mathbb{N}^\circ$  where  $\mathbb{N}^\circ = ([\oplus_{i \in \mathbb{N}}]^\uparrow)^\downarrow$ . The process, when invoked at  $x$  with a single name, immediately outputs 3 via that name.
3. Let  $\text{double}_{xy} \stackrel{\text{def}}{=} !x(w).\bar{y}(u)u[\&_{n \in \mathbb{N}}.\bar{w}\text{in}_{n \times 2}]$ . Then we have  $\vdash \text{double}_{xy} \triangleright y : \bar{\mathbb{N}}^\circ, x : \mathbb{N}^\circ$ . When invoked at  $x$ , the process asks back at  $y$ , receives  $n$ , and gives its double as the answer to the original question.  $[[3]]_y$  and  $\text{double}_{xy}$  interact as:

$$[[3]]_y | \text{double}_{xy} | \bar{x}(w)w[\&_i \bar{f}\text{in}_i] \longrightarrow^+ [[3]]_y | \text{double}_{xy} | \bar{f}\text{in}_6.$$

4. Let  $[x \rightarrow y]^\tau \stackrel{\text{def}}{=} !x(w).\bar{y}(u)u.\bar{w}$  with  $\tau \stackrel{\text{def}}{=} ((\uparrow)^\downarrow)^\downarrow$ . Then we have  $\vdash [x \rightarrow y]^\tau \triangleright x : \tau, y : \bar{\tau}$ . This is a *copycat of type*  $\tau$  (Hyland and Ong 2000; Berger et al. 2001), which acts as a transparent link between two locations, as the following reduction indicates (below  $\approx$  is the standard weak bisimilarity (Milner 92)).

$$[x \rightarrow y]^\tau | \bar{x}(w)P \longrightarrow^+ \approx [x \rightarrow y]^\tau | \bar{y}(w)P$$

Similarly we can define a copycat for a general  $!/\downarrow$ -type  $\tau$  as follows. Below, in  $[\&_i \vec{\rho}_i]^\downarrow$ , we let  $\vec{\rho}_i \stackrel{\text{def}}{=} \rho_{i1} \dots \rho_{ij} \dots \rho_{im_i}$  for some  $m_i$  for each  $i$ ,

$$\begin{aligned} [x \rightarrow y]^\tau &\stackrel{\text{def}}{=} \begin{cases} !x(\vec{a}).\bar{y}\langle \vec{a} \rangle^{\vec{\rho}_i} & (\tau = (\vec{\rho})^\downarrow) \\ x[\&_i(\vec{a}_i).\bar{y}\text{in}_i\langle \vec{a}_i \rangle^{\vec{\rho}_{ij}}] & (\tau = [\&_i \vec{\rho}_i]^\downarrow) \end{cases} \\ \bar{x}\langle \vec{a} \rangle^{\vec{\tau}} &\stackrel{\text{def}}{=} \bar{x}(\vec{b})\Pi_i[b_i \rightarrow a_i]^{\vec{\tau}_i} \\ \bar{x}\text{in}_i\langle \vec{a} \rangle^{\vec{\tau}} &\stackrel{\text{def}}{=} \bar{x}\text{in}_i(\vec{b})\Pi_i[b_i \rightarrow a_i]^{\vec{\tau}_i}. \end{aligned}$$

Above,  $\bar{x}\langle \vec{a} \rangle^{\vec{\tau}}$  emits local names linked to free names  $\vec{a}$  by copycats. So, in effect, it sends free names  $\langle \vec{a} \rangle$  (note whenever the receiving side interrogates at one of the received names, s/he eventually reach the corresponding name in  $\vec{a}b$ ). In other words, it acts as a free output  $\bar{x}\langle \vec{a} \rangle$  using only bound outputs, similarly for  $\bar{x}\text{in}_i\langle \vec{a} \rangle^{\vec{\tau}}$ .

5. Using a generalised copycat, we can represent recursion. Let  $\vdash P \triangleright \Gamma$ ,  $\bar{y} : \bar{\tau}$ ,  $\bar{x} : \bar{\tau}^\dagger$ , write  $\Pi_i P_i$  for the  $n$ -fold parallel composition, and define:

$$\mu \bar{y} \bar{\tau} = \bar{x}.P \stackrel{\text{def}}{=} (\nu \bar{y})(P | \Pi_i [y_i \rightarrow x_i]^{\tau_i}).$$

Then we have  $\vdash \mu \bar{y} \bar{\tau} = \bar{x}.P \triangleright \bar{x} : \bar{\tau}^\dagger$ . In this agent, each output from  $P$  at  $y_i$  is mediated to its own  $x_i$ , realising recursive behaviour via a circular link.

6. Basic data types have clean representation as sequential processes. For example, the (call-by-value) pair of 2 and 3 located at  $u$ , becomes

$$\bar{u}(xy)([[2]]_x [[3]]_y),$$

which outputs through  $u$  two names, one naming the process for 2 and the other naming 3. The left-projection becomes its dual agent,

$$u(xy).\bar{w}(x)$$

which receives two names and projects the first name to  $w$ .

7. As another example of a data type, a sum type is represented by branching/selection. For example, the injection  $\text{inl}(3)$  located at  $u$  becomes

$$\bar{u}\text{inl}(x)[[3]]_x,$$

while its dual, the case construct, has the shape  $u[\&_{i \in \mathbb{B}}(y_i).P_i]$ , which has two branches and, when one is selected,  $P_i$  may interrogate at  $y_i$  for the communicated value.

## 2.2 Transition and Duality

A starting point of our study is a logic for processes pioneered by Hennessy and Milner (Hennessy and Milner 85): and how, in the typed setting, it allows a surprisingly simple presentation. 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  $z$ .

$$P \stackrel{\text{def}}{=} \bar{y}(c)c[\&_{n \in \mathbb{N}} \bar{z}\text{in}_{n*2}]. \quad (3)$$

The behaviour of  $P$  can be understood in terms of the following transition sequence:  $\bar{y}(c) \cdot c\text{in}_n \cdot \bar{z}\text{in}_{n*2}$ . Since the  $\pi$ -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  $\pi$ -calculus, we may as well be able to represent many kinds of behaviours, all using atomic name passing sequences. Next consider the following process.

$$P' \stackrel{\text{def}}{=} \bar{y}(c)c[\&_{n \in \mathbb{N}} \bar{y}(c')c'[\&_{m \in \mathbb{N}} \bar{z}\text{in}_{n+m}]. \quad (4)$$

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  $\pi$ -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 1 (2)). Then we can write the behaviour of  $P$  in the presence of that agent as follows:

$$P \models \mathbf{rely} \bar{y}(c)cin_3 \mathbf{guar} \bar{z}in_6 \quad (5)$$

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  $\bar{y}(c)cin_3$  in the rely part means the environment is prepared to return 3 how many times it is asked. Since we can argue the same holds for arbitrary  $n$ , we may conclude  $P$  and  $P'$  have indeed the same behaviour.

### 2.3 Replicated behaviour

Milner (Milner 90) introduced the replication  $!x(\bar{y}).P$  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:

$$Q \stackrel{\text{def}}{=} !x(yz).P, \quad Q' \stackrel{\text{def}}{=} !x(yz).P' \quad (6)$$

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:

$$Q \models \mathbf{rely} \mathbf{0} \mathbf{guar} x(yz)(\bar{y}(c)cin_3, \bar{z}in_6) \quad (7)$$

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  $\bar{y}(c)cin_3$  and  $\bar{z}in_6$  are not temporally related but simply juxtaposed, reflecting the previous specification (5). 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 formula:

$$Q \models \mathbf{rely} \mathbf{0} \mathbf{guar} \bigwedge_{n \in \mathbb{N}} x(yz)(\bar{y}(c)cin_n, \bar{z}in_{n*2}) \quad (8)$$

This is an infinite conjunction, saying:

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

### 2.4 Behaviours as Types

In the specifications (5) and (7), 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.

$$\begin{aligned} \bar{y}(c)c\mathbf{in}_3 &\mapsto y : (\mathbf{in}_3()^\downarrow)^\uparrow \\ \bar{z}\mathbf{in}_6 &\mapsto z : \mathbf{in}_6()^\uparrow \\ \bigwedge_{n \in \mathbb{N}} x(yz)(\bar{y}(c)c\mathbf{in}_n, \bar{z}\mathbf{in}_{n*2}) &\mapsto x : \bigwedge_{n \in \mathbb{N}} ((\mathbf{in}_n()^\downarrow)^\uparrow)^\uparrow (\mathbf{in}_{n*2}()^\uparrow)^\uparrow \end{aligned}$$

Here we put symbols ! etc. to indicate the directions and modes of actions. The structure of these simplified types 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 these types. Then we obtain, instead of (5) and (7):

$$\begin{aligned} P &\models \mathbf{rely} \ y : (\mathbf{in}_3()^\downarrow)^\uparrow \ \mathbf{guar} \ z : \mathbf{in}_6()^\uparrow, & (9) \\ Q &\models \mathbf{rely} \ \emptyset \ \mathbf{guar} \ x : \bigwedge_{n \in \mathbb{N}} ((\mathbf{in}_n()^\downarrow)^\uparrow)^\uparrow (\mathbf{in}_{n*2}()^\uparrow)^\uparrow. & (10) \end{aligned}$$

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

### 2.5 Assertions for typed processes

Since behavioural types we have seen in the previous subsection are in general infinitary entities, they are not suitable for logical reasoning and specifications (while a proof system for deriving judgements as (9) 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 it is 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 conceptual challenges to reach a suitable notion of assertions starting from behavioural types is how we can treat an infinitary entity such as  $\bigwedge_{n \in \mathbb{N}} ((\mathbf{in}_n()^\downarrow)^\uparrow)^\uparrow (\mathbf{in}_{n*2}()^\uparrow)^\uparrow$  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 types. A behavioural type as we have seen so far represents communication, and the best way to unfold it is to interact — or to test its content by interaction (Hennessy and Milner 85). For example, we may consider a testing of the form  $x(yz).\bar{z}\mathbf{in}_6$ , where we send two names to  $x$ , and we receive 6 as a return. We can represent this as:

$$x \bullet y = \mathbf{in}_6(). \quad (11)$$

In the above formula,  $x \bullet y$  and  $\mathbf{in}_6()$  are terms, 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  $y$ , is best understood as abstraction of an output of form  $\bar{x}(yc)$  (where we omit the compulsory linear name  $c$ ); while  $\mathbf{in}_6()$  can be understood as a linear

output  $\bar{c}\text{in}_6()^\dagger$  (for some  $c$ ). Thus  $x \bullet y = \text{in}_6()$  says that the invocation  $\bar{x}(yc)$  is answered by a linear output  $\bar{c}\text{in}_6$ . 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  $\text{in}_6()$  via  $z$ ”.

In the same way, the behaviour of the natural number agent  $[[3]]_y$  can be characterised by the formula “ $y \bullet \varepsilon = \text{in}_3()$ ” (where  $\varepsilon$  is the empty string). Using this, we can refine (11) by adding a constraint on  $y$ , obtaining:

$$\forall y. (y \bullet \varepsilon = \text{in}_3() \supset x \bullet y = \text{in}_6()). \quad (12)$$

We can further refine this into:

$$\forall y, i. (y \bullet \varepsilon = \text{in}_i() \supset x \bullet y = \text{in}_{i*2}()), \quad (13)$$

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

$$Q \models \mathbf{rely} \top \mathbf{guar} \forall y, i. (y \bullet \varepsilon = \text{in}_i() \supset x \bullet y = \text{in}_{i*2}()). \quad (14)$$

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:

$$Q \models \mathbf{rely} \top \mathbf{guar} \forall y, n. (x \bullet y = \text{in}_n() \supset Even(n)). \quad (15)$$

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

$$P \models \mathbf{rely} y \bullet \varepsilon = \text{in}_n() \mathbf{guar} \exists m. z = \text{in}_m() \quad (16)$$

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 judgements.

## 2.6 Partial and total correctness

So far we have blurred one aspect of our specifications for sequential processes, the distinction between *partial correctness* and *total correctness*, the idea originating in Hoare Logic. 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  $\Omega_u$  (which diverges whenever it is invoked, that is:  $\Omega_u | \bar{u}(c)P \uparrow$ ) satisfies all specifications (13), (14) and (16); while it satisfies none of them in the logic for total correctness. We may also consider the third variant, where we can discuss both convergence and divergence of processes, from which other two logics arise as special instances of this logic. The present paper focusses on total correctness, which has a simplest syntactic presentation.

### 2.7 A Program logics for PCFv

Equational logics for higher-order functions have been studied since Church's  $\lambda$ -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 PCFv, as our target calculus. We first show how the typing judgement in PCFv translates to that of the  $\pi$ -calculus.

$$y_1 : \beta_1, \dots, y_n : \beta_n \vdash M : \alpha \quad \mapsto \quad \vdash \langle\langle M \rangle\rangle_u \triangleright y_1 : \bar{\beta}_1^\circ, \dots, y_n : \bar{\beta}_n^\circ, u : (\alpha^\circ)^\dagger$$

where  $\alpha^\circ$  is given:  $\mathbb{N}^\circ \stackrel{\text{def}}{=} ([\oplus_{i \in \mathbb{N}} \varepsilon]^\dagger)^\dagger$  and  $(\alpha \Rightarrow \beta)^\circ \stackrel{\text{def}}{=} (\bar{\alpha}^\circ(\beta^\circ)^\dagger)^\dagger$ , where  $\bar{\alpha}$  is the dual of  $\alpha$  (i.e. the result of simply turning each symbol in  $\alpha$  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 PCFv is to retain this name  $u$  when asserting about the given term. This name  $u$  is called *anchor*. Thus we obtain, for example:

$$\{Even(x)\} x + 1 :_u \{Odd(u)\} \tag{17}$$

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 (17) with its ‘‘original’’  $\pi$ -calculus specification:

$$!u(c).\bar{x}(c')c'[\&_n \bar{c} \text{in}_{n+1}] \quad \models \quad \begin{array}{l} \mathbf{rely} \quad \exists n. (x \bullet \varepsilon = \text{in}_n() \wedge Even(n)) \\ \mathbf{guar} \quad \exists m. (u \bullet \varepsilon = \text{in}_m() \wedge Odd(m)) \end{array}$$

here  $u$  and  $x$  are indeed interaction points, which are ‘‘folded’’ into simple variables in the assertion (17) for PCFv.

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 (Longley 98): 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 PCFv. The first one uses a higher-order free variable, while the second one uses recursion.

$$\{\forall n. Even(y \bullet n)\} \lambda x. yx + 1 :_u \{Odd(u \bullet 3)\} \tag{18}$$

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.

$$\{\top\} \mu f. \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } x * f(x - 1) :_u \{\forall n. u \bullet n = n!\}. \tag{19}$$

The specification may need no illustration. In Section 6, we shall formally introduce the logic for PCFv and demonstrate how we can derive these assertions using compositional proof rules.

### 3 Sequential Affine Typing

#### 3.1 Preliminaries

We briefly present the sequential affine typing we shall use in the present inquiries. First, in the subsequent development, we only use types of the following shape. Below  $\vec{\tau}^p$  indicates that each type in  $\vec{\tau}$  has mode  $p$ .

$$(\vec{\tau}^? \rho^\dagger)^\dagger, (\vec{\tau}^\dagger \rho^?)^?, [\&_{i \in I} \vec{\tau}_i^?]^\dagger, [\oplus_{i \in I} \vec{\tau}_i^\dagger]^\dagger.$$

Further we assume, in  $[\&_{i \in I} \vec{\tau}_i^?]^\dagger$  and  $[\oplus_{i \in I} \vec{\tau}_i^\dagger]^\dagger$ , that  $\vec{\tau}_i = \vec{\tau}_j$  for each  $i, j \in I$  whenever  $I$  is infinite. Intuitively, a replicated input type carries zero or more  $?$ -types and a unique affine output type which intuitively stand for arguments of an invocation and a return channel. Dually for a  $?$ -type. Types under these constraints, coming from game semantics (Hyland and Ong 2000) and typed  $\pi$ -calculi, (Milner 90; Berger et al. 2001), are sufficient for the present inquiries, while making the typing rules simpler. We recall:

- Types of modes  $!$  and  $\dagger$  are *positive*.
- Types of modes  $?$  and  $\downarrow$  are *negative*.

We let  $\tau \odot \rho$  be the least partial operation which satisfies:

$$(a-1) \tau^? \odot \tau^? = \tau^?, \quad (a-2) \tau^\dagger \odot \bar{\tau}^? = \tau^\dagger, \quad (b) \tau^\dagger \odot \bar{\tau}^\dagger = \uparrow.$$

(a-1) and (a-2) together say a replicated input is ready to absorb as many dual outputs. (b) says a linear input and its dual compensate each other to become uncomposable. Note  $\odot$  is *not* commutative, unlike in (Berger et al. 2001): the rule is made so that, in (a-2) and (b) which combine two types of different polarities, the first argument is always positive. This is later used for preventing circularity.

Recall an action type is *sequential* if there is at most one name which is mapped to a  $\uparrow$ -type in that action type. We extend  $\odot$  to sequential action types, using the same symbol. First, The partial operation  $\odot$  on sequential action types is given as:

$$\begin{aligned} \Gamma \odot \Delta &= \Gamma \cup \Delta && (\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset, \Gamma \cup \Delta \text{ sequential}) \\ (\Gamma \cup x:\tau) \odot (\Delta \cdot x:\tau') &= \Theta \cdot x:\rho && (\tau \odot \tau' = \rho, \Gamma \odot \Delta = \Theta) \end{aligned}$$

In other cases  $\Gamma \odot \Delta$  is undefined. When  $\Gamma \odot \Delta$  is defined, we write  $\Gamma \times \Delta$ .  $\times$  prevents formation of new circular dependency by composition.

#### 3.2 Typing Rules

The typing system is given in Figure 1. In each rule, we assume all action types are sequential. Unlike the original system in (Berger et al. 2001), the presented rules do not use IO-modes for ensuring sequentiality, made possible by the use of sequential action types. Below we briefly illustrate the typing rules.

- (Zero) assigns the empty type to  $\mathbf{0}$ .
- (Par) composes two processes using  $\odot$ . By  $\Gamma_1 \times \Gamma_2$ , it is ensured that the resulting action type is sequential and that no new circular dependency is constructed.
- (Rec) is the rule for recursion, realised by well-typed substitution. As a simplest example, a non-circular  $[x \rightarrow w]^\tau[[w \rightarrow y]^\tau]$ , of type  $x:\tau, w:\tau, y:\tau$ , which is typable by (Par), becomes circular by (Rec), obtaining  $[x \rightarrow w]^\tau[[w \rightarrow x]^\tau]$  of type  $x:\tau, w:\tau$ ,

(Zero)	(Par) $(\Gamma_1 \times \Gamma_2)$	(Rec)	(Res)	(Weak)
$-$	$\vdash P_i \triangleright \Gamma_i \ (i=1,2)$	$\vdash P \triangleright \Gamma, \vec{y}:\vec{\tau}, \vec{x}:\vec{\tau}^\dagger$	$\vdash P \triangleright \Gamma, x:\tau^\dagger, \downarrow$	$\vdash P \triangleright \Gamma^{-x}$
$\vdash \mathbf{0} \triangleright -$	$\vdash P_1   P_2 \triangleright \Gamma_1 \odot \Gamma_2$	$\vdash P[\vec{x}/\vec{y}] \triangleright \Gamma, \vec{x}:\vec{\tau}$	$\vdash (v x)P \triangleright \Gamma$	$\vdash P \triangleright \Gamma, x:\tau^\dagger, \downarrow$
(Bra <sup>↓</sup> )	(Sel <sup>↑</sup> )			
$\forall i. \vdash P_i \triangleright ?\Gamma^{-x}, \vec{y}_i:\vec{\tau}_i, v:\rho^\dagger$	$\vdash P \triangleright ?\Gamma, \vec{y}:\vec{\tau}_i$			
$\vdash x[\&_i(\vec{y}_i).P_i] \triangleright \Gamma, x: [\&_i \vec{\tau}_i]^\dagger, v:\rho$	$\vdash \bar{x} \text{in}_i(\vec{y})P \triangleright \Gamma, x: [\oplus_{i \in I} \vec{\tau}_i]^\dagger$			
(In <sup>↑</sup> )	(Out <sup>?</sup> ) $(\Gamma(x) = (\vec{\tau}\rho)^\dagger)$			
$\vdash P \triangleright ?\Gamma^{-x}, \vec{y}:\vec{\tau}, z:\rho$	$\vdash P \triangleright ?\Gamma, \vec{y}:\vec{\tau}, z:\rho', v:\rho^\dagger$			
$\vdash !x(\vec{y}z).P \triangleright \Gamma, x:(\vec{\tau}\rho)^\dagger$	$\vdash \bar{x}(\vec{y}z)P \triangleright \Gamma, v:\rho$			

Fig. 1. Sequential Affine Typing

- (Res) hides a channel of mode ! and  $\downarrow$  (thus channels of mode ?,  $\downarrow$  and  $\uparrow$  need their duals before hiding).
- (Weak) weakens ?- as well as  $\uparrow$ -channels.
- (Bra) says that if, for each  $i$ , a process is typed as specified, then we can prefix it with a branching input (note each type in  $\vec{\tau}_i$  has mode ? by well-formedness).
- (Sel) says that, if there is a process with !-channels and ?-channels, then we can construct a linear selection output which carries those !-channels.
- (In) says that if a process has a specified action type, then we can abstract  $\vec{y}$  (of type  $\vec{\tau}$ , which are ?-types) and  $z$  (of type  $\rho$ , which is a  $\uparrow$ -type) to construct a replicated input, which suppresses free outputs  $\Gamma$ . Note  $\Gamma$  has mode ?, saying only ?-channels are suppressed under replication.
- (Out) is similar to (Sel), except the ?-type at  $x$  should already appear in the antecedent and there is no selection.

Processes typable with the typing rules in Figure 1 are a subset of affine processes in (Berger et al. 2001) but are enough to allow fully abstract embeddings of both call-by-value and call-by-name PCF with products and sums. For the proof of the following result, see Appendix A.

**Proposition 1** *If  $\vdash P \triangleright \Gamma$  and  $P \longrightarrow Q$  then for some  $Q'$  we have  $Q' \equiv Q$  and  $\vdash Q' \triangleright A$ .*

In the typing system in Figure 1, we used a substitution for representing recursion, which gives the correspondence with affine processes in (Berger et al. 2001). An alternative presentation of recursion  $\mu\vec{y} = \vec{x}.P$  given in Example 1 (5) can be typed as follows.

$$\text{(Rec-}\mu\text{)} \frac{\vdash P \triangleright \Gamma, \vec{y}:\vec{\tau}, \vec{x}:\vec{\tau}^\dagger}{\vdash \mu\vec{y} = \vec{x}.P \triangleright \Gamma, \vec{x}:\vec{\tau}} \quad (20)$$

Syntactically, (Rec- $\mu$ ) is derivable from (Rec), (Par) and the typing for copycats (cf. Example 1 (3)). Semantically, however, the rule is equipotent to (Rec) since  $P[\vec{y}/\vec{x}]$  and  $\mu\vec{y} = \vec{x}.P$  in the rule are behaviourally equivalent (in e.g. the standard weak bisimilarity).

## 4 Process Logic

### 4.1 Terms, Formulae and Judgement

We formally introduce the syntax of our logical language. Terms in the logic are given by the following grammar. Below in the second line we let  $\tau$  be of mode  $!$  or  $\uparrow$ , and  $\rho$  be of mode  $\uparrow$ .  $\mathbf{n}$  ranges over the set of numerals.

$$\begin{aligned} e & ::= * \mid \mathbf{t} \mid \mathbf{f} \mid \mathbf{n} \mid x^{\text{nat}} \mid x^{\text{bool}} \mid \text{op}(\vec{e}) \mid \dots \\ a & ::= x^\tau \mid a \bullet \vec{b} \mid \text{in}_e^\rho(\vec{a}) . \end{aligned}$$

The first set of terms ( $e, e', \dots$ ) are called *data expressions*, while the second set ( $a, a', b, b', \dots$ ) *behaviour expressions*. Data expressions are typed with atomic data types, ranged over by  $T, T', \dots$ , which we identify with sets of constants of that type. In the present inquiry, we only consider the following three.

- A distinguished singleton  $\{*\}$  (which may be considered as the unit type).
- The boolean type  $\mathbb{B}$ , inhabited by  $\mathbf{t}$  and  $\mathbf{f}$ .
- The natural number type  $\mathbb{N}$ , inhabited by non-negative integers.

The last construct in the first line,  $\text{op}(\vec{e})$ , is a first-order total computable operator which take a vector of data expressions as arguments. We consider, among others, the standard numeric operations (addition, multiplication, etc.) and the standard boolean operations.

Intuitively, data expressions denote elements of atomic types, while behaviour expressions abstract process interactions. The well-typedness of data/behaviour expressions is naturally given. For data expressions this is standard (for each operator we assume a fixed signature). For behaviour expressions, we say *a has type  $\tau$*  by the following induction.

- (1)  $x^\tau$  has type  $\tau$ ;
- (2)  $a \bullet \vec{b}$  has type  $\tau^\uparrow$  iff  $a$  has type  $(\vec{\rho}\tau)^\uparrow$  and  $\vec{b}$  has type  $\vec{\rho}$ ;
- (3)  $\text{in}_e^\rho(\vec{a})$  has type  $\rho = [\oplus_{i \in T} \vec{\tau}_i]^\uparrow$  iff  $e$  has type  $T$  and that, under each valuation  $\xi$ , if  $e\xi = i \in T$  then  $\vec{a}$  has type  $\vec{\tau}_i$ .

In (3), a *valuation*  $\xi$  is a well-typed map from names to elements: whereas the expression  $e\xi$  gives the obvious evaluation of  $e$  under  $\xi$ .<sup>2</sup> In (2) and (3), we say  $a_1..a_n$  has type  $\tau_1..\tau_n$  if each  $a_i$  has type  $\tau_i$ .

Note each well-typed term has a unique type. Further it is linearly derivable whether a term is typable or not and, if it is, what is its type. Some conventions:

- We often omit types from variables.
- Similarly we usually omit  $\rho$  from  $\text{in}_e^\rho(\vec{a})$ , writing  $\text{in}_e(\vec{a})$ .
- We write  $(\vec{a})^\uparrow$  for  $\text{in}_*(\vec{a})$  for brevity.

Note the third convention corresponds to the notation  $\bar{x}(\vec{y})P$  for a linear selection with the selection index  $*$ .

<sup>2</sup> Since summands in a branching/selection type are identical with each other when the indexing set is  $\mathbb{N}$  (cf. §3.1), the use of valuation in condition (3) does not jeopardise feasibility of type checking of terms.

Formulae are those of the standard first-order logic with equality (Menselson 87). Below  $\star$  ranges over  $\{\wedge, \vee, \supset\}$  and  $\mathcal{Q}$  over  $\{\forall, \exists\}$ .

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

We also use the truth  $\top$  (which is, say,  $1 = 1$ ) and the falsity  $\perp$  (which is  $\neg\top$ ). The quantifications induce binding, for which we assume the standard bound name convention. A substitution  $A[e/x]$  is standard.  $\text{fn}(A)$  denotes free names in  $A$ . Formulae are often called *assertions*. A formula  $A$  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 write  $\Gamma \vdash A$  iff names in  $A$  are given types as they are by  $\Gamma$ . *We only consider well-typed formulae from now on.*

A judgement in the process logic has the following shape (we give one for validity: for provability we simply replace  $\models$  with  $\vdash$ , similarly for the subsequent definitions).

$$P^\Gamma \models \mathbf{rely} A \mathbf{guar} B \tag{21}$$

In (21), the formula  $A$  (resp.  $B$ ) is the *rely formula* (resp. the *guarantee formula*) of the judgement. Those names in  $A$  and  $B$  which are from  $\Gamma$  are called *primary names*, others *auxiliary names*. A judgement is sometimes written  $P \models \mathbf{rely} A \mathbf{guar} B$ , leaving the action type implicit. In the subsequent development, well-typedness of judgements is important, which is stipulated as follows.

**Definition 1** (well-typedness of judgement) Let  $\Gamma = \Delta_1, \Delta_2$  such that types in  $\Delta_1$  (resp.  $\Delta_2$ ) are negative (resp. positive/neutral). Then  $P^\Gamma \models \mathbf{rely} A \mathbf{guar} B$  is *well-typed* if the following holds for an action type  $\Theta$  whose domain is disjoint from  $\Delta_{1,2}$ .

- $\vdash P \triangleright \Gamma$ .
- $\overline{\Delta_1}, \Theta \vdash A$ .
- $\overline{\Delta_1}, \Delta_2, \Theta \vdash B$ .

where  $\overline{\Delta_1}$  denotes the result of dualising each type in  $\Delta_1$ .

**Remark 1** Note  $\Theta$  in Definition 1 type auxiliary names. We may as well omit  $\overline{\Delta_1}$  from the typing for  $B$ , as is done in (Honda 2004c). This does not lead to a loss of generality, even though having negative names in the conclusion is practically convenient.

**Convention 1** *Henceforth we only consider well-typed judgements.*

Note free names in formulae in a well-typed judgement are always strictly typed by the underlying action type. Also note neutral names, i.e. those typed by  $\downarrow$ , never occur in formulae since a name in a formula is always typed with either a positive type or a negative type. This indicates logical insignificance of neutral names (which reflects their behavioural insignificance: no visible action is possible at neutral names).

For a judgement in an abbreviated form,  $P \models \mathbf{rely} A \mathbf{guar} B$ , we informally assume well-typedness under an omitted action type for  $P$ .<sup>3</sup>

<sup>3</sup> As a reference we note this convention in fact does not lose precision in the following sense: under the conditions stipulated for channel types in Section 3, each typable process  $P$  can be given a unique minimum action type, and because the validity of judgements is preserved by weakening, so that we can always consider the minimum action type for  $P$  in  $P \models \mathbf{rely} A \mathbf{guar} B$  when considering typability.

## 4.2 Models

In the following we present a semantics of assertions and judgements. To present key ideas in a simplest setting, we use typed processes modulo the contextual congruence to construct models (we can also use, for example, CPOs). We first define a contextual congruence for sequential affine processes (Berger et al. 2001), denoted  $\cong_\pi$ . First, a *typed relation* is a relation over typed processes which only relate processes with the same action type. For a typed relation  $\mathcal{R}$ , we write  $\vdash P_1 \mathcal{R} P_2 \triangleright \Gamma$  when  $\vdash P_1 \triangleright \Gamma$  and  $\vdash P_2 \triangleright \Gamma$  are related by  $\mathcal{R}$ . A *typed congruence* is a typed equivalence relation which is closed under the sequential affine typing rules (cf. §3.2). Then  $\cong_\pi$  is the maximum typed congruence which satisfies the following condition:

$$\vdash P \cong_\pi Q \triangleright x : ()^\dagger \quad \supset \quad (P \Downarrow_x \equiv Q \Downarrow_x).$$

where  $P \Downarrow_x$  indicates  $P \longrightarrow^* (\bar{x}(\bar{y})P' | Q')$  for some  $\bar{y}$ ,  $P'$  and  $Q'$  (in the above case  $\bar{y}$  is empty). We have shown in (Berger et al. 2001) that  $\cong_\pi$  is maximally consistent, i.e. if we try to non-trivially extend this equality then it leads to the universal typed relation.

Models are constructed using processes modulo  $\cong_\pi$ .

**Definition 2** (model) Let  $\tau$  be positive. An *abstract process*  $p$  of type  $\tau$  is a map from the names to the  $\cong_\pi$ -congruence classes such that:

- (1)  $\vdash P \triangleright \Gamma \in p(x)$  implies  $\Gamma = x : \tau$ ; and
- (2)  $p$  is closed under injective renaming, i.e.  $p(x) = p(y) \binom{xy}{yx}$  for each  $x, y$ , where  $\binom{xy}{yx}$  is a name permutation acting on processes as an injective substitution.

A *model*  $\xi$  is a finite map from names to abstract processes. A model  $\xi$  is *total* if  $x : p \in \xi$  implies  $\neg p \uparrow$ , where  $p \uparrow$  if a process (hence every process) in  $p$  has infinite reductions. We say  $\xi$  *has type*  $\Gamma$  iff, for each  $x \in \text{dom}(\xi)$ , we have  $\Gamma(x) = \tau$  iff  $\xi(x)$  has type  $\tau$ .

**Remark 2** The renaming closure in Definition 2 (2) abstracts away concrete names from behaviours (such abstraction is used/studied in (Milner 92; Honda 2000)). Using processes with a single name suffices since, under the sequential affine typing, the meaning of each process can be represented by “closing” its negatively typed names with dual processes.

There are operations over abstract processes which correspond to those on behaviour expressions. Assume  $p$  has type  $(\tau_1.. \tau_n \rho^\dagger)^\dagger$  and  $q_i$  has type  $\bar{\tau}_i$  for each  $i$ . Then we define  $p \bullet \vec{q}$  as the abstract process of type  $\rho$  such that, for each  $z$  and omitting types:

$$(p \bullet \vec{q})(z) = \{(\nu x \bar{y})(P \mid \bar{x}(\bar{y}z')(\Pi Q_i \mid [z' \rightarrow z]^{\bar{\rho}})) \mid P \in p(x), Q_i \in q_i(y_i)\} \quad (22)$$

where  $x\bar{y}z'$  are pairwise distinct (since we are identifying processes up to the  $\alpha$ -equality, concrete choice of  $x\bar{y}z'$  does not matter). Intuitively,  $p \bullet \vec{q}(z)$  sends to  $p(x)$  a message carrying names representing  $\vec{q}$ , and mediates the result, if any, to  $z$ . Similarly, with each  $p_i$  of type  $\tau_i$ , we define  $\text{in}_j(p_1..p_n)$  as the abstract process such that, for each  $z$ :

$$(\text{in}_j(p_1..p_n))(z) = \{\bar{z} \text{in}_j(\bar{y}) \Pi_i P_i \mid P_i \in p_i(y_i)\} \quad (23)$$

where  $\bar{y}$  are pairwise distinct. Intuitively,  $\text{in}_j(p_1..p_n)(z)$  is a linear selection with index  $j$  which outputs a vector of names which represent  $p_1..p_n$ .

### 4.3 Satisfaction and Semantics of Judgement

We can now interpret terms in the sequential process logic using abstract processes and operations on them. Let  $\mathcal{J}$  be a model which maps typed names occurring in a term to constants and abstract processes, respecting types. For data expressions, the interpretation  $\llbracket e \rrbracket I$  is standard, starting from  $\llbracket x \rrbracket I = I(x)$ . For behaviour expressions, the interpretation  $\llbracket a \rrbracket I \cdot \xi$  is given by the following induction.

- $\llbracket x \rrbracket I \cdot \xi = (I \cup \xi)(x)$ ,
- $\llbracket x \bullet y_1 \dots y_n \rrbracket I \cdot \xi = p \bullet q_1 \dots q_n$  where  $p = \llbracket x \rrbracket I \cdot \xi$  and  $q_i = \llbracket y_i \rrbracket I \cdot \xi$ .
- $\llbracket \text{in}_e(x_1 \dots x_n) \rrbracket I \cdot \xi = \text{in}_{\llbracket e \rrbracket I}(p_1 \dots p_n)$  where  $p_i = \llbracket x_i \rrbracket I \cdot \xi$ .

The notion of satisfaction of  $A$  by a model  $\xi$  under an interpretation  $I$ , written  $\xi \models^I A$ , is given by induction on the structure of  $A$ . We start from:

$$\begin{aligned} \xi \models^I e_1 = e_2 &\equiv \llbracket e_1 \rrbracket I = \llbracket e_2 \rrbracket I. \\ \xi \models^I a_1 = a_2 &\equiv \llbracket a_1 \rrbracket I \cdot \xi = \llbracket a_2 \rrbracket I \cdot \xi \end{aligned}$$

Logical connectives are interpreted classically:

$$\begin{aligned} \xi \models^I A_1 \wedge A_2 &\equiv (\xi \models^I A_1) \wedge (\xi \models^I A_2) \\ \xi \models^I \neg A &\equiv \neg (\xi \models^I A) \\ \xi \models^I \forall x^T. A &\equiv \forall c \in T. \xi \models^{I \cdot xc} A, \\ \xi \models^I \forall x^\tau. A &\equiv \forall p \in \llbracket \tau \rrbracket. \xi \models^{I \cdot xp} A, \end{aligned}$$

where, in the last line,  $\llbracket \tau \rrbracket$  denotes the set of all abstract processes of type  $\tau$ . The rest is de Morgan duality.

We also need the notion of satisfaction of  $A$  by a process  $P$  under an interpretation  $I$ , written  $P^\Gamma \models^I A$ , given as follows. Below let  $\vdash P \triangleright \Gamma$ .

$$P^\Gamma \models^I A \stackrel{\text{def}}{=} \exists \xi. (P^\Gamma \models \xi \wedge \xi \models^I A)$$

Above  $P^\Gamma \models \xi$  denotes  $\Gamma \vdash \xi$ .  $\xi = x_1 : p_1, \dots, x_n : p_n$  and  $\vdash P \cong_\pi \prod_i P_i \Gamma$  such that  $P_i \in p_i(x_i)$  for each  $i$ . We can now formally define the semantics of judgements.

**Definition 3** Let  $\vdash P \triangleright \Gamma, \Delta$  where  $\Gamma$  is negative and  $\Delta$  is positive/neutral. Then  $P^{\Gamma, \Delta} \models \mathbf{rely} A \mathbf{guar} B$  when, for each  $I$  and  $\vdash R \triangleright \bar{\Gamma}$ ,  $R \models^I A$  implies  $P|R \models^I B$ .

**Remark 3** While we do not use in the present inquiry, we list a notable property of the sequential process logic for reference, which says that the logic enjoys a precise correspondence with observable behaviours of processes, in the sense that assertions valid for each process precisely characterise its behaviour up to the contextual congruence. Below  $\vdash P =_{\mathcal{L}} Q \triangleright \Gamma$  stands for  $\forall A, B. (P^\Gamma \models \mathbf{rely} A \mathbf{guar} B \equiv Q^\Gamma \models \mathbf{rely} A \mathbf{guar} B)$ .

**Proposition.**  $\vdash P =_{\mathcal{L}} Q \triangleright \Gamma$  iff  $\vdash P \cong_\pi Q \triangleright \Gamma$ .

For the proof, the “if” direction is immediate, while the “only if” direction uses representation of a finite process (context) by a formula. A full technical development on the correspondence between observability, validity and provability will be presented in the sequels to the present paper.

$\frac{}{\mathbf{0} \vdash \mathbf{rely} A \mathbf{guar} A}$	$\frac{\text{(Par)} \quad \begin{array}{l} P_1 \vdash \mathbf{rely} A \mathbf{guar} C \\ P_2 \vdash \mathbf{rely} C \mathbf{guar} B \end{array}}{P_1   P_2 \vdash \mathbf{rely} A \mathbf{guar} B}$	$\frac{\text{(Rec)} \quad \begin{array}{l} P \vdash \mathbf{rely} A^{-\bar{y}} \wedge \forall j \leq i. B(j)[\bar{y}/\bar{x}] \mathbf{guar} B(i)^{-\bar{y}} \end{array}}{P[\bar{x}/\bar{y}] \vdash \mathbf{rely} A \mathbf{guar} \forall i. B(i)}$
$\frac{\text{(Res)} \quad P \vdash \mathbf{rely} A \mathbf{guar} B^{-x}}{(\forall x)P \vdash \mathbf{rely} A \mathbf{guar} B}$	$\frac{\text{(Weak)} \quad P^\Gamma \vdash \mathbf{rely} A \mathbf{guar} B}{P^{\Gamma, x:\tau} \vdash \mathbf{rely} A \mathbf{guar} B}$	
$\frac{\text{(Bra}^\downarrow) \quad \forall i. P_i \vdash \mathbf{rely} A[\text{in}_i(\bar{y}_i)/x] \mathbf{guar} B}{x[\&_i(\bar{y}_i).P_i] \vdash \mathbf{rely} A \mathbf{guar} B}$	$\frac{\text{(Sel}^\uparrow) \quad P \vdash \mathbf{rely} A \mathbf{guar} B[\text{in}_i(\bar{y})/x]}{\bar{x} \text{in}_i(\bar{y})P \vdash \mathbf{rely} A \mathbf{guar} B}$	
$\frac{\text{(In}^\downarrow) \quad P \vdash \mathbf{rely} A^{-\bar{y}} \wedge B_1^{\bar{y}} \mathbf{guar} B_2}{!x(\bar{y}z).P \vdash \mathbf{rely} A \mathbf{guar} B_1 \supset B_2[x \bullet \bar{y}/z]}$	$\frac{\text{(Out}^\uparrow) \quad P \vdash \mathbf{rely} A^{-z} \wedge C^z \mathbf{guar} B^{-\bar{y}z} \wedge C[x \bullet \bar{y}/z] \wedge x \bullet \bar{y} \Downarrow}{\bar{x}(\bar{y}z)P \vdash \mathbf{rely} A \mathbf{guar} B}$	

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

#### 4.4 Proof Rules for Process Logic

One of the significant consequences of the representation of environments in judgements in the present logic is the existence of simple compositional proof rules for valid judgements. We use the judgement of the form  $P^\Gamma \vdash \mathbf{rely} A \mathbf{guar} B$  for provability, which should follow the same typing constraint on primary/auxiliary names in formulae as given in the previous subsection. As before, action types are often omitted. The proof rules are given in Figures 2 and 3. We assume the following conventions.

- In each rule, all occurring judgements, including the conclusion, are well-typed.
- $A^{-\bar{y}}$  indicates no name in  $\bar{y}$  occurs in  $A$ .
- $A^{\bar{y}}$  indicates all *primary* names in  $A$  are in  $\{\bar{y}\}$ .
- $i, j, \dots$  exclusively range over auxiliary names (in a given judgement).
- In each rule, no auxiliary names in a judgement in its premise should overlap with primary names in a judgement in its conclusion.

There are two kinds of proof rules. The *main rules*, given in Figure 1, follow the typing of processes in Figure 1: for these rules, we assume each rule is applied following a type derivation of a given process. The *structural rules*, given in Figure 3, do not change the shape of a typed process (including its action type) and only manipulate formulae.

Among the main rules, (Zero) should need no illustration. (Res) is reminiscent of Hoare's rule for local name (Hoare and Wirth 73). (Weak) only weakens the action type.

(Par) rule hinges on the non-circular parallel composition in Figure 1. As a result, we obtain the simple proof rule given above, close to the cut rule in the sequent calculus. In the premise, if a free name in  $E$  is primary in the first sequent, then it should also be so in the second sequent, because of our convention on name usage noted above.

(Rec) combines mathematical induction and recursive behaviour, ensuring total correctness. In the rule,  $A(e)$  denotes the result of substituting  $e$  for a fresh name occurring in  $A$ , avoiding name capture. The rule is close to Harel's rule for the while command (Harel 80), which is known to be complete for strong models (Apt 81). For tractable reasoning, one may extend the rule to well-founded induction (Floyd 67). Following the derived typing rule (Rec- $\mu$ ) in (20), §3.2, Page 18, we may also consider the following proof rule.

$$\text{(Rec-}\mu\text{)} \frac{P^{\Gamma, \bar{x}:\bar{\tau}^!, \bar{y}:\bar{\tau}^!} \vdash \mathbf{rely} A \wedge \forall j \leq i. B(j)[\bar{y}/\bar{x}] \mathbf{guar} B(i)}{\mu\bar{y} = \bar{x}. P \vdash \mathbf{rely} A \mathbf{guar} \forall i. B(i)} \quad (24)$$

As may be expected, (Rec- $\mu$ ) is derivable from (Rec), combined with (Par) and the copycat law  $[x \rightarrow y]^{\tau^!} \vdash \mathbf{rely} A[x/y] \mathbf{guar} A$ , which will be discussed later. Semantically, however, it is equipotent to (Rec) because we have  $\vdash \mu\bar{y} = \bar{x}. P \cong_{\pi} P[\bar{x}/\bar{y}] \triangleright \bar{x} : \bar{\tau}^!, \Gamma$ . The rule (Rec- $\mu$ ) is useful since it directly corresponds to a natural encoding of recursion in  $\lambda$ -calculi and, indeed, in programming languages in general (cf. (Hasegawa 99)).

(Bra $^{\downarrow}$ ) says that: *if  $x[\&_i(\bar{y}_i).P_i]$  is to guarantee  $B$  relying on  $A$ , each branch  $P_i$  should guarantee  $B$  relying on  $A$  in which  $x$  is specialised into its  $i$ -th branch by substitution.*

(Sel $^{\uparrow}$ ) is the dual of (Bra $^{\downarrow}$ ), saying: *for an output  $\bar{x}\text{in}_i(\bar{y})P$  to guarantee  $B$  relying on  $A$ ,  $P$  itself should guarantee  $B$  relying on  $A$ , with  $x$  in  $B$  replaced by its carried value.*

(In $^{\uparrow}$ ) and (Out $^{\uparrow}$ ) are another pair of mutually dual proof rules. For (In $^{\uparrow}$ ), first note, by the typing, there are at least the following primary names in the premise:

- $z$  as the (unique) positive primary name;
- $\bar{y}$  as negative primary names;

Note also, by our convention,  $x$ , which is primary in the conclusion, cannot occur as auxiliary in the premise; nor can it occur as primary there, because of the typing constraint. Also note  $\bar{y}$  in  $B_{1,2}$  (if any) become auxiliary in the conclusion, so that, by (Aux- $\forall$ ) discussed later, they can later be universally quantified. The rule reads: *Suppose  $P$  guarantees  $B_2$  at  $z$  relying on  $A$  for primary names other than  $\bar{y}$  and on  $B_1$  for  $\bar{y}$ . Then  $!x(\bar{y}z).P$  guarantees, if  $\bar{y}$  satisfies  $B_1$ , the same  $B_2$  in which  $z$  is replaced by  $x \bullet \bar{y}$ .* Note  $z$  in the conclusion arises as the result of invoking  $!x(\bar{y}z).P$  through  $x$ .

In (Out $^{\uparrow}$ ), we note, recalling  $C^z$  indicates  $z$  exhausts all primary names in  $C$ , that:

- $x\bar{y}$  exhaust all primary names in  $C[x \bullet \bar{y}/z]$  (if any).
- $P$  must have a unique positive name, say  $v$ , which should be a linear output by typing, cf. Figure 1, §3.2, Page 18. Since  $B$  does not mention  $\bar{y}$ , if ever it mentions a primary name, it should be  $v$ .
- $A$  does not mention  $z$  but may as well mention  $x$ .

Under this understanding, the rule says: *Suppose that, under the assumption that the environment guarantees  $A$  for negative names except  $z$ , and  $C$  for  $z$ ,  $P$  guarantees  $B$  and  $C[x \bullet \bar{y}/z]$  and, moreover, we know invoking the environment at  $x$  with arguments  $\bar{y}$  converges. Then, assuming only  $A$ , the process  $\bar{x}(\bar{y}z)P$  guarantees  $B$ .* The rule relies on the property that what the process may receive through  $z$  has no causal connection with its behaviours at  $\bar{y}$  nor with the environment behaviour at  $x$ .

Figure 3 lists structural rules (which are those rules which do not involve change of the syntactic structure of a process). The first rule, (Consequence), is similar to the corresponding rule in Hoare Logic, except that the entailment  $\supset$  in the present logic is not only about

<p>(Consequence)</p> $\frac{A \supset A' \quad P \vdash \mathbf{rely} A' \mathbf{guar} B' \quad B' \supset B}{P \vdash \mathbf{rely} A \mathbf{guar} B}$	<p>(<math>\wedge - \supset</math>)</p> $\frac{P^{\Delta} \vdash \mathbf{rely} A \wedge B \mathbf{guar} C}{P \vdash \mathbf{rely} A \mathbf{guar} B \supset C}$	<p>(<math>\supset - \wedge</math>)</p> $\frac{P^{\Gamma, \Delta} \vdash \mathbf{rely} B \mathbf{guar} A^{\Delta} \supset C}{P^{\Gamma, \Delta} \vdash \mathbf{rely} A \wedge B \mathbf{guar} C}$
<p>(Conj)</p> $\frac{P^{\Gamma} \vdash \mathbf{rely} A \mathbf{guar} B_i \quad (i = 1, 2)}{P^{\Gamma} \vdash \mathbf{rely} A \mathbf{guar} B_1 \wedge B_2}$	<p>(Disj)</p> $\frac{P^{\Gamma} \vdash \mathbf{rely} A_i \mathbf{guar} B \quad (i = 1, 2)}{P^{\Gamma} \vdash \mathbf{rely} A_1 \vee A_2 \mathbf{guar} B}$	
<p>(Inv-<math>\wedge</math>)</p> $\frac{P^{\Gamma} \vdash \mathbf{rely} A \mathbf{guar} B}{P^{\Gamma} \vdash \mathbf{rely} A \wedge C \mathbf{guar} B \wedge C}$	<p>(Inv-<math>\vee</math>)</p> $\frac{P^{\Gamma} \vdash \mathbf{rely} A \mathbf{guar} B}{P^{\Gamma} \vdash \mathbf{rely} A \vee C \mathbf{guar} B \vee C}$	
<p>(Aux-<math>\forall</math>)</p> $\frac{P \vdash \mathbf{rely} A^{-i} \mathbf{guar} B}{P \vdash \mathbf{rely} A \mathbf{guar} \forall i. B}$	<p>(Aux-<math>\exists</math>)</p> $\frac{P \vdash \mathbf{rely} A \mathbf{guar} B^{-i}}{P \vdash \mathbf{rely} \exists i. A \mathbf{guar} B}$	

Fig. 3. Structural Rules

number-theoretic facts but also about the universe of sequential behaviour. Apart from the standard axioms and theorems from the first-order logic with equalities, as well as those for the formal number theory, we may use the following axioms for behaviour expressions.

$$\begin{array}{ll} \text{(extensionality)} & \forall \vec{x}. u \bullet \vec{x} = v \bullet \vec{x} \quad \supset \quad u = v. \\ \text{(data)} & \mathbf{in}_e(\vec{x}) = \mathbf{in}_{e'}(\vec{y}) \quad \supset \quad e = e' \wedge \vec{x} = \vec{y}. \end{array}$$

The next rule, ( $\wedge - \supset$ ), assumes  $P$  is typed with replicated action type. Without this condition, the rule is not sound when, among others,  $B$  is falsity. ( $\supset - \wedge$ ) is its converse. (Conj) (resp. (Disj)) combines the guarantee (resp. rely) conditions of two judgements for the same process. (Inv- $\wedge$ ) and (Inv- $\vee$ ) add assertions on negative names to both the rely and guarantee formulae. The rules emphasise typing, which prevents  $C$  from mentioning positive names.

Finally (Aux- $\forall$ ) and (Aux- $\exists$ ) are immediate consequences of the semantics of auxiliary names, which are universally bound in the whole sequent (and, for (Aux- $\exists$ ), observing that the rely condition is interpreted contravariantly). By combining (Aux- $\forall$ ) with ( $\mathbf{In}^!$ ) we can derive the following more convenient version of ( $\mathbf{In}^!$ ).

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

A basic result on the semantics of these rules follows. The proof is by easy rule induction (for the proof rule for recursion, we use the standard unfolding), see (Honda 2004b).

**Theorem 1** (soundness)

For each  $\vdash P \triangleright \Gamma$ , if  $P^{\Gamma} \vdash \mathbf{rely} A \mathbf{guar} B$  then  $P^{\Gamma} \models \mathbf{rely} A \mathbf{guar} B$ .

#### 4.5 A Simple Reasoning Example

As an example of reasoning, we show a derivation of the judgement presented in Section 2. As a further example of reasoning, the next subsection will establish a non-trivial property of copycats. Let  $R \stackrel{\text{def}}{=} u[\&_n \bar{w} \text{in}_{2^*n}]$  below.

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

In Line 2, we used the reflexivity of equality. Lines 3–6 are direct from the rules. Line 6 uses  $y \bullet \varepsilon = \text{in}_n() \supset \exists n.(y \bullet \varepsilon = \text{in}_n()) \stackrel{\text{def}}{=} y \bullet \varepsilon \Downarrow$ .

#### 4.6 Copycat Laws

The purpose of this section is to prove a basic property of copy-cats. 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, *link agent* in CCS/ $\pi$ -calculus (Honda and Yoshida 95; Milner 80; Sangiorgi 96), 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  $\lambda$ -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).

**Lemma 1** (copycat laws) *Assuming well-typedness, for each  $\tau$ , we have:*

$$[x \rightarrow y]^{\tau^!} \vdash \mathbf{rely} A[y/x] \mathbf{guar} A \quad (25)$$

$$A \Downarrow_x \supset [x \rightarrow y]^{\tau^!} \vdash \mathbf{rely} A[x/y] \mathbf{guar} A \quad (26)$$

where we write  $A \Downarrow_x$  for  $A \supset x \Downarrow$ .

For the proof we simultaneously prove:

$$A \Downarrow_z \supset \bar{x}(\bar{y}z)^{\bar{p}^\tau} \vdash \mathbf{rely} A[x \bullet \bar{y}/z] \mathbf{guar} A \quad (27)$$

$$\bar{x} \text{in}_i(\bar{y})^{\bar{p}} \vdash \mathbf{rely} A[\text{in}_i(\bar{y})/x] \mathbf{guar} A \quad (28)$$

where we recall:

$$\begin{aligned} \bar{x}'\langle\bar{y}z\rangle^{\bar{\tau}p} &\stackrel{\text{def}}{=} \bar{x}'(\bar{y}'z')(\Pi_i[y'_i \rightarrow y_i]^{\bar{\tau}_i} \mid [z' \rightarrow z]^{\bar{p}}) \\ \bar{x}'\text{in}_i\langle\bar{y}\rangle^{\bar{\tau}} &\stackrel{\text{def}}{=} \bar{x}'\text{in}_i(\bar{y}')\Pi_i[y'_i \rightarrow y_i]^{\bar{\tau}_i}. \end{aligned}$$

Note  $[x \rightarrow x']^{\bar{\tau}p} \stackrel{\text{def}}{=} !x(\bar{y}z).\bar{x}'\langle\bar{y}z\rangle^{\bar{\tau}p}$  while  $[x \rightarrow x']^{[\&_i\bar{\tau}_i]^{\downarrow}} \stackrel{\text{def}}{=} x[\&_i(\bar{y}_i).\bar{x}'\text{in}_i\langle\bar{y}_i\rangle^{\bar{\tau}_i}]$ . We use the following, easily inferred from (25), writing  $[\bar{x} \rightarrow \bar{y}]$  for  $\Pi_i[x_i \rightarrow y_i]$ .

$$[\bar{x} \rightarrow \bar{y}]^{\bar{\tau}^{\downarrow}} \vdash \mathbf{rely} A[\bar{y}/\bar{x}] \mathbf{guar} A \quad (29)$$

In the following we always assume typability. We first infer (27) from (29) and (26). Assume  $A \Downarrow_z$  below and let  $C \stackrel{\text{def}}{=} A[x \bullet \bar{y}'/z] \wedge A[x \bullet \bar{y}/z] \wedge A[z'/z]$ .

1.  $[\bar{y}' \rightarrow \bar{y}] \vdash \mathbf{rely} A[x \bullet \bar{y}'/z] \wedge A[z'/z] \mathbf{guar} C \quad ((29), \text{Inv}, \text{Conseq})$

---

2.  $[z' \rightarrow z] \vdash \mathbf{rely} C \mathbf{guar} A \quad (26)$

---

3.  $[\bar{y}' \rightarrow \bar{y}] \mid [z' \rightarrow z] \vdash \mathbf{rely} A[x \bullet \bar{y}'/z] \wedge A[z'/z] \mathbf{guar} A \quad (26)$

---

4.  $\bar{x}(\bar{y}'z')([\bar{y}' \rightarrow \bar{y}] \mid [z' \rightarrow z]) \vdash \mathbf{rely} A[x \bullet \bar{y}/z] \mathbf{guar} A \quad (1,2,\text{Out})$

To derive (28) from (29), we infer:

1.  $[\bar{y}' \rightarrow \bar{y}] \vdash \mathbf{rely} A[\text{in}_i(\bar{y})/x] \mathbf{guar} A[\text{in}_i(\bar{y}')/x] \quad (\text{from } 29)$
2.  $\bar{x}\text{in}_i(\bar{y}')[\bar{y}' \rightarrow \bar{y}] \vdash \mathbf{rely} A[\text{in}_i(\bar{y})/x] \mathbf{guar} A \quad (\text{Sel})$

We next derive (26) from (27). Let  $A \Downarrow_x$ .

1.  $\forall i.\bar{x}'\text{in}_i\langle\bar{y}_i\rangle \vdash \mathbf{rely} A[x'/x][\text{in}_i(\bar{y}_i)/x'] \mathbf{guar} A \quad (29)$
2.  $x[\&_i(\bar{y}_i).\bar{x}'\text{in}_i\langle\bar{y}_i\rangle] \vdash \mathbf{rely} A[x'/x] \mathbf{guar} A \quad (\text{Bra})$

Finely we infer (25) from (27) and (28), let  $A' \stackrel{\text{def}}{=} A[i/x] \wedge z = i \bullet \bar{j}$  with  $i \bar{j}$  fresh.

1.  $\bar{x}'\langle\bar{y}z\rangle \vdash \mathbf{rely} x' = i \wedge A'[x' \bullet \bar{y}/z] \mathbf{guar} A' \quad (29)$

---

2.  $\bar{x}'\langle\bar{y}z\rangle \vdash \mathbf{rely} x' = i \wedge A[i/x] \wedge \bar{y} = \bar{j} \mathbf{guar} A' \quad (\text{Conseq})$

---

3.  $!x(\bar{y}z).\bar{x}'\langle\bar{y}z\rangle \vdash \mathbf{rely} x' = i \wedge A[i/x] \mathbf{guar} \forall \bar{y}.\bar{j}.(\bar{y} = \bar{j} \supset A'[x \bullet \bar{y}/z]) \quad (\text{In-}\forall)$

---

4.  $!x(\bar{y}z).\bar{x}'\langle\bar{y}z\rangle \vdash \mathbf{rely} x' = i \wedge A[i/x] \mathbf{guar} A[i/x] \wedge \forall \bar{j}.x \bullet \bar{j} = i \bullet \bar{j} \quad (\text{Conseq})$

---

5.  $!x(\bar{y}z).\bar{x}'\langle\bar{y}z\rangle \vdash \mathbf{rely} x' = i \wedge A[i/x] \mathbf{guar} \exists i.(A[i/x] \wedge x = i) \quad (\text{Conseq})$

---

6.  $!x(\bar{y}z).\bar{x}'\langle\bar{y}z\rangle \vdash \mathbf{rely} \exists i.(x' = i \wedge A[i/x]) \mathbf{guar} \exists i.(A[i/x] \wedge x = i) \quad (\text{Aux-}\exists)$

---

7.  $!x(\bar{y}z).\bar{x}'\langle\bar{y}z\rangle \vdash \mathbf{rely} A[x'/x] \mathbf{guar} A \quad (\text{Conseq})$

Note we used extensionality in Line 5. This concludes the proof of Lemma 1.

## 5 PCFv and its Process Encoding

### 5.1 Review of PCFv

This section reviews PCFv and its encoding into processes. First we summarise the syntax of PCFv-preterms.

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

For simplicity of presentation, we focus on the simplest form of this calculus, even though such standard extensions as sums, products and letrec can be easily treated. Binding etc. are standard, based on which we assume the standard bound name convention. Preterms are considered up to the renaming of bound names.  $\alpha, \beta, \dots$  are types, whose grammar is to be given soon.  $\mathbf{c}$  is a constant for which we consider booleans  $\mathbf{t}, \mathbf{f}$  and natural numbers  $n$ .  $\mathbf{op}(M_0, \dots, M_{n-1})$  denotes an  $n$ -ary arithmetic or boolean operation (e.g.  $\text{succ}(M), M \wedge N$ ). *Values*  $(V, V', \dots)$  are variables, constants,  $\lambda$ -abstractions and recursions.

We use the standard weak call-by-value one-step reduction (Gunter 92), written  $M \longrightarrow M'$ . For reference we list the reduction rules. Below in the rule for recursion note  $x \notin \text{fv}(V)$  by the bound name convention. In the rule for first-order operations,  $\underline{\mathbf{op}}$  is the function underlying  $\mathbf{op}$ , similarly  $\underline{\mathbf{c}}$  is the value underlying a constant.

$$\begin{aligned} (\lambda x.M)V &\longrightarrow M[V/x] \\ (\mu x.\lambda y.M)V &\longrightarrow M[V/y][(\mu x.\lambda y.M)/x] \\ \text{if } \mathbf{t} \text{ then } N_1 \text{ else } N_2 &\longrightarrow N_1 \\ \text{if } \mathbf{f} \text{ then } N_1 \text{ else } N_2 &\longrightarrow N_2 \\ \mathbf{op}(\mathbf{c}_1.. \mathbf{c}_n) &\longrightarrow \mathbf{c}' \quad (\underline{\mathbf{op}} : \underline{\mathbf{c}}_1.. \underline{\mathbf{c}}_n \mapsto \underline{\mathbf{c}}') \end{aligned}$$

We then close the rules under the weak call-by-value evaluation context, generated from:

$$C[\cdot] ::= [\cdot] \mid C[\cdot]M \mid VC[\cdot] \mid \mathbf{op}(\vec{VC}[\cdot]\vec{M}) \mid \text{if } C[\cdot] \text{ then } N_1 \text{ else } N_2$$

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

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

$\mathbb{B}$  and  $\mathbb{N}$  are *atomic types*, while  $\alpha \Rightarrow \beta$  is an *arrow type*. A typed term is written  $\Gamma \vdash M : \alpha$ , where  $\Gamma$  is a *basis*, which is a finite map from variables to types. The typing rules are standard (Gunter 92) and are omitted.

We use the standard contextual congruence on PCFv-terms, which we write  $\cong_\lambda$ . A *typed relation on PCFv-terms* is a relation which only relate two terms of the same basis and type. We write  $\Gamma \vdash M_1 R M_2 : \alpha$  when  $\Gamma \vdash M_{1,2} : \alpha$  are well-typed terms and they are related by a typed relation  $R$ . Then  $\cong_\lambda$  is the maximum typed congruence satisfying the following condition.

$$\Gamma \vdash M_1 \cong_\lambda M_2 : \mathbb{N} \quad \supset \quad (M_1 \Downarrow \equiv M_2 \Downarrow) \quad (30)$$

where  $M \Downarrow$  denotes  $\exists V.(M \longrightarrow^* V)$ . We can show  $\cong_\lambda$  is maximally consistent, i.e. adding any extra equation and taking the congruent closure leads to the universal typed relation.

## 5.2 Encoding and Definability

PCFv can be fully abstractly embedded into affine processes (Berger et al. 2001) using the standard encoding by Milner (Milner 90) (cf. (Honda and Yoshida 99)). We first give the encoding of types.

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

We may consider  $\alpha^\circ$  as the embedding of a type as a set of values. For each term which may possibly diverge, its type is mapped into  $(\alpha^\circ)^\dagger$ . From this viewpoint,  $\mathbb{N}^\circ$  and  $\mathbb{B}^\circ$  had better contain only those processes which, upon invocation, immediately emits an output, such as  $!x(c).\bar{c}i_{n_3}$ : their interaction is better to be total (rather than partial). For this reason we shall consider both  $\mathbb{B}^\circ$  and  $\mathbb{N}^\circ$  are inhabited by these total behaviours: the typing rules which precisely type total behaviours are given in Appendix C.<sup>4</sup>

The encoding of programs closely follows that of types, given below. There are two kinds:  $\llbracket V \rrbracket_m$  (for values) and  $\langle\langle M \rangle\rangle_u$  (for general terms). The encoding maps a typed term to a process: for simplicity we assume variables are explicitly typed, and add type annotation to terms as needed. In the sixth line,  $\mathcal{S}(\mathbf{i})$  is the successor of  $\mathbf{i}$ .

$$\begin{aligned} \llbracket \mathbf{b} \rrbracket_m &\stackrel{\text{def}}{=} !m(c).\bar{c}i_{n_{\mathbf{b}}} \\ \llbracket \mathbf{n} \rrbracket_m &\stackrel{\text{def}}{=} !m(c).\bar{c}i_{n_{\mathbf{n}}} \\ \llbracket x^\alpha \rrbracket_m &\stackrel{\text{def}}{=} [m \rightarrow x]^{\alpha^\circ} \\ \llbracket \lambda x^\alpha.M \rrbracket_m &\stackrel{\text{def}}{=} !m(xc).\langle\langle M \rangle\rangle_c \\ \llbracket \mu x^{\alpha \Rightarrow \beta}.V \rrbracket_m &\stackrel{\text{def}}{=} (v x)(\llbracket V \rrbracket_m | [x \rightarrow m]^{(\alpha \Rightarrow \beta)^\circ}) \\ \langle\langle V \rangle\rangle_u &\stackrel{\text{def}}{=} \bar{u}(m)\llbracket V \rrbracket_m. \\ \langle\langle \text{succ}(M) \rangle\rangle_u &\stackrel{\text{def}}{=} (m_1)(\langle\langle M \rangle\rangle_m | m(c).\bar{c}(e)e[\&\mathbf{i}.\llbracket \mathcal{S}(\mathbf{i}) \rrbracket_u]) \\ \langle\langle M^{\alpha \Rightarrow \beta} N^\alpha \rangle\rangle_u &\stackrel{\text{def}}{=} (v m)(\langle\langle M \rangle\rangle_m | m(c).(v n)(\langle\langle N \rangle\rangle_n | n(e).\bar{c}\langle eu \rangle^{\overline{\alpha^\circ}(\beta^\circ)^\dagger})) \\ \langle\langle \text{if } M \text{ then } N_t \text{ else } N_f \rangle\rangle_u &\stackrel{\text{def}}{=} (v m)(\langle\langle M \rangle\rangle_m | m(b).\bar{b}(c)c[\&_{b \in \mathbb{B}} \langle\langle N_b \rangle\rangle_u]) \end{aligned}$$

The following properties are established in (Berger et al. 2001) (for (2), we augment the arguments for definability in (Berger et al. 2001) with the constraint on computability of processes, cf. Appendix B). The results extend to products, sums and other constructs. In (1),  $\overline{\Gamma^\circ}$  is the result of pointwise mapping and dualising each channel type.

**Proposition 2** (Berger, Honda, Yoshida (Berger et al. 2001))

1. (type preservation) *If  $\Gamma \vdash M : \alpha$  then  $\vdash \langle\langle M \rangle\rangle_u \triangleright \overline{\Gamma^\circ}$ ,  $u : (\alpha^\circ)^\dagger$ .*
2. (definability) *For each  $\vdash P \triangleright u : (\alpha^\circ)^\dagger$ , there exists a PCFv-term  $\vdash M : \alpha$  such that  $\langle\langle M \rangle\rangle_u \cong_\lambda P$ .*
3. (full abstraction) *For each  $\Gamma \vdash M_{1,2} : \alpha$ , we have  $\Gamma \vdash M_1 \cong_\lambda M_2 : \alpha$  iff  $\langle\langle M_1 \rangle\rangle_u \cong_\pi \langle\langle M_2 \rangle\rangle_u$ .*

<sup>4</sup> Linear outputs of these agents are *truly linear* in the sense of (Yoshida et al. 2001). We can also use a terser, but less uniform, encoding in (Berger et al. 2001), in which case we can use affine processes as themselves. The present encoding is more convenient for illustration of encodings. Apart from the definability result in this section, this notion is never used.

## 6 Program Logic from Process Logic

### 6.1 Assertions for PCFv

As we briefly outlined in Section 2.7, the significance of type preservation in Proposition 2 (1) is that it guides us how we can assert on PCFv-terms through translation. We start from an assertion on the encoding of, say,  $\Gamma \vdash M : \alpha$ . By the type preservation, we know it has the following form:

$$\langle\langle M \rangle\rangle_u \vdash \mathbf{rely} A \mathbf{guar} B \quad (31)$$

where  $A$  is about  $\overline{\Gamma}^\circ$  and  $B$  is about  $u : (\alpha^\circ)^\dagger$  as well as  $\overline{\Gamma}^\circ$ . A key idea is to decode these assertions back to  $M$ , to obtain:

$$\{A'\} M :_u \{B'\} \leftrightarrow \langle\langle M \rangle\rangle_u \vdash \mathbf{rely} A \mathbf{guar} E \quad (32)$$

where  $A'$  and  $B'$  are suitably defined counterparts of  $A$  and  $B$ . Inheriting from the process encoding, the assertion uses a (fresh) name,  $u$ , which we call *anchor*. This anchor is always total, in the same way  $u$  in  $B$  of (31) is always total (except when  $A$  is unsatisfiable). For example,  $B$  may be  $\exists m.(u = (m)^\dagger \wedge \forall x.m \bullet x = (x)^\dagger)$ , in which case it can be decoded into  $\forall x.u \bullet x = x$ . An anchor is used for representing the point of operation (hence specification) of  $M$ . Using  $u$ , the formula  $B'$  can now talk about what this program does, under the assumption  $A'$  for the environment. The “rely” formula is now placed in the position of the type environment, while the original “guarantee” formula is placed in the position of the type of  $M$ . We use the notation from Hoare triple for familiarity: but the meaning of assertions is quite different from the standard Hoare triple. Among others there is no temporary precedence between  $A'$  and  $B'$ .

We formally introduce syntax of assertions. Below  $\star \in \{\wedge, \vee, \supset\}$  and  $Q \in \{\forall, \exists\}$ .

$$\begin{aligned} e & ::= x^\alpha \mid b \mid n \mid \mathbf{t} \mid \mathbf{f} \mid \perp^\alpha \mid e_1 \bullet e_2 \mid e_1 + e_2 \mid e_1 \times e_2 \mid \dots \\ A & ::= \mathbf{T} \mid \mathbf{F} \mid e_1 = e_2 \mid \neg A \mid A_1 \star A_2 \mid Qx.A \end{aligned}$$

$\perp^\alpha$  denotes divergence, often written  $\perp$ .  $\perp$  is seldom used directly in specifications but is needed for semantic completeness: even if  $e$  and  $e'$  are total  $e \bullet e'$  can become partial, and this partiality (divergence) is represented by this constant. In practice, it is used to represent convergence by negation. For this purpose the following notation is useful.

$$e \Downarrow \stackrel{\text{def}}{=} e \neq \perp. \quad (33)$$

Note  $e \Downarrow$  corresponds to  $a \Downarrow$  in the process logic, cf. §4.2. The operator  $\bullet$  can be understood as the call-by-value (or strict) version of the application operation in combinatory logic (Curry and Feys 58) (cf. (Fiore and Honda 1998)). Terms are typed in the following way.

- A constant is given the atomic type it belongs to.
- A name  $x^\alpha$  is typed with  $\alpha$ . Similarly for  $\perp^\alpha$ .
- $op(e_1..e_n)$  has type  $\beta$  if  $op$  has signature  $\alpha_1 \times \dots \times \alpha_n \Rightarrow \beta$  and each  $e_i$  has type  $\alpha_i$ .
- $e_1 \bullet e_2$  is typed with  $\beta$  if  $e_1$  (resp.  $e_2$ ) has type  $\alpha \Rightarrow \beta$  (resp.  $\alpha$ ) for some  $\alpha$ .

Formulae, for which we always assume the standard bound name convention, are well-typed if any pair of two occurrences of a variable have the same type, and that terms equated by  $=$  are of the same type. We write  $\Gamma \vdash A$  when all free names in  $A$  are typed following  $\Gamma$ . We observe:

**Remark 4** The logical language distinguishes a term of a boolean type (e.g.  $\mathbf{t}$ ) from a formula (e.g.  $\top$ ). This is necessary since a boolean term can take the value  $\perp$ . In practice, however, we can often infer a boolean term in question is total, in which case it can be soundly considered to be a formula, as is usually done in Hoare Logic and its extensions.

We have two forms of judgements:

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

We write  $\{A\} M^{\Gamma;\alpha} :_u \{B\}$  when it does not matter whether it is about provability or validity, or when it is clear which from a given context. In  $\{A\} M^{\Gamma;\alpha} :_u \{B\}$ , we always assume the following well-typedness conditions:

- $\Gamma \vdash M : \alpha$ .
- For some  $\Theta$  such that  $\text{dom}(\Gamma) \cap \text{dom}(\Theta) = \emptyset$ , we have  $\Gamma \cdot \Theta \vdash A$  and  $\Gamma \cdot \Theta \cdot u : \alpha \vdash B$ .

Following the process logic in Section 4, we call  $A$  (resp.  $B$ ) *rely formula* (resp. *guarantee formula*) of the judgement. We stress there is *no* temporary precedence relation between  $A$  and  $B$ , unlike in the standard Hoare logic. In the judgement, the primary names in  $A$  are those from  $\text{dom}(\Gamma)$ , while the primary names in  $B$  are those from  $\text{dom}(\Gamma) \cup \{u\}$ . Those free names in rely/guarantee formulae which are not primary are *auxiliary*. In Hoare logics, auxiliary names are often used for denoting content of imperative variables before state change (as in, for example,  $\{x = i\} x := x + 1 \{x = i + 1\}$ ). In the present setting, auxiliary names are useful for (among others) reasoning about recursive behaviours, as we shall see later. We often omit type annotation on PCFv-term, writing e.g.  $\{A\} M :_u \{B\}$ .

Intuitively, a judgement  $\{A\} M :_u \{B\}$  under the typing  $\vec{x} : \vec{\tau} \vdash M : \beta$  means that:

*If closed values of type  $\vec{\tau}$  satisfy  $A$ , then the result of substituting them for variables in  $M$  converges to a value whose behaviour satisfies  $B$ , under an arbitrary interpretation of auxiliary names.*

Note the judgement entails convergence of the resulting term as far as  $A$  is satisfiable. Also note we consider all free variables of a term are substituted for values, rather than general terms. This is because having divergent behaviours in the environment entails divergence of the program itself in the standard call-by-value semantics. Thus all primary names in a judgement denote total behaviours, either by assumption (for variables from the basis) or as a property to be guaranteed by the judgement itself (for the anchor). In contrast, auxiliary names may denote possibly divergent data (which is natural given terms are in general partial). We shall later substantiate this informal reading through formal semantics.

**Remark 5** (alternative formulation of PCFv-logic for total correctness) The distinction between totality for primary names and potential partiality for auxiliary names can be avoided using an alternative formulation of the PCFv-logic for total correctness. In this alternative formulation, the syntax of terms takes off  $\perp$  and  $e \bullet e'$ , and assume all terms, including both primary and auxiliary names, denote total values. The application operation is now part of a special formula, written  $e_1 \bullet e_2 \searrow e_3$ , which says applying  $e_1$  to  $e_2$  converges and becomes  $e_3$ . The proof rules only differ in the rules for abstraction and application. While we do not treat this formulation in the present paper, it is worth noting that it offers a natural basis for imperative extensions of the present logic.

Let us look at a couple of examples. The first one is a simple judgement for the identity.

$$\{\top\} \lambda x^\alpha. x :_u \{ \forall y^\alpha. u \bullet y = y \} \quad (34)$$

The judgement says that the program, when applied to any well-typed argument of type  $\alpha$ , will return that argument itself as a result. This sequent has a precise analogue in the process logic via encoding.

$$\bar{u}(c)!c(xm).\bar{m}\langle x \rangle^{\bar{\alpha}^\circ} \vdash \mathbf{rely} \top \mathbf{guar} \exists c. (u = (c)^\dagger \wedge \forall y. c \bullet y = (y)^\dagger) \quad (35)$$

which is easily derivable in the proof rules in §2.3. Note  $(y)^\dagger$  in (35) becomes simply  $y$  in (34), similarly for  $u$ .

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

$$\{\forall x^\mathbb{N}. \mathit{Even}(f \bullet x)\} (f3) + 1 :_u \{\mathit{Odd}(u)\} \quad (36)$$

where  $\mathit{Even}(n)$  (resp.  $\mathit{Odd}(n)$ ) says that  $n$  is even (resp. odd). In the process logic, this assertion becomes, writing  $\underline{a \text{ gives } e}$  for  $a \bullet \varepsilon = \mathit{in}_e()$  for brevity:

$$\begin{aligned} & \bar{f}(xc)(\llbracket 3 \rrbracket_x | c(y).\bar{y}(e)e[\&n.\bar{u}(c)\llbracket n+1 \rrbracket_c]) \vdash \\ & \mathbf{rely} \forall x. \exists i. (\underline{f \bullet x \text{ gives } i} \wedge \mathit{Even}(i)) \mathbf{guar} \exists c, j. (u = (c)^\dagger \wedge \underline{c \text{ gives } j} \wedge \mathit{Odd}(j)) \end{aligned}$$

Indirection complicates the formulae: yet we can precisely read out a more abstract assertion in (36) from its decomposed version.

Finally we show an example where an auxiliary variable is used non-trivially. The following is an assertion for a factorial program before it gets  $\mu$ -binding.

$$\{\forall j \preceq i. f \bullet j = j!\} \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x-1) :_u \{f \bullet i = i!\} \quad (37)$$

In the above judgement, we use the course-of-value induction. It says that, for any number  $j$  under  $i$ , the result of feeding  $f$  with  $j$  is the factorial of  $j$ , then the result of feeding  $u$  with  $i$  is the factorial of  $i$ . Note  $i$  is auxiliary in this judgement: by using  $i$ , we can connect what we assumed for  $f$  and what we can guarantee for  $u$  inductively.

## 6.2 Semantics of Judgement

As we already noted, a judgement  $\{A\} M :_u \{B\}$  under the typing  $\vec{x} : \vec{\tau} \vdash M : \beta$  intuitively means that:

*If a vector of closed values of type  $\vec{\tau}$  satisfy  $A$ , then the result of substituting them for variables in  $M$  converges to a value whose behaviour satisfies  $B$ , under an arbitrary interpretation of auxiliary names.*

Below we substantiate this informal reading in a simplest possible way, using a model constructed from behaviour of terms. Alternatively we may use any (extensional) model of PCFv. Such models can be constructed from bottomless CPOs; we can also construct such models from the sequential affine processes. Indeed, the following model is fully abstractly embeddable to the notion of models in Section 4. We use the contextual congruence for PCFv,  $\cong_\lambda$ , introduced in Section 5.1, (30). Below a PCFv-term is *closed* if its basis is empty. Note a closed term has a unique type.

**Definition 4** (model) *Abstract behaviours*, or simply *behaviours*, ranged over by  $\kappa, \kappa', \dots$ , are the  $\cong_\lambda$ -congruence classes of typed closed terms. We say  $\kappa$  *has type*  $\alpha$ , written  $\kappa : \alpha$ , if its component term is of type  $\alpha$ . For each type  $\alpha$ , there is a unique congruent class of the diverging terms of that type, which we write  $\perp^\alpha$  or simply  $\perp$ . If  $\kappa \neq \perp$ , we say  $\kappa$  is a *total behaviour*. The behaviour containing a constant  $c$  is written  $\underline{c}$ . A *model*, ranged over by  $\xi, \xi', \dots$ , is a finite map from names to total behaviours. An *interpretation*, ranged over by  $I, I', \dots$ , is a finite map from names to possibly non-total behaviours. We write  $\Gamma \vdash \xi$  (resp.  $\Gamma \vdash I$ ) when, for each  $x \in \text{dom}(\xi)$  (resp.  $x \in \text{dom}(I)$ ),  $\xi(x) : \alpha$  (resp.  $I(x) : \alpha$ ) iff  $\Gamma(x) = \alpha$ .

For interpretation of terms, we define operations on abstract behaviours through their counterpart in PCFv in the natural way, which we summarise below for reference.

- Given  $\kappa_1 : \alpha \Rightarrow \beta$  and  $\kappa_2 : \alpha$ , the operation  $\kappa_1 \bullet \kappa_2$  is given as the congruence class including  $MN$  for  $M \in \kappa_1$  and  $N \in \kappa_2$ .
- For each first-order operator  $\text{op}$  with signature  $\alpha_1 \times \dots \times \alpha_n \Rightarrow \beta$  and each  $\kappa_i : \alpha_i$  ( $1 \leq i \leq n$ ), the operation  $\text{op}(\kappa_1.. \kappa_n)$  is defined as the congruence class including  $\text{op}(M_1..M_n)$  for  $M_i \in \kappa_i$  ( $1 \leq i \leq n$ ).

We shall later use a model for interpreting primary names and an interpretation for interpreting auxiliary names. Given a model  $\xi$  and an interpretation  $I$ , we define the map  $\llbracket e \rrbracket I \cdot \xi$  by induction on  $e$  in the obvious way.

$$\begin{aligned} \llbracket \perp^\alpha \rrbracket I \cdot \xi &\stackrel{\text{def}}{=} \perp^\alpha \\ \llbracket c \rrbracket I \cdot \xi &\stackrel{\text{def}}{=} \underline{c} \\ \llbracket x \rrbracket I \cdot \xi &\stackrel{\text{def}}{=} (I \cup \xi)(x). \\ \llbracket \text{op}(e_1..e_n) \rrbracket I \cdot \xi &\stackrel{\text{def}}{=} \text{op}(\kappa_1.. \kappa_n) \quad (\llbracket e_i \rrbracket I \cdot \xi = \kappa_i) \\ \llbracket e \bullet e' \rrbracket I \cdot \xi &\stackrel{\text{def}}{=} (\llbracket e \rrbracket I \cdot \xi) \bullet (\llbracket e' \rrbracket I \cdot \xi) \end{aligned}$$

The satisfaction relation  $\xi \models^I A$  is standard, interpreting the equality as the identity relation. For reference we again list the defining clauses. Below connectives/quantifiers on the right-hand side are semantic ones. We let  $\star \in \{\wedge, \vee, \supset\}$  and  $\mathcal{Q} \in \{\forall, \exists\}$ .

$$\begin{aligned} \xi \models^I e_1 = e_2 &\equiv \llbracket e_1 \rrbracket I \cdot \xi = \llbracket e_2 \rrbracket I \cdot \xi \\ \xi \models^I A_1 \star A_2 &\equiv (\xi \models^I A_1) \star (\xi \models^I A_2) \\ \xi \models^I \neg A &\equiv \neg (\xi \models^I A) \\ \xi \models^I \mathcal{Q}x^\alpha. A &\equiv \mathcal{Q}\kappa \in \llbracket \alpha \rrbracket. \xi \models^{I \cdot x : \kappa} A \end{aligned}$$

In the last line, quantified name are treated in the same way as auxiliary names. This is because they may denote partial behaviours. The alternative formulation discussed in Remark 5, §6.1, Page 31, uses only total terms, treating interpretation of primary, auxiliary and quantified names on the same footing.

Let  $\vdash M_i : \alpha_i$  ( $1 \leq i \leq n$ ). We define:

$$x_1 : M_1, \dots, x_n : M_n \models^I A \stackrel{\text{def}}{=} \exists \kappa_1..n. (\bigwedge_{1 \leq i \leq n} M_i \in \kappa_i \wedge x_1 : \kappa_1, \dots, x_n : \kappa_n \models^I A) \quad (38)$$

**Definition 5** (semantics of judgement) *Assume  $\vec{x} : \vec{\beta} \vdash M : \alpha$ . Then  $\models \{A\} M :_u \{B\}$  (which we read: relying on  $A$ ,  $M$  named as  $u$  satisfies  $B$ ), iff, for each well-typed  $I$  and each closed  $\vec{V}$  of type  $\vec{\beta}$  such that  $\vec{x} : \vec{V} \models^I A$ , we have  $\vec{x} : \vec{V} \cdot u : M[\vec{V}/\vec{x}] \models^I B$ .*

### 6.3 Embedding and Logical Full Abstraction

The validity in the PCFv-logic is precisely embeddable into that of the process logic. To see how this can be done, we consider the now familiar assertion for the identity and how it can be translated into formulae in the process logic.

$$\forall x^\alpha. u \bullet x = x \quad (39)$$

First assume  $u$  is an anchor. In that case,  $u$  is always total. However  $x$  is not primary hence can be divergent. This means we can expand (39) without changing meaning as follows.

$$\forall x^\alpha. (x \Downarrow \supset u \bullet x = x) \wedge (x = \perp \supset u \bullet x = \perp) \quad (40)$$

We can then observe the part  $x = \perp \supset u \bullet x = \perp$  is tautology (which is made into an axiom later), so that the following is in fact logically equivalent to (39) and (40).

$$\forall x^\alpha. (x \Downarrow \supset u \bullet x = x) \quad (41)$$

A natural translation of this assertion into the process logic follows (we use  $v$  instead of  $u$ , for the reason which becomes clear soon).

$$\forall x^{(\alpha^\circ)^\dagger}. \forall y^{\alpha^\circ}. (x = (y)^\dagger \supset v \bullet y = x) \quad (42)$$

Using the substitutivity (replacing  $x$  with  $(y)^\dagger$ ) this can be simplified into:

$$\forall y^{\alpha^\circ}. v \bullet y = (y)^\dagger \quad (43)$$

Finally, remembering  $u$  should have the type  $(\beta^\circ)^\dagger$  with  $\beta = \alpha \Rightarrow \alpha$  in the encoding, and noting  $v$  above is of type  $\alpha^\circ$ , we arrive at:

$$\exists v. (u = (v)^\dagger \wedge \forall y^{\alpha^\circ}. v \bullet y = (y)^\dagger) \quad (44)$$

which says the process surely outputs a name that denotes the behaviour acting as the identity. We may compare (44) with the original assertion (39). While the partiality complicates the latter's direct translation (42), we arrive at a succinct logical description of the program's behaviour as interacting processes through logical equivalences.

If, on the other hand,  $u$  in (39) is primary but is *not* an anchor, then  $u$  in the encoded assertion should have the type  $\alpha^\circ$ , hence we can simply replace  $v$  in (43) with  $u$ , reaching:

$$\forall y^{\alpha^\circ}. u \bullet y = (y)^\dagger \quad (45)$$

Finally, if  $u$  is auxiliary, then  $u$  should be treated just as  $x$  above: it can be partial, and, if so, the result of application diverges. Elaborating (44), which only treats the case when  $u$  is total, we reach:

$$u \Downarrow \supset \exists v. (u = (v)^\dagger \wedge \forall y^{\alpha^\circ}. v \bullet y = (y)^\dagger) \quad (46)$$

Since  $(u = (x)^\dagger \wedge u = (y)^\dagger) \supset x = y$  (cf. the axiom **data** in Page 25), this is logically equivalent to:

$$\forall v. (u = (v)^\dagger \supset \forall y^{\alpha^\circ}. v \bullet y = (y)^\dagger) \quad (47)$$

which precisely captures the situation where, in (39),  $u$  can be partial in the process encoding of the corresponding behaviour. In this way, the same original assertion (39) in PCFv-logic is translated into three assertions in the process logic, depending on the typing on free names.

We now introduce the encoding formally. The encoding is written  $\langle\langle A \rangle\rangle^\Gamma$ , where  $\Gamma$  denote those names which we stipulate to be total and which are translated through the mapping  $(\cdot)^\circ$ , i.e. into replicated types (the special case where a primary name is an anchor is treated at the level of judgement, on the basis of this mapping). Names in  $A$  other than  $\Gamma$  are typed with  $((\cdot)^\circ)^\dagger$ , allowing the possibility to denote divergence. The encoding is by induction on formulae, given by the following clauses. As always, we let  $\star \in \{\wedge, \vee, \supset\}$  and  $\Omega \in \{\forall, \exists\}$ .

$$\begin{aligned} \langle\langle e_1 = e_2 \rangle\rangle^\Gamma &\stackrel{\text{def}}{=} \exists g. (\langle\langle e_1 \rangle\rangle_g^\Gamma \wedge \langle\langle e_2 \rangle\rangle_g^\Gamma) \\ \langle\langle A_1 \star A_2 \rangle\rangle^\Gamma &\stackrel{\text{def}}{=} \langle\langle A_1 \rangle\rangle^\Gamma \star \langle\langle A_2 \rangle\rangle^\Gamma \\ \langle\langle \neg A \rangle\rangle^\Gamma &\stackrel{\text{def}}{=} \neg \langle\langle A \rangle\rangle^\Gamma \\ \langle\langle \Omega x^\alpha. A \rangle\rangle^\Gamma &\stackrel{\text{def}}{=} \Omega x^{(\alpha^\circ)^\dagger}. \langle\langle A \rangle\rangle^\Gamma \end{aligned}$$

Note an equation of terms in the first line is not translated directly into an equation of terms. This is because terms in the PCFv-logic include composite values, such as  $x \bullet (y \bullet 3)$ , which has no counterpart in the sequential process logic: a term in the process logic can only represent the result of interaction at one time. Thus we use the translation  $\langle\langle e \rangle\rangle_u^\Gamma$ , which translates  $e$  into its decomposed form, naming it as  $u$ . The map is defined by structural induction on  $e$ . Below we write, with  $e$  being a data expression of the process logic,  $x$  emits  $e$  for  $\exists y. (x = (y)^\dagger \wedge y \bullet \varepsilon = \text{in}_e())$ .

$$\begin{aligned} \langle\langle x \rangle\rangle_u^\Gamma &\stackrel{\text{def}}{=} \begin{cases} u = x & (x \notin \text{dom}(\Gamma)) \\ u = (x)^\dagger & (x \in \text{dom}(\Gamma)) \end{cases} \\ \langle\langle \mathbf{c} \rangle\rangle_u^\Gamma &\stackrel{\text{def}}{=} \underline{u \text{ emits } \mathbf{c}} \\ \langle\langle \mathbf{op}(\vec{e}) \rangle\rangle_u^\Gamma &\stackrel{\text{def}}{=} \exists \vec{y}. (\wedge_i \langle\langle e_i \rangle\rangle_{y_i}^\Gamma \wedge \forall \vec{z}. (\wedge_i y_i \text{ emits } z_i \supset \underline{u \text{ emits } \mathbf{op}(\vec{z})}) ) \\ \langle\langle e_1 \bullet e_2 \rangle\rangle_u^\Gamma &\stackrel{\text{def}}{=} \exists y_1 y_2. (\wedge_i \langle\langle e_i \rangle\rangle_{y_i}^\Gamma \wedge \forall z_1 z_2. (\wedge_i y_i = (z_i)^\dagger \supset g = z_1 \bullet z_2)) \\ \langle\langle \perp \rangle\rangle_u^\Gamma &\stackrel{\text{def}}{=} \begin{cases} \neg u \Downarrow & (u \notin \text{dom}(\Gamma)) \\ \mathbf{F} & (u \in \text{dom}(\Gamma)) \end{cases} \end{aligned}$$

In the last line, observe  $u$  is mapped to a name with the replicated type when  $u \in \text{dom}(\Gamma)$ , in which case the statement  $u \Downarrow$  does not make sense. This precisely corresponds to the situation where, at the level of PCFv-logic, it does not make sense to say a primary name diverges.

As an example, we consider the previous assertion for identity. Let  $\Gamma = u : \alpha$  below.

$$\begin{aligned} \langle\langle \forall x^\alpha. u \bullet x = x \rangle\rangle^\Gamma &\stackrel{\text{def}}{=} \forall x^{(\alpha^\circ)^\dagger}. \exists g. (\langle\langle u \bullet x \rangle\rangle_g^\Gamma \wedge \langle\langle x \rangle\rangle_g^\Gamma) \\ &\equiv \forall x^{(\alpha^\circ)^\dagger}. \exists g. (\langle\langle u \bullet x \rangle\rangle_g^\Gamma \wedge x = g) \\ &\equiv \forall x^{(\alpha^\circ)^\dagger}. \langle\langle u \bullet x \rangle\rangle_x^\Gamma \\ &\equiv \forall x^{(\alpha^\circ)^\dagger}. (\exists y. (y = (u)^\dagger \wedge (\forall z_1 z_2. (y = (z_1)^\dagger \wedge x = (z_2)^\dagger \supset z_1 \bullet z_2 = x))) \\ &\equiv \forall x^{(\alpha^\circ)^\dagger}. (\forall z^{\alpha^\circ}. (x = (z)^\dagger \supset u \bullet z = x)) \\ &\equiv \forall z^{\alpha^\circ}. (x = (z)^\dagger \supset u \bullet z = (z)^\dagger). \end{aligned}$$

The translation shows the process of decomposing coalesced PCFv-types into their affine (partial) and replicated (total) parts, taking partiality into consideration along the way. This

decomposition makes the mapping complex, which in turn suggests abstraction we obtain when we move from the fine-grained process logic to a logic tailored for a specific programming language.

A key result on the logical embedding follows. The property is called *logical full abstraction* by Longley and Plotkin (Longley 98), though in a different setting. The proof uses Proposition 2 (in particular definability). Below we use the entailment (noting  $u$  is primary)  $u \Downarrow \supset (\exists m. (u = (m)^\dagger \wedge \langle\langle B \rangle\rangle^{\Gamma, u: \alpha} [m/u]) \supset \langle\langle B \rangle\rangle^{\Gamma, u: \alpha})$ , which allows us to simplify the guarantee formula.

**Theorem 2** (logical full abstraction) *Let  $\Gamma \vdash M : \alpha$ . Then  $\models \{A\} M^{\Gamma, \alpha} :_m \{B\}$  if and only if  $\langle\langle M \rangle\rangle_u^{\overline{\Gamma}, u: (\alpha^\circ)^\dagger} \models \mathbf{rely} \langle\langle A \rangle\rangle^{\Gamma^\circ} \mathbf{guar} \langle\langle B \rangle\rangle^{\Gamma^\circ}$ .*

*Proof*

Given a total abstract behaviour  $\kappa$ , let  $\llbracket \kappa \rrbracket$  denote the result of translating a term in  $\kappa$  into a process, say  $P$ , and then taking the abstract process represented by the concrete process  $P$ . Further, given a model  $\xi$  in the PCFv-logic, write  $\llbracket \xi \rrbracket$  for the point-wise extension of  $\llbracket \kappa \rrbracket$  (resulting in a model in the process logic), similarly for  $\llbracket I \rrbracket$ . We first observe, for each well-typed, closed  $M$ , that:

$$M \models_u^I A \quad \equiv \quad M \Downarrow V \wedge \llbracket V \rrbracket_u \models^{\llbracket I \rrbracket} A^\circ. \quad (48)$$

For this we show:

- (a)  $M \models \kappa$  iff  $\langle\langle M \rangle\rangle \models u : \langle\langle \kappa \rangle\rangle$ , and;
- (b)  $\xi \models^I A$  iff  $\llbracket \xi \rrbracket \models^{\llbracket I \rrbracket} A$  (hence  $\kappa \models_u^{\llbracket I \rrbracket} A$  iff  $\llbracket \kappa \rrbracket_u \models^I A^\circ$ ).

(a) is easy by induction on types, while (b) is by induction on formulae in the PCFv-logic, using Proposition 2 (3) (full abstraction) when  $A$  is  $e_1 = e_2$ . We can now infer as follows. For the “if” direction, let  $\langle\langle M \rangle\rangle_u \models \mathbf{rely} A^\circ \mathbf{guar} B'$  with  $B' \stackrel{\text{def}}{=} \exists m. (u = (m)^\dagger \wedge B^\circ)$ . Below we write  $\llbracket \vec{V} \rrbracket_{\vec{x}}$  for the  $n$ -fold parallel composition of  $\llbracket V_i \rrbracket_{x_i}$ .

$$\begin{aligned} \vec{x} : \vec{V} \models^I A & \supset \llbracket \vec{V} \rrbracket_{\vec{x}} \models^{\llbracket I \rrbracket} A^\circ && ((48) \text{ above}) \\ & \supset (\nu \vec{x}) (\langle\langle M \rangle\rangle_u | \llbracket \vec{V} \rrbracket_{\vec{x}}) \models^{\llbracket I \rrbracket} B' && (\text{Assumption}) \\ & \supset \langle\langle M[\vec{V}/\vec{x}] \rangle\rangle_u \models^{\llbracket I \rrbracket} B' && (\text{Replication Theorem}) \\ & \supset M[\vec{V}/\vec{x}] \Downarrow W \wedge \llbracket W \rrbracket_u \models^{\llbracket I \rrbracket} B^\circ && (\text{Proposition 2 (1)}) \\ & \supset M[\vec{V}/\vec{x}] \models_m^I B && ((48) \text{ above}). \end{aligned}$$

For the “only if” direction, assume  $\{A\} M^{\Gamma, \alpha} :_m \{B\}$  with  $\Gamma = \vec{x} : \vec{\tau}$ . If  $A \equiv F$ , the statement is vacuous. Assume not. By Proposition 2 (3) (definability), we can set  $R_i \cong_{\pi} \llbracket V_i \rrbracket_{x_i}$  for some  $V_i$ .

$$\begin{aligned} \Pi_i R_i^{x_i} \models^{\llbracket I \rrbracket} A^\circ & \supset \llbracket \vec{V} \rrbracket_{\vec{x}} \models^{\llbracket I \rrbracket} A^\circ && (R_i \cong \llbracket V_i \rrbracket_{x_i}) \\ & \supset \vec{x} : \vec{V} \models^I A && ((48) \text{ above}) \\ & \supset M[\vec{V}/\vec{x}] \Downarrow W \wedge \llbracket W \rrbracket_u \models^{\llbracket I \rrbracket} B' && (\text{assumption}) \\ & \supset (\nu \vec{x}) (\langle\langle M \rangle\rangle_u | \llbracket \vec{V} \rrbracket_{\vec{x}}) \models^{\llbracket I \rrbracket} B' && (\text{Replication}) \\ & \supset (\nu \vec{x}) (\langle\langle M \rangle\rangle_u | \Pi_i R_i) \models^{\llbracket I \rrbracket} B'. && (R_i \cong \llbracket V_i \rrbracket_{x_i}). \end{aligned}$$

Note the definability is essential for the second direction.  $\square$

$$\begin{array}{c}
[Var] \frac{-}{\{A[x/u]\} x :_u \{A\}} \quad [Op] \frac{A \stackrel{\text{def}}{=} C_0 \{C_i\} M_i :_{m_i} \{C_{i+1}\} C_n \stackrel{\text{def}}{=} B[\text{op}(m_1, \dots, m_n)/u]}{\{A\} \text{op}(M_1, \dots, M_n) :_u \{B\}} \\
\\
[If] \frac{\{A\} M :_b \{C\} \quad \{C[t/b]\} N_1 :_u \{B\} \quad \{C[f/b]\} N_2 :_u \{B\}}{\{A\} \text{if } M \text{ then } N_1 \text{ else } N_2 :_u \{B\}} \\
\\
[Abs] \frac{\{A^{-x} \wedge B_1^x\} M :_m \{B_2\} \quad B_1 \supset x \Downarrow}{\{A\} \lambda x. M :_u \{B_1 \supset B[u \bullet x/m]\}} \quad [App] \frac{\{A\} M :_m \{C\} \quad \{C\} N :_x \{B[m \bullet x/u] \wedge m \bullet x \Downarrow\}}{\{A\} MN :_u \{B\}} \\
\\
[Rec] \frac{\{A^{-x} \wedge \forall j \leq i. B(j)[x/u]\} \lambda y. M :_u \{B(i)^{-x}\}}{\{A\} \mu x. \lambda y. M :_u \{\forall i \geq 0. B(i)\}} \\
\\
[Consequence] \frac{A \supset A_0 \quad \{A_0\} M :_u \{B_0\} \quad B_0 \supset B}{\{A\} M :_u \{B\}} \\
\\
[\wedge - \supset] \frac{\{C \wedge A\} V :_u \{B\}}{\{A\} V :_u \{C \supset B\}} \quad [\supset - \wedge] \frac{\{A\} V :_u \{C^{-u} \supset B\}}{\{C \wedge A\} V :_u \{B\}} \\
\\
[Aux-\forall] \frac{\{A^{-i}\} M :_u \{B\}}{\{A\} M :_u \{\forall i. B\}} \quad [Aux-\exists] \frac{A M :_u \{B^{-i}\}}{\{\exists i. A\} M :_u \{B\}} \\
\\
[Conj] \frac{A M :_u \{B_{1,2}\}}{A M :_u \{B_1 \wedge B_2\}} \quad [Disj] \frac{A_{1,2} M :_u \{B\}}{A_1 \vee A_2 M :_u \{B\}}
\end{array}$$

Fig. 4. Proof Rules for PCFv

#### 6.4 Proof Rules for PCFv

Below we show how the correspondence between the assertions in the PCFv-logic and those in the process logic leads to simple and natural proof rules for PCFv. Following the proof rules for PCFv, we stipulate:

- In each rule, all occurring judgements, including the conclusion, are well-typed.
- $A^{-\vec{y}}$  indicates no name in  $\vec{y}$  occurs in  $A$ .
- $A^{\vec{y}}$  indicates all *primary* names in  $A$  are in  $\{\vec{y}\}$ .
- $i, j, \dots$  exclusively range over auxiliary names (in a given judgement).
- In each rule, no auxiliary names in a judgement in its premise should overlap with primary names in a judgement in its conclusion.

Each rule can be read naturally from the viewpoint of PCFv-computation.  $[Var]$  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  $u$ . Note the rule precisely corresponds to the copy-cat laws studied in §4.6 through the process encoding of a variable given in §5.2.

[Op] is the rule for first-order operations. To understand the rule, let us consider its specific instances. We start from the case when the arity of operators is zero and one.

$$[Num] \frac{-}{\{A\} [\mathbf{n}/u] \mathbf{n} :_u \{A\}} \quad [Succ] \frac{\{A\} M :_m \{B[m+1/u]\}}{\{A\} succ(M) :_u \{B\}}$$

which are both natural. We also give an example of a binary operator.

$$[Eq] \frac{\{A\} M^{\Gamma;\mathbb{N}} :_{m_1} \{C\} \quad \{C\} N^{\Gamma;\mathbb{N}} :_{m_2} \{B[m_1=m_2/u]\}}{\{A\} (M=N)^{\Gamma;\mathbb{B}} :_u \{B\}}$$

In the premise, the assumption  $A$  and the property on  $m_1$  (hence on  $M$ ) can be together represented in  $C$ . This is carried over to the second premise as an assumption, so that  $B[m_1=v_2/b]$  stipulate on both  $m_1$  and  $m_2$ . By substitution, this property becomes that of  $u$ . While the rule looks as if the order of evaluation — first  $M$ , then  $N$  — matters in the inference, this is not the case because of the stateless nature of PCFv. Indeed, the following rule gives an equivalent presentation (the equivalence is easily established through the structural rules discussed later).

$$[Eq] \frac{\{A\} M :_{m_1} \{B_1\} \quad \{A\} N :_{m_2} \{B_2\} \quad B_1 \wedge B_2 \supset B[m_1=m_2/u]}{\{A\} M=N :_u \{B\}}$$

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

- (1)  $N_1$  under the assumption that  $C$  holds with  $b$  denoting truth;
- (2)  $N_2$  under the assumption that  $C$  holds with  $b$  denoting falsity;

then, again under  $A$ , surely  $B$  holds for  $\text{if } M \text{ then } N_1 \text{ else } N_2$ . Note the rule keeps clean symmetry, as in the corresponding rule in Hoare logic. Observe also  $b \notin \text{fn}(A)$  by the well-formedness of  $\{A\} M :_b \{A'\}$ , and that  $\{A\} M :_b \{A'\}$  indicates (as far as  $A$  is non-trivial)  $M$  terminates. Thus, in a closed term, the conditional branch can surely terminate, reaching one of  $N_i$ , which in turn is guaranteed to terminate and satisfies  $B$  by the premise.

We now move to the three key rules for PCFv-logic, abstraction, application and recursion. First, [Abs] says that, whenever  $M$  named as  $m$  satisfies  $C$  relying on  $A$  (which is not about  $x$ ) and  $B_1$  (which is about  $x$ ), then  $\lambda x.M$  named  $u$  has the behaviour such that, whenever  $u$  is fed with an argument  $x$  satisfying  $B_1$ , returns the result  $(u \bullet x)$  which satisfies  $B_2$  for  $m$ . Note  $x$  becomes auxiliary in the conclusion. This is why the rule has the side condition saying  $B_1$  should mention  $x$  is total.<sup>5</sup> This condition is necessary since, in the premise,  $x$  denotes automatically a total behaviour by being a primary name (which may as well be used for deriving the judgement). When  $x$  becomes auxiliary in the conclusion, we need to maintain this condition, hence the side condition. Without this condition, the following unsound inference is possible.

$$\begin{array}{l} 1. \frac{\{T \wedge T\} 3 :_m \{m = 3\}}{\{T\} \lambda x.3 :_u \{u \bullet x = 3\}} \quad (\text{Op, Conseq}) \\ 2. \frac{\{T\} \lambda x.3 :_u \{u \bullet x = 3\}}{\{T\} \lambda x.3 :_u \{\forall x. u \bullet x = 3\}} \quad (\text{Abs, Conseq}) \\ 3. \{T\} \lambda x.3 :_u \{\forall x. u \bullet x = 3\} \quad (\text{Aux-}\forall) \end{array}$$

<sup>5</sup> The abstraction rule in (Honda 2004c) regrettably fails to mention this side condition.

The guarantee formula in Line 3 is in fact absurdity, since when  $x = \perp$  we always have  $u \bullet x = \perp$ . The side condition in fact naturally arises if we carefully inspect its embedding into the inference in the process logic given as follows, using the encoding we discussed in §6.3. Below we set  $\Gamma \cdot x : \alpha \vdash M : \beta$  and, for brevity, let  $\Delta = \Gamma \cdot x : \alpha$  and  $(\cdot)^\bullet = ((\cdot)^\circ)^\dagger$ . We also set  $C \stackrel{\text{def}}{=} \exists c. (u = (c)^\dagger \wedge \langle\langle B_1^x \supset B_2[c \bullet x/m] \rangle\rangle^{\Delta^\circ})$  and remember we have  $B_1 \Downarrow_x$ .

1.  $\langle\langle M \rangle\rangle_m^{\Delta^\circ, m: \beta^\bullet} \vdash \mathbf{rely} \langle\langle A^{-x} \rangle\rangle^{\Gamma^\circ} \wedge \langle\langle B_1^x \rangle\rangle^{\Delta} \mathbf{guar} \langle\langle B_2 \rangle\rangle^{\Delta} \quad B_1 \Downarrow_x \quad (\text{premise})$
2.  $(!c(xm)). \langle\langle M \rangle\rangle_m^{\Gamma^\circ, c: (\alpha \Rightarrow \beta)^\circ} \vdash \mathbf{rely} \langle\langle A^{-x} \rangle\rangle^{\Gamma} \mathbf{guar} \langle\langle B_1^x \rangle\rangle^{\Delta} \supset \langle\langle B_2[c \bullet x/m] \rangle\rangle^{\Delta} \quad (\text{In})$
3.  $(\bar{u}(c)!c(xm)). \langle\langle M \rangle\rangle_m^{\Gamma^\circ, u: (\alpha \Rightarrow \beta)^\bullet} \vdash \mathbf{rely} \langle\langle A^{-x} \rangle\rangle^{\Gamma} \mathbf{guar} C \quad (\text{Sel})$
4.  $(\bar{u}(c)!c(xm)). \langle\langle M \rangle\rangle_m^{\Gamma^\circ, u: (\alpha \Rightarrow \beta)^\bullet} \vdash \mathbf{rely} \langle\langle A^{-x} \rangle\rangle^{\Gamma} \mathbf{guar} \forall y. (x = (y)^\dagger \supset C[y/x])$
5.  $(\bar{u}(c)!c(xm)). \langle\langle M \rangle\rangle_m^{\Gamma^\circ, u: (\alpha \Rightarrow \beta)^\bullet} \vdash \mathbf{rely} \langle\langle A^{-x} \rangle\rangle^{\Gamma} \mathbf{guar} \langle\langle B_1 \supset B_2[u \bullet x/m] \rangle\rangle^{\Gamma} \quad (\text{Conseq})$

Line 4 is by combination of (Consequence) and (Aux- $\forall$ ) (note renaming by a fresh name is always possible by applying universal quantification and instantiating it to another fresh name). Line 5 is possible because of  $B_1 \Downarrow_x$ : the condition  $x = (y)^\dagger$  is absorbed into  $\langle\langle B_1 \rangle\rangle^{\Gamma^\circ}$  since the formula already says, given  $x$  is auxiliary, that  $x = (z)^\dagger$  for some  $z$ .

The next rule is the one for application. The rule says that, if  $M$ , named as  $m$ , satisfies  $C$ , and  $N$ , relying on  $C$  and named as  $x$ , satisfies  $B[m \bullet x/u]$  as well as it says that  $m$  applies to  $x$  converges, then  $MN$  named as  $u$  satisfies  $B$ . As for other rules, the rule has a symmetric version, given as follows.

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

The rule is easily equivalent to the original rule through the structural rules. It is again instructive to decompose  $[App]$  into inferences in the process logic via encoding. We use the symmetric version above. For brevity, assume  $M$  and  $N$  are values, so that the encoding becomes:

$$\langle\langle MN \rangle\rangle_u \stackrel{\text{def}}{=} (\nu m) (\llbracket M \rrbracket_m | \bar{m}(xu') . (\llbracket N \rrbracket_x | [u' \rightarrow u])).$$

Note  $m$  of  $\llbracket M \rrbracket_m$  is typed as  $(\alpha \Rightarrow \beta)^\circ$ ,  $x$  of  $\llbracket N \rrbracket_x$  is typed as  $\alpha^\circ$ , and  $u$  is typed as  $(\beta^\circ)^\dagger$ . We can now derive the proof rule for application as follows (we let  $\Gamma$  to be the basis for  $M$ ).

1.  $\llbracket M \rrbracket_m \vdash \mathbf{rely} \langle\langle A \rangle\rangle^{\Gamma} \mathbf{guar} \langle\langle B_1 \rangle\rangle^{\Gamma, m: \alpha \Rightarrow \beta} \quad (\text{prem 1})$
2.  $\llbracket N \rrbracket_x \vdash \mathbf{rely} \langle\langle A \rangle\rangle^{\Gamma} \mathbf{guar} \langle\langle B_1 \rangle\rangle^{\Gamma, m: \alpha \Rightarrow \beta} \quad (\text{prem 2})$
3.  $[u' \rightarrow u] \vdash \mathbf{rely} \langle\langle B \rangle\rangle^{\Gamma} [u'/u] \mathbf{guar} \langle\langle B \rangle\rangle^{\Gamma} \quad (\text{copycat law})$
4.  $\bar{m}(xu') (\llbracket N \rrbracket_x | [u' \rightarrow u]) \vdash \mathbf{rely} \langle\langle B_1 \rangle\rangle^{\Gamma} \wedge \langle\langle A \rangle\rangle^{\Gamma} \mathbf{guar} \langle\langle B \rangle\rangle^{\Gamma} \quad (2, 3, \text{prem 3}, (\text{Out}))$
5.  $\llbracket M \rrbracket_m | \bar{m}(xu') . (\llbracket N \rrbracket_x | [u' \rightarrow u]) \vdash \mathbf{rely} \langle\langle A \rangle\rangle^{\Gamma} \mathbf{guar} \langle\langle B \rangle\rangle^{\Gamma} \quad (1, 4, (\text{Par}))$
6.  $\langle\langle MN \rangle\rangle_u \vdash \mathbf{rely} \langle\langle A \rangle\rangle^{\Gamma} \mathbf{guar} \langle\langle B \rangle\rangle^{\Gamma} \quad (\text{Res})$

$$\begin{array}{c}
[Let] \frac{\{A\} M :_x \{C\} \quad \{C\} N :_u \{B\}}{\{A\} \text{let } x = M \text{ in } N :_u \{B\}} \\
[Letrec] \frac{\{A \wedge \forall m \leq n. E(m)[y/x]\} N[y/x] :_x \{E(n)\} \quad \{A \wedge \forall n. E(n)\} M :_u \{B\}}{\{A\} \text{letrec } x = N \text{ in } M :_u \{B\}} \\
[Mletrec] \frac{\{A \wedge \forall m \leq n. E(m)[\vec{y}/\vec{x}]\} V_i :_{x_i} \{E_i(n)\} \quad \{A \wedge \forall n. E(n)\} M :_u \{B\}}{\{A\} \text{letrec } \vec{x} = \vec{V} \text{ in } M :_u \{B\}} (E(n) \stackrel{\text{def}}{=} \bigwedge_i E_i(n))
\end{array}$$

Fig. 5. Let and Letrec

We now turn to the rule for recursion. **[Rec]** is the proof rule for the total correctness of recursion.  $i$  in the premise should be auxiliary by our convention. Observe  $i$  is implicitly universally quantified. The rule says: suppose we have, whenever the environment satisfies  $B(j)[x/u]$  holds for each  $j$  strictly smaller than  $i$ , in addition to  $A$  which is not about  $x$ , the program can guarantee  $B(i)$  (note  $i$  is auxiliary, so that  $i$  is universally quantified). Then we conclude, assuming  $A$  for the environment, the same program in which  $x$  is mu-abstracted satisfies  $\forall i \geq 0. B(i)$ . The condition  $i \geq 0$  (missing from (Honda 2004c)) avoids  $i = \perp$ . When  $i = 0$ , the rely condition becomes  $A$  (since:  $\forall j \leq 0. B(j)[x/u] \equiv \forall j. (j \leq 0 \supset B(j)[x/u]) \equiv \top$ ), giving the base case. In fact, we can derive the following rule as its special case, which is essentially equipotent.

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

The rule says: under  $A$ , the term satisfies, without assuming anything about  $x$ , already  $B(0)$ . The term named as  $u$  also satisfies  $B(n+1)$ , if we assume  $B(n)[x/u]$  for  $x$  in addition to  $A$ . So we expect it satisfies  $B(n)$  for each  $n \geq 0$ . The **[Rec]** rule closely follows **(Rec)** in the total process logic (the latter in fact suggests that the body of recursion in the former can be any value of non-atomic types).

The recursion rule can be used, combined with the application/abstraction rules, to derive the rules for recursive let constructs, as shown in Figure 5. Each rule may need no illustration. We observe the  $(m) \text{letrec}$  construct has a natural representation using wires (Hasegawa 99), which is in a direct correspondence with recursion  $\mu \vec{x} = \vec{y}. P$  in the process.

$$\langle\langle m \text{letrec } \{\vec{x} = \vec{V}\}_{i \in I} \text{ in } M \rangle\rangle_u \stackrel{\text{def}}{=} (\nu \vec{x}) (\langle\langle M \rangle\rangle_u \mid \mu \vec{y} = \vec{x}. \Pi_i [\![V_i[y_i/x_i]\!]_{x_i}]) \quad (49)$$

By applying **(Par)** and **(Res)**, we arrive at the conclusion of **[Mletrec]**.

**Remark 6** From the viewpoint of the process encoding, the let-rules may as well be considered as more primitive than abstraction/application rules. For example,  $\text{let } x = V \text{ in } N$  can be directly translated into:

$$\langle\langle \text{let } x = V \text{ in } N \rangle\rangle_u \stackrel{\text{def}}{=} (\nu c) (\langle\langle V \rangle\rangle_c \mid c(x). \langle\langle N \rangle\rangle_u) \approx (\nu x) (\llbracket V \rrbracket_x \mid \langle\langle N \rangle\rangle_u),$$

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

$$\begin{array}{ll}
\text{(ext)} & \forall y. e_1 \bullet y = e_2 \bullet y \quad \supset \quad e_1 = e_2 \\
\text{(\(\perp\)-left)} & \perp \bullet e = \perp \\
\text{(\(\perp\)-right)} & e \bullet \perp = \perp
\end{array}$$

Fig. 6. Axioms for Strict  $\bullet$ 

Finally the consequence rule (which can be strengthened following (Klemann 83)) and other structural rules all come from the process logic, so that they may not need illustration. To put the consequence rule to real use, 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 6, all easily justifiable from the underlying model (similar rules exist for first-order operations).

As we have shown for [Abs] and [App], each proof rule in Figure 4 is decomposable into a sequence of inferences in the process logic, reaching:

**Proposition 3** *Let  $\Gamma \vdash M : \alpha$ . Then  $\{A\} M :_u \{B\}$  implies  $\langle\langle M \rangle\rangle_u \vdash \mathbf{rely} \langle\langle A \rangle\rangle^\Gamma \mathbf{guar} \langle\langle B \rangle\rangle^\Gamma$ .*

We can now establish a key property of the proof rules.

**Theorem 3** (soundness of PCFv-logic)  $\vdash \{A\} M :_u \{B\}$  implies  $\models \{A\} M :_u \{B\}$ .

*Proof*

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

$$\begin{array}{ll}
\vec{x} : \vec{V} \models^I A & \supset \quad \llbracket \vec{V} \rrbracket_{\vec{x}} \models^{\llbracket I \rrbracket} \langle\langle A \rangle\rangle^{\vec{x}} & \text{(Theorem 2)} \\
& \supset \quad (\nu \vec{x}) (\langle\langle M \rangle\rangle_u \llbracket \vec{V} \rrbracket_{\vec{x}}) \models^{\llbracket I \rrbracket} \langle\langle B \rangle\rangle^u & \text{(Prop. 3, Thm. 1)} \\
& \supset \quad \langle\langle M[\vec{V}/\vec{x}] \rangle\rangle_u \models^{\llbracket I \rrbracket} \langle\langle B \rangle\rangle^u & \text{(by } \cong_\pi \text{)} \\
& \supset \quad M[\vec{V}/\vec{x}] \models^I B & \text{(Theorem 2)}
\end{array}$$

Note the proof needs both directions of Theorem 2.  $\square$

### 6.5 Reasoning Examples in PCFv-Logic

We conclude this section with a couple of simple inference examples. We show the derivation of the following three judgements. Below we write  $\forall x \Downarrow . A$  for  $\forall x. (x \Downarrow \supset A)$ .

$$\{ \top \} \lambda x. x :_u \{ \forall x. (u \bullet x = x) \}. \quad (50)$$

$$\{ \top \} \mathit{Fact} :_u \{ \forall n \Downarrow . u \bullet n = n! \}. \quad (51)$$

$$\{ \top \} \mathit{Gcd} :_u \{ \forall x, y \geq 1. \mathit{gcd}(u \bullet (x, y), x, y) \} \quad (52)$$

where we set:

$$\mathit{Fact} \stackrel{\text{def}}{=} \mu f. \lambda x. \text{if } x=0 \text{ then } 1 \text{ else } x * f(x-1).$$

$$\mathit{Gcd} \stackrel{\text{def}}{=} \mu f. \lambda (x, y). \text{if } x \leq y \text{ then } f(y-x, x) \text{ else} \\ \text{if } x \geq y \text{ then } f(x-y, y) \text{ else } x.$$

$$\begin{array}{c}
\text{[Simple]} \frac{}{\{A[e/u]\} e :_u \{A\}} \\
\text{[If-Simple]} \frac{\{A \wedge e\} M_1 :_u \{B\} \quad \{A \wedge \neg e\} M_2 :_u \{B\}}{\{A\} \text{ if } e \text{ then } M_1 \text{ else } M_2 :_u \{B\}} \\
\text{[App-Simple]} \frac{\{A\} M :_m \{B[m \bullet e/u] \wedge m \bullet e \Downarrow\}}{\{A\} M e :_u \{B\}}
\end{array}$$

Fig. 7. Rules for Simple Expressions

*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 (we shall later present the proof rule for multiple abstraction).

The predicate  $\text{gcd}(m, x, y)$  (read: *m is the g.c.d. of x and y*) is given as:

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

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

We first derive the judgement (50). Below and henceforth we shall freely use trivial logical equivalences.

1.  $\frac{\{x = x\} x :_m \{m = x\}}{\text{}} \quad \text{(Var)}$
2.  $\frac{\{T \wedge x \Downarrow\} x :_m \{m = x\}}{\text{}} \quad \text{(Consequence)}$
3.  $\frac{\{T\} \lambda x. x :_u \{x \Downarrow \supset u \bullet x = x\}}{\text{}} \quad \text{(Abs)}$
4.  $\frac{\{T\} \lambda x. x :_u \{(x \Downarrow \supset u \bullet x = x) \wedge (x = \perp \supset u \bullet x = \perp)\}}{\text{}} \quad \text{(Conseq)}$
5.  $\frac{\{T\} \lambda x. x :_u \{u \bullet x = x\}}{\text{}} \quad \text{(Conseq)}$
6.  $\frac{\{T\} \lambda x. x :_u \{\forall x. u \bullet x = x\}}{\text{}} \quad \text{(Aux-}\forall\text{)}$

In Line 2, we observe  $x$  being total is the tautology when  $x$  is primary. In Line 4, we used the axiom ( $\perp$ -right).

In the next examples, we shall use the following convention, which is often useful in practice (cf. Remark 4).

**Convention 2** *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  $b = T$  (with an implicit outermost conjunction with  $e \Downarrow$ , which should be implied from the preceding derivation).*

The convention is useful but 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).

Under this convention, we shall also use the proof rules in Figure 7. In the rules, we regard a term in the logic  $e$  which does *not* contain application and divergence as a program (note they coincide syntactically). For clarity, we emphasise  $e$  as  $\mathbf{e}$ .

We now prove the judgement (51). Let

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

as well as

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

We also set, for brevity:

$$C \stackrel{\text{def}}{=} \forall j \preceq i. B(f)(i) \wedge x = i$$

We now present the derivation.

1. $\{C \wedge 1 = 1 \wedge x = 0\} 1 :_y \{C \wedge y = 1 \wedge x = 0\}$	(Op)
2. $\{C \wedge x = 0\} 1 :_y \{C \wedge y = i!\}$	(Conseq)
3. $\{C \wedge x \neq 0\} f :_m \{C[m/f] \wedge i \succeq 0\}$	(Var)
4. $\{C \wedge x \neq 0\} f(x-1) :_m \{m = (i-1)!\}$	(App-Simple)
5. $\{C \wedge x \neq 0\} x \times f(x-1) :_m \{m = i!\}$	(Mult)
6. $\{C\} \text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x-1) :_u \{m = i!\}$	(If-Simple)
7. $\{\forall j \preceq i. B(f)(i)\} \lambda x. M :_u \{B(u)(i)\}$	(Abs)
8. $\{\mathbf{T}\} \mu f. \lambda x. M :_u \{B(u)(i)\}$	(Rec)
9. $\{\mathbf{T}\} \mu f. \lambda x. M :_u \{\forall i. B(u)(i)\}$	(Aux- $\forall$ )

Above we have freely used trivial logical equivalences. Note also  $j \preceq i$  implies  $j \Downarrow$ .

We next move to (52), an assertion on GCD. We use the decomposed form of the recursion rule mentioned in Page 40. We let:

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

$$M \stackrel{\text{def}}{=} \text{if } x \preceq y \text{ then } f(y-x, x) \text{ else if } x \succeq y \text{ then } f(x-y, y) \text{ else } x.$$

The base case follows.

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

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

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

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  $C \stackrel{\text{def}}{=} (x = i \wedge y = j \wedge i + j \leq n + 2)$ ,  $E_1 \stackrel{\text{def}}{=} i \leq j$ ,  $E_2 \stackrel{\text{def}}{=} i \geq j$  and  $E_3 \stackrel{\text{def}}{=} i = j$ . In the following inferences, we often do not mention trivial applications of (Conseq).

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

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

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

Further, in Lines 4, 8 and 9, the following implications 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  $i + j - i \leq i + j \leq n + 1$  (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 (52).

## 6.6 Extensions

The specification for the identity function in the previous subsection,  $\forall x.(u \bullet x = x)$ , may at first look no different from the  $\beta_v$ -equality  $(\lambda x.x)x = x$ . A basic difference is that the statement  $\forall x.(u \bullet x = x)$  in the present logic does not mention concrete programs. This allows us to discuss the universe of behaviour insulated from specific programs and programming languages, offering a clean perspective on semantics of specifications.

But it does not end there. More importantly, this method of specifications — 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 significance when the complexity of behaviour increases, either in type/data structures or in the nature of computation such as statefulness. The proposed approach scales in both dimensions, backed up by the underlying process logics (Honda 2004a). For example, to treat sums and products, one only has to add the following terms to the PCFv-logic.

$$e ::= \dots \mid () \mid \langle e_1, e_2 \rangle \mid \pi_i(e) \mid \text{in}_i(e) \quad (i = 1, 2)$$

A pair  $\langle e_1, e_2 \rangle$  is redundant but is convenient in proofs. We can then write, for example,  $\forall z.(\pi_1(u) \bullet z = 2 \times z)$ , which says  $u$  is a pair whose left value is a doubling function. There are (almost obvious) proof rules associated with them, all embeddable into the affine process logic.

To have potentially circular data structure, which is omnipresent in real-world programming, we can add recursive types, which can be simply treated in the process/PCFv-logic if we use the iso-recursive approach (Pierce 2002): we extend types with recursion  $\mu X.\alpha$  (taken up to the standard isomorphism), with no need to add new proof rules. Using recursive types, we can represent, for example, the type of lists as  $\text{List}(\alpha) \stackrel{\text{def}}{=} \mu Y.(\text{Unit} + (\alpha \times Y))$ . We can then reason about a list by unfolding it one by one (which is the best way if a list is large: and would be the only way if it is infinite). For clarity, one may add new terms to the logic, even though all are definable from the given constructs.

$$e ::= \dots \mid [\varepsilon] \mid [e :: e']$$

We can then write down, for example, a specification for a program which eliminates all zero valued-cells from a given list (say  $l$ ), using the following two predicates:

- $A(u, l) \stackrel{\text{def}}{=} l = [\varepsilon] \supset u \bullet l = [\varepsilon]$ .
- $B(u, l) \stackrel{\text{def}}{=} l = [x :: y] \supset (x \neq 0 \supset u \bullet l = [x :: u \bullet y]) \wedge (x = 0 \supset u \bullet l = u \bullet y)$ .

As a first step, the required specification may be written as  $\forall^{\text{List}(\mathbb{N})}. (A(u.l) \wedge B(u.l))$  (see §10 for further detail). The proof rules for lists are also easily derived.

In the same way, the approach smoothly extends to two forms of second-order polymorphism (for both universal and existential abstraction), call-by-name evaluation, higher-order imperative procedures, computation with local state, aliasing and data structure with destructive update (cf. (Reybolds 99)). All these extensions are based on the idea of naming data and procedural objects, and defining suitable operations on typed names. For example, we may describe the behaviour of the universal identity in the second-order  $\lambda$ -calculus,  $\Lambda X. \lambda x^X. x : \forall X. X \Rightarrow X$ , with anchor  $u$ , as follows.

$$\forall X. \forall x^X. (u^{\forall Y. Y \Rightarrow Y} \bullet [x] \bullet x = x).$$

The assertion says that a type-abstracted behaviour named as  $u$ , when it is applied to any type  $\alpha$  and well-typed argument  $y$  of type  $\alpha$ , will return that argument itself as a result. Similarly we name an existentially pack and open it; or name a stateful procedure and assert how it changes a state and produces a value upon invocation. See (Honda 2004a; Honda and Yoshida 2004) for the account of part of these experiments.

## References

- Abadi, M.,  $\top\top$ -closed relations and admissibility, *MSCS* 10(3) (June 2000), pp. 313-320.
- Abadi, M., Cardelli, L., Curien, P.-L., *Formal Parametric Polymorphism*, *TCS* 121, 1-2, (Dec), 9-58. Elsevier, 1993.
- Abadi, M., Leino, R. A logic for object-oriented programs, pages 682–696, *TAPSOFT'97*, LNCS, 1997.
- Abramsky, S., *Domain Theory in Logical Form*, *Annals of Pure and Applied Logic*, 51:1-77, 1991.
- Abramsky, S., Jagadeesan, R. and Malacaria, P., *Full Abstraction for PCF*. *Info. & Comp.* 163 (2000), 409-470.
- Apt, K.R. Ten Years of Hoare Logic: a survey. *TOPLAS*. 3:431-483, 1981.
- Baeten, J.C.M. and Weijland, W.P. *Process Algebra*. Cambridge University Press, 1990.
- Berger, M., Honda, K. and Yoshida, N., *Sequentiality and the  $\pi$ -Calculus*, *TLCA01*, LNCS 2044, 29–45, 2001.
- Barbanera, F., Dezani-Ciancaglini, M. and de'Liguoro, U., *Intersection and Union Types: syntax and semantics*. *Inf. and Comp.*, 119:202–230, 1995.
- Berger, M., Honda, K. and Yoshida, N., *Genericity and the  $\pi$ -Calculus*, *FoSSaCs'03*, LNCS 2620, 103–119, 2003.
- Bornat, R., *Proving pointer programs in Hoare Logic*. *Mathematics and Program Construction*, 2000.
- Boudol, G., *Asynchrony and the pi-calculus*, INRIA Research Report 1702, 1992.
- Caires, L. and Cardelli, L., *A Spatial Logic for Concurrency (Part I)*. *Info. & Comp.*, 2003.
- Church, A. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- Caml Home Page, <http://caml.inria.fr/>.

- Clint, M. and Hoare C.A.R. Program proving: jumps and functions. *Acta Inf.* 1:214–224, 1971.
- Cousot, P. Methods and logics for proving programs. *Handbook of Theoretical Computer Science*, volume B, 843–993. Elsevier, 1999.
- Curien, P. L., Sequentiality and full abstraction. *Proc. of Application of Categories in Computer Science*, LNM 177, 86–94, Cambridge Press, 1995.
- Curry, B. and Feys, R. *Combinatory Logic*. North-Holland, 1958.
- Dam, M. Proof Systems for  $\pi$ -Calculus Logics. *Logic for Concurrency and Synchronization*. Studies in Logic and Computing, Oxford University Press, 2003.
- Dunfield, J. and Pfenning, F., Type Assignment for Intersections and Unions in Call-by-Value, *FoSSaCs'03*, LNCS 2620, 250–266, Springer, 2003.
- Fiore, M. and Honda, K. Games for Recursive Types, *LICS'98*, IEEE Computer Society Press, 1998.
- Floyd, W. Assigning meaning to programs. *Proc. Symp. in Applied Mathematics*. 19:19–32, 1967.
- Francez N. and Pnueli, A. A proof method for cyclic programs. *Acta Inf.* 9, pp.133–157, 1978.
- Girard, J.-Y., Linear Logic, *TCS*, 50:1–102, 1987.
- Gordon, M., Milner, A. and Wadsworth, C., Edinburgh LCF, LNCS 78, 1979.
- Gunter, C., *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992.
- Grossman, D., et al. Region-Based Memory Management in Cyclone, *PLDI'02*, ACM, 2002.
- Harel, D. Proving the correctness of regular deterministic programs. *TCS*, 12:61–81, 1980.
- Hasegawa, M., Models of Sharing Graphs: A Categorical Semantics of let and letrec, Distinguished Dissertation Series, Springer-Verlag (1999).
- The Haskell home page, <http://haskell.org>.
- Hennessy, M. and Milner, R. Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM*, 32:1, pp.137–161, 1985.
- Hoare, C.A.R. An axiomatic basis of computer programming. *CACM*, 12:576–580, 1969.
- Hoare, C.A.R. Procedures and Parameters: an axiomatic approach. *Lecture Notes in Mathematics* 188, 102–116, *Semantics of Algorithmic Languages*, Springer, 1971.
- Hoare, C.A.R. *Communicating Sequential Processes*, *Comm. ACM* 21, 666–677. 1978.
- Hoare, C.A.R. *Communicating Sequential Processes*, Prentice Hall, 1985.
- Hoare, C.A.R. and Wirth, N., An axiomatic definition of the programming language PASCAL. *Acta Inf.* 2:335–355, 1973.
- Honda, K., Composing Processes, *POPL'96*, 344–357, ACM, 1996.
- Honda, K. *Process Logic and Duality: Part (1) Sequential Processes*. Typescript, 223pp. March 2004. Available at: [www.dcs.qmul.ac.uk/~kohei/logics](http://www.dcs.qmul.ac.uk/~kohei/logics).
- 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](http://www.dcs.qmul.ac.uk/~kohei).
- Honda, K. From Process Logic to Program Logic. *ICFP'04*, 2004, ACM Press.
- Honda, K. and Tokoro, M. An object calculus for asynchronous communication. *ECOOP'91*, LNCS 512, 133–147, 1991.
- Honda, K. and Yoshida, N., A Compositional Logic for Polymorphic Higher-Order Functions. *PPDP'04*, ACM, 2004.
- Honda, K., Yoshida, N. and Berger, M., A Compositional Program Logic for Imperative Higher-Order Functions. Typescript, 49 pp, November, 2004.
- Honda, K. and Yoshida, N. On Reduction-Based Process Semantics, *TCS*, 151, 1995.
- Honda, K. and Yoshida, N. Game-theoretic analysis of call-by-value computation. *TCS*, 221:393–456, 1999.
- Honda, K. Elementary Structures for Process Theory (1): Sets with Renaming, *MSCS*, 2001:50–54, CUP.

- Honda, K. and Yoshida, N., A Uniform Type Structure for Secure Information Flow, *POPL'02*, 81–92, ACM, 2002.
- Honda, K. and Yoshida, N., Noninterference proofs from Flow Analysis. To appear in: *Journal of Functional Programming*, CUP, 2004.
- Honda, K., Yoshida, N. and Berger, M., Control in the  $\pi$ -Calculus, *Proc. CW'04*, ACM, 2004.
- 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.
- Hyland, M. and Ong, L., "On Full Abstraction for PCF": I, II and III. *Info. & Comp.* 163 (2000), 285–408.
- Jacobs, B. et al. Reasoning about Java Classes (Preliminary Report). *OOPSLA'98*, ACM, 1998.
- Java home page. Sun Microsystems Inc., <http://www.javasoft.com/>, 1995.
- Jones, C.B. Specification and Design of (Parallel) Programs. *Proc. IFIP 9th World Computer Congress*. North Holland, 321–332, 1983.
- Kleymann, T. Hoare Logic and Auxiliary Variables, University of Edinburgh, LFCS ECS-LFCS-98-399, 1998.
- Kobayashi, N., Pierce, B., and Turner, D., Linear types and the  $\pi$ -calculus, *TOPLAS*, 21(5):914–947, 1999.
- Kowaltowsky, T. Axiomatic approach to side effects and general jumps. *Acta Informatica* 7, 357–360., 1977.
- Landin, P., A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. *Comm. ACM*, 8, 2 (1965), 1965.
- Laurent, O., Polarized games, *LICS 2002*, 265–274, IEEE, 2002.
- Leavens, G. and Baker, A. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. *FM'99: World Congress on Formal Methods*, Springer, 1999.
- Longley, J. and Plotkin, G. Logical Full Abstraction and PCF. *Tbilisi Symposium on Logic, Language and Information*. pp.333–352, Stanford, CSLI, 1998.
- Mendelson, E., *Introduction to Mathematical Logic* (third edition). Wadsworth Inc., 1987.
- Pitts, A.M., Existential Types: Logical Relations and Operational Equivalence, *Proceedings ICALP'98*, LNCS 1443, 309–326, Springer, 1998.
- Pitts, A.M., Parametric Polymorphism and Operational Equivalence, *Mathematical Structures in Computer Science*, 2000, 10:321–359.
- Milner, R., Processes: a mathematical model of computing agents. *Logic Colloquium '73*, North Holland, 1973.
- Milner, R., Fully Abstract Models of Typed Lambda-Calculi, *Theoretical Computer Science*, volume 4 (1977), 1–22, 1997.
- Milner, R., *A Calculus of Communicating Systems*, LNCS 92, Springer, Berlin, 1980.
- Milner, R., *Communication and Concurrency*, Prentice Hall, 1989.
- Milner, R., Functions as Processes, *MSCS*. 2(2):119–141, 1992,
- Milner, R., Polyadic  $\pi$ -Calculus: a tutorial. *Proceedings of the International Summer School on Logic Algebra of Specification*, Marktobendorf, 1992.
- 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](http://www.cs.unibo.it/icalp/Lauree_milner.html).
- Milner, R., Tofte, M. and Harper, R.W., *The Definition of Standard ML*, MIT Press, 1990.
- Milner, R., Parrow, J. and Walker, D., A Calculus of Mobile Processes, *Info. & Comp.* 100(1):1–77, 1992.
- Milner, R., Parrow, J.G. and Walker, D.J., Modal logics for mobile processes, *TCS*, Vol 114, pp.149–171, 1993.

- 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/>.
- 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.
- Morrisett, G. et al. Cyclone: A Safe Dialect of C. Usenix Annual Technical Conference, Monterey, CA, June 2002.
- Microsoft Corporation, .NET Framework Developer's Guide, <http://msdn.microsoft.com>, 2003.
- Moggi, E., Notions of Computations and Monads. *Information and Computation*, 93(1):55–92, 1991.
- Naur, P. et al. Revised Report on the Algorithmic Language Algol 60. *Comm. ACM*, 6, 1 (1963).
- Naur, P. Proof of algorithms by general snapshots. *BIT* 6, pp.310–316, 1966.
- Nickau, M., Hereditarily Sequential Functionals, LNCS 813, pp.253–264, Springer-Verlag, 1994.
- Oheimb, D.V. Hoare Logic for Java in Isabelle/HOL. *Concurrency: Practice and Experience*. John Wiley, 2002.
- O'Hearn, P., Yang, H. and Reynolds, J., Separation and Information Hiding, POPL'04, 268–280, 2004.
- Pierce, B.C., *Types and Programming Languages*, MIT, 2002.
- Pierce, B.C. and Sangiorgi, D., Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.
- Plotkin, G., Call-by-name, Call-by-value and the lambda-calculus, *TCS*, 1:125–159, 1975.
- Plotkin, G. and Abadi, M., A Logic for Parametric Polymorphism, *LICS'98*, 42–53, IEEE Press, 1998.
- Poetzsh-Heffter, A. and Muller, P. A programming logic for sequential Java. ESOP'99, LNCS 1576, pp.162-176, Springer, 1999,
- Pratt, V.R., Six Lectures on Dynamic Logic, Foundations of Computer Science III, Part 2, 53-82, Mathematical Centre Tracts 109, Amsterdam, 1980.
- Reynolds, J. Intuitionistic Reasoning about Shared Mutable Data Structure, Millennial Perspectives in Computer Science, Oxford, 1999.
- Reynolds, J.C., Separation logic: a logic for shared mutable data structures. Invited Paper, *LICS*, 2002.
- Sangiorgi, D.,  $\pi$ -calculus, internal mobility, and agent-passing calculi. *TCS*, 167(2):235–271, 1996.
- Scott, D. and Strachey, C., Towards a mathematical semantics for computer languages, Proceeding of 21st Symposium on Computers and Automata, 19–46, Polytechnic Institute of Brooklyn, 1971.
- Jones, C.B. A Software Development: A Rigorous Approach. Prentice Hall, 1980.
- Wadler, P. *Theorems for Free*. Functional Programming and Computer Architecture, Addison Wesley, 1989.
- Walker, D., Objects in the  $\pi$ -calculus. *Info. Comp.*, 116(2):253-271, 1995.
- Whitehead, A. N. and Russell, B., *Principia Mathematica*, I–III, 1910–1913.
- Winskel, G. *The Formal Semantics of Programming Languages*. MIT Press, 1993.
- Yoshida, N., Graph Types for Monadic Mobile Processes. Proc. of 16th FST/TCS, LNCS 1180, 1996.
- Yoshida, N., Berger, M. and Honda, K., Strong Normalisation in the  $\pi$ -Calculus, *LICS'01*, 311–322, IEEE, 2001.
- Yoshida, N., Honda, K. and Berger, M. Linearity and Bisimulation, FoSSaCs 2002, LNCS 2303, 417–433, 2002.
- Woodcock, J. and Davies, J. *Using Z: specification, refinement and proof*. Prentice-Hall International, 1996.

## A Omitted Proofs

### A.1 Proposition 1

By observing any process typable in the present system is typable by the system in (Berger et al. 2001) and, conversely, any process typable by the system in (Berger et al. 2001) with a sequential action type is typable in the present system up to  $\equiv$ . For the latter we show, by rule induction, that any typable process in (Berger et al. 2001) is a parallel composition of processes typable by the above process, which is mechanical (the need to place  $\equiv$  is because the use of sequential action types makes  $\equiv$  not closed under associativity of parallel composition, e.g.  $(\nu yz)(x.\bar{y} \mid y.\bar{z} \mid z.\bar{w})$  is typable by  $x : ()^\downarrow, w : ()^\uparrow$  when regarded as  $(x.\bar{y} \mid y.\bar{z}) \mid z.\bar{w}$  but not so when regarded as  $(x.\bar{y} \mid z.\bar{w}) \mid y.\bar{z}$ ).

## B Restriction on Summands of Infinitary Branching

In the following we briefly discuss restriction we need for summands of infinitary branching. It suffices to consider only sequential affine processes, which makes the description of restrictions more concise.

### B.1 Finite Bounds

First, in each branching  $x[\&_{i \in I}(\bar{y}_i).P_i]$ , we assume there is a finite bound on the length of  $\bar{y}_i$ , a finite bound on the size of  $P_i$ , and a finite set of free names which is a superset of the free names of  $P_i$  for each  $i$ . This is to avoid an anomaly where, for example, the branching itself has an unbounded size even if each summand is of a finite size.

### B.2 Computability

Second, for the full definability result necessary for logical full abstraction (so that all processes, not only finite ones, become the image of the encoding, cf. §5.2), we assume that indices of selections under  $\mathbb{N}$ -indexed branches are specified by a computable total function. For example, in  $x[\&_{i \in \mathbb{N}} \bar{y}_i \text{in}_{F(i)}]$ , we demand  $F(i)$  to be computable. In essence,

In the following we formally stipulate this restriction, after a brief discussion of its background. We also refer to Part III where Hyland and Ong stipulated how interactions (transitions) can be restricted so that the class of behaviours only represent computable ones (we conjecture the following construction generates precisely those processes whose typed transitions coincide with strategies in (Hyland and Ong 2000)).

#### B.2.1 Simple Method.

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 goal.

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 (as in, for example, PCF). Combined with recursion, this directly leads to the class of computable processes we want. 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.

### B.2.2 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  $\bar{u} \text{in}_j$ , the *index* of  $u$  is  $j$ . By affine typing we know each  $P_i$  has the form:

$$\text{either } \bar{y}(\vec{z}w)P'_i \quad \text{or} \quad \bar{u} \text{in}_j(\vec{w})P'_i.$$

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:

$$x[\&_{i \in \mathbb{N}} \bar{y}(z)z[\&_{j \in \mathbb{N}} \bar{u} \text{in}_{i \times j}]].$$

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  $\llbracket P \rrbracket$  together with the notion of computable processes inductively as follows.

1.  $x[\&_{i \in \mathbb{N}} \bar{u} \text{in}_{f(i)}]$  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  $i \mapsto \llbracket P_i \rrbracket$  is given by a computable total function, where  $\llbracket P_i \rrbracket$  is given by induction. The resulting function is its encoding.
3.  $\bar{x}(\vec{y}z)(\vec{R}^y | P^z)$  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.  $(\nu x)P$  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. It is easy to see that, as far as processes are computable in the above sense, affine processes of suitable shape and types represent precisely those processes whose behaviours coincide with the image of the encoding of PCFv-terms.

### C Restriction to Linearity for Atomic Types

We restrict inhabitants of  $\mathbb{B}^\circ$  and  $\mathbb{N}^\circ$  to total behaviours. While this can be uniformly done using the linear typing discipline extensively discussed in (Yoshida et al. 2001), it suffices to use the following specific typing rules for these two atomic types. Their duals need no restriction.

$$\begin{array}{c}
 \text{(Bool)} \\
 \frac{-}{\vdash !x(c).\bar{c}\text{in}_{\mathbf{b}} \triangleright x : \mathbb{N}^\circ}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{(Nat)} \\
 \frac{-}{\vdash !x(c).\bar{c}\text{in}_{\mathbf{n}} \triangleright x : \mathbb{N}^\circ}
 \end{array}$$

Note these rules prohibit suppressing a linear output by a linear input. This makes these outputs truly linear.