From Process Logic to Program Logic

Kohei Honda
Department of Computer Science
Queen Mary, London
kohei@dcs.qmul.ac.uk

Abstract

We present a process logic for the π-calculus with the linear/affine type discipline [6, 7, 31, 32, 33, 59, 60]. 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 [27, 26].

Categories and Subject Descriptors

D.3.1 [**Formal Definitions and Theory**]: Semantics; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Assertions, Logics of programs, Specification techniques; F.3.2 [**Semantics of Programming Lanugages**]: Process models

General Terms

Language, Theory, Verification

Keywords

Types, Mobile Processes, Higher-Order Functions, Hoare Logic, Duality, π-Calculus

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 [21], Hoare Logic, developed on the basis of earlier work by Floyd [16] and Naur [49]. 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 [4, 11]). 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. [51, 35, 55, 40].

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

ICFP'04, September 19–21, 2004, Snowbird, Utah, USA.

functionalities of other software. A Java program relies on its APIs which are implemented using native code and OS libraries, an OS library uses kernel calls, and even a kernel relies on, for example, the proper working of the network for carrying out its essential functionalities. Thus reasoning about behaviour of software across language boundaries, but still on a rigorous logical basis, is fundamental for guaranteeing the safe behaviour of software. In spite of many studies on Hoare Logic and its derivatives, it seems this topic has not been investigated as extensively as the subject may deserve.

We present in this paper one possible approach to this problem domain. The idea is to develop compositional logics for a tiny formalism of interaction, and use it to derive, analyse and mediate, via encoding of language constructs in it, logics for more complex programming languages. This tiny formalism — which is a minimal core of a process calculus, the π-calculus, introduced by Milner, Parrow and Walker [46] — has a simple operation, *communication of names*, and a small set of algebraic operators, including *parallel composition*. The π-calculus is a calculus of concurrency, developed along the line of such process calculi as CCS [43], CSP [23] and ACP [5]. But, as the originators noted [46], 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 [45]. In fact, if we constrain the composition of processes using a class of type annotations (which come from Linear Logic [18, 39], Game Semantics [3, 30, 34] and study of types for the π-calculus [48, 53, 37, 25]), we can embed a wide class of programming languages into name passing processes without losing essential semantic properties [6, 59, 60, 31, 7, 32, 33]. If, therefore, we can find a suitable notion of logic for typed name passing processes together with its sound compositional proof rules, we may as well be able to derive logics for different programming languages via encoding; and, in turn, the typed π-calculus would serve as a common stratum of these program logics, allowing the combined reasoning for multiple languages.

Thus our process logic starts from typed processes. Following Hennessy-Milner Logic [20], 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 π-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 [27]. 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, together with extensions of the framework to the realm of behaviours beyond those treated so far, which include, among others, concurrency and distribution.

1.1 Possible Technical Contributions

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

1. The finding that duality-based types (originating in the foregoing studies on types and semantics of interaction) leads to a highly structured form of Hennessy-Milner logic for the π-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 π-calculi to derive, analyse and verify program logics for these languages. This extends various semantic embeddings of programming languages in processes [44, 58, 6, 59] 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

As may be clear from the preceding discussions, the present work owes much to the three research strands of semantics of computation: compositional program logics, theories of processes, and types for functions and interaction. In the following we discuss those works directly related with the material presented in this short paper: more comprehensive discussions are found in [27].

Among extensions of Hoare's axiomatic method, the proof rules for procedures studied by Hoare and Clint [22, 10] predict the logical treatment of replicated processes in the present work. Kowaltowsky [38] proposes a way to use names to express return values of procedures. Abadi and Leino [2] propose 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 [36] 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 [17]). We owe to Jones' work the idea of representation of the environments' behaviour in logical specifications of programs, as recorded in the shape of the sequent in the present process logic.

In a tradition different from Hoare Logic, equational logics for the λ-calculi have been studied since the classical work by Church, Curry and others. LCF [14] augments the standard equational theory of the λ-calculus with Scott's fixed point induction. Other prominent logics along this line include those for polymorphic λ-calculi [54, 1]. The program logics for higher-order functions derived from process logics differ in that their assertion describes behavioural properties of programs rather than equates programs, allowing specifications with arbitrary degrees of precision as well as smoothly extending to non-functional behaviour (for further technical comparisons see [27, §1.2]).

Hennessy-Milner Logic [20], one of the starting points of the present work, enjoys a sound and complete equational characterisation for bisimilarity. Similarly the present logic enjoys the same property for the typed contextual congruence. The embeddability of a program logic in a process logic is first demonstrated by Milner for a basic specification logic for shared variable concurrency [43]. Existing studies on logics for the π-calculus include Hennessy-Milner Logic [47] (for characterising different bisimilarities) and Spatial Logic [9] (for capturing spatial structures of processes). Dam's survey [13] details the difficulties in developing the proof system of Hennessy-Milner Logic for the untyped π-calculus. All these preceding 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.

1.3 Structure of the paper

In the remainder, Section 2 introduces the basic ideas of the typed process logic, including assertions, their semantics, and proof rules. Section 3 derives a compositional program logic for call-by-value PCF [15] from the process logic, outlines how a fully abstract logical embedding leads to a simple proof of soundness of the proof rules, and concludes with simple inference examples. Detailed proofs as well as experiments of the framework for a wide range of sequential processes and programs are found in [27, 26, 28].

1.4 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).

Acknowledgement. The author thanks Martin Berger, Adriana Compagnoni, Masahito Hasegawa, Alex Simpson, Nobuko Yoshida and anonymous referees for enriching discussions, communications and comments. The author is partially supported by EPSRC grant GR/S55545.

2. Typed Process Logic

2.1 Processes

The π-calculus used in this paper is a typed variant of the standard asynchronous π-calculus [29, 8]. The following gives the reduction rule of the asynchronous π-calculus.

$$x(\vec{y}).P \,|\, \overline{x}\langle\vec{v}\rangle \longrightarrow P[\vec{v}/\vec{y}] \tag{1}$$

Here $\vec{y}$ denotes a potentially empty vector $y_1...y_n$ of names, $|$ denotes parallel composition, $x(\vec{y}).P$ is an input, and $\overline{x}\langle\vec{v}\rangle$ 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, $!x(\vec{y}).P$:

$$!x(\vec{y}).P \,|\, \overline{x}\langle\vec{v}\rangle \longrightarrow !x(\vec{y}).P \,|\, P[\vec{v}/\vec{y}]. \tag{2}$$

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 [57]. Using bound name passing leads to simple proof rules without sacrificing expressiveness. Syntactically we restrict outputs to the form $(\nu\vec{y})(\overline{x}\langle\vec{y}\rangle|P)$ (where $(\nu\vec{y})$ indicate name hiding: names in $\vec{y}$ should be pairwise distinct), which we henceforth write $\overline{x}(\vec{y})P$ (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.

$$\begin{array}{lcl} x(\vec{y}).P \,|\, \overline{x}(\vec{y})Q & \longrightarrow & (\nu\vec{y})(P \,|\, Q) \\ !x(\vec{y}).P \,|\, \overline{x}(\vec{y})Q & \longrightarrow & !x(\vec{y}).P \,|\, (\nu\vec{y})(P \,|\, Q) \end{array}$$

"$\overline{x}(\vec{y})Q$" indicates that $\overline{x}(\vec{y})$ 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* $\overline{x}\mathtt{in}_i\langle\vec{w}\rangle$, the latter written $\overline{x}\mathtt{in}_i(\vec{w})P$ in the bound output notation. These constructs are used for representing base values, conditionals and the case construct [6, 59, 18]. They play a basic role in the typed setting, just as the case/injection constructs in the typed λ-calculus with sums. The dynamics of branching/selection is given by the following reduction rule.

$$x[\&_{i\in I}(\vec{y}_i).P_i] \,|\, \overline{x}\mathtt{in}_j(\vec{y}_j)P \longrightarrow (\nu\vec{y}_j)(P_j \,|\, P)$$

The reduction involves selection of one branch, discarding remaining ones, combined with name passing.

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

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

We often omit the index set I in $x[\&_{i\in I}(\vec{y}_i).P_i]$. In the present paper we only consider $\{*\}$ (a distinguished singleton), $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$, and $\mathbb{N} = \{0, 1, \ldots\}$ as the indexing sets for branching.[1] When the indexing set is $\{*\}$, the branching is written $x(\vec{y}).P$, regaining the non-replicated unary prefix. $P|Q$ 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, $\overline{x}$ for $\overline{x}()$, $x.P$ for $x().P$, $x[\&_i .P_i]$ for $x[\&_i ().P_i]$, and $\overline{x}\mathtt{in}_i$ for $\overline{x}\mathtt{in}_i()$.

The structural congruence $\equiv$ (which includes the α-equality) is standard [59]. The reduction relation $\longrightarrow$ is generated starting from the rules already given, closing the relation under contexts except under 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 *affine type discipline* [6] for constraining their behaviour to purely functional ones. We use the following *action modes* [6, 59], which represent different modes of communication actions at channels.

$\downarrow$	Affine input	$\uparrow$	Affine output
!	Replicated server	?	Client request to !

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 [18]). 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.

[1] For infinitary branching, we assume a suitable restriction on the use of indices of selections (detailed in [27, Appendix A]) so that processes represent only computable functions.

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

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

$(\vec{\tau})^!$ 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.

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

A classification of channel types called *polarities* [32, 39] plays a basic rôle 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. $?, \downarrow$). $\updownarrow$ 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 [32]; 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 Γ, or P is typed under Γ). The typing rules are presented and illustrated in Appendix A: however the following technical development should be understood without details of the typing rules. A few examples of typed processes follow.

1. The process $\overline{w}\mathtt{in}_3$ simply selects the third branch at w, for which we have $\vdash \overline{w}\mathtt{in}_3 \triangleright w : [\oplus_{i \in \mathbb{N}}]^{\uparrow}$.

2. Let $[\![3]\!]_x \stackrel{\text{def}}{=} !x(w).\overline{w}\mathtt{in}_3$. Then $\vdash [\![3]\!]_x \triangleright x : \mathbb{N}^\circ$ where $\mathbb{N}^\circ = ([\oplus_{i \in \mathbb{N}}]^{\uparrow})^!$. The process, when invoked at x with a single name, immediately outputs 3 via that name.

3. Let $\mathsf{double}_{xy} \stackrel{\text{def}}{=} !x(w).\overline{y}(u)u[\&_{n \in \mathbb{N}}.\overline{w}\mathtt{in}_{n \times 2}]$. Then we have $\vdash \mathsf{double}_{xy} \triangleright y : \overline{\mathbb{N}^\circ}, x : \mathbb{N}^\circ$. 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 double_{xy} interact as:

$$[\![3]\!]_y \,|\, \mathsf{double}_{xy} \,|\, \overline{x}(w)w[\&_i \overline{f}\mathtt{in}_i] \longrightarrow^+ [\![3]\!]_y \,|\, \mathsf{double}_{xy} \,|\, \overline{f}\mathtt{in}_6.$$

4. Let $[x \to y]^{\tau} \stackrel{\text{def}}{=} !x(w).\overline{y}(u)u.\overline{w}$ with $\tau \stackrel{\text{def}}{=} (()^{\uparrow})^!$. Then we have $\vdash [x \to y]^{\tau} \triangleright x : \tau, y : \overline{\tau}$. This is a *copycat of type* τ [34, 6], which acts as a transparent link between two locations, as the following reduction indicates (below $\approx$ is the standard weak bisimilarity [46]).

$$[x \to y]^{\tau} \mid \overline{x}(w)P \quad \longrightarrow^+ \approx \quad [x \to y]^{\tau} \mid \overline{y}(w)P$$

Similarly we can define a copycat for general $!/\downarrow$-types, as formally given in Appendix B.

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

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

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

2.2 Assertions for Typed Processes

2.2.1 Basic Ideas

Processes are inherently interactional, and, for that reason, their behaviour is dependent on the environment's behaviour. Recall double_{xy} above. If we invoke this process at x, and if $!y(c).\overline{c}\mathtt{in}_3$ is present, then double_{xy} will return 6: but if at y there is $!y(c).\overline{c}\mathtt{in}_1$ then it answers with 2. Thus a process in general can guarantee a certain behaviour only by relying on the behaviour of an environment [36], arriving at the following form of judgement:

$$P^{\Gamma} \vdash \ \textbf{rely} \ A \ \textbf{guar} \ B \tag{3}$$

where A and B are formulae whose grammar is given later. Negative names in Γ may occur in A (but not in B), while positive names in Γ may occur in B (but not in A). The judgement can be read as:

> *If P typed under Γ can rely on A as the behaviour of an environment, then the process combined with the environment can guarantee B.*

We often write $P \vdash \textbf{rely} A \, \textbf{guar} B$, leaving the action type implicit.

As a concrete example, we can specify the behaviour of double_{xy} as follows. Below we assume the action type $y : \overline{\mathbb{N}^\circ}, x : \mathbb{N}^\circ$. Below the symbol ε stands for the empty vector of names.

$$\mathsf{double}_{xy} \vdash \ \textbf{rely} \ y \bullet \varepsilon = \mathtt{in}_3() \ \textbf{guar} \ x \bullet \varepsilon = \mathtt{in}_6() \ . \tag{4}$$

The term $y \bullet \varepsilon$, which consists of a name y, operator $\bullet$, and the argument ε, is best understood as abstraction of an output of form $\overline{y}(c)$, with ε indicating y carries no arguments except for a return channel. $\mathtt{in}_3()$ can be understood as abstraction of an output $\overline{c}\mathtt{in}_3$. Thus $y \bullet \varepsilon = \mathtt{in}_3()$ says that the invocation $\overline{y}(c)$ is answered by an output $\overline{c}\mathtt{in}_3$. The bound c is not mentioned since its concrete naming is insignificant. Thus $\bullet$ indicates typed interaction: in the next section we encode function application into this operation. Reading the second formula in the same way, (4) as a whole says that, if the environment returns 3 when invoked at y with the null argument, then the process itself would return 6 when invoked similarly at x. How such formulae arise from those in Hennessy-Milner logic is discussed in [27, Section 2].

The next assertion is more detailed than (4).

$$\mathsf{double}_{xy} \vdash \textbf{rely} \ y \bullet \varepsilon = \mathtt{in}_i() \ \textbf{guar} \ x \bullet \varepsilon = \mathtt{in}_{2 \times i}() \tag{5}$$

Since i does not occur in Γ, i is auxiliary [22]. As in Hoare Logic, auxiliary names are universally quantified in the whole sequent. Thus (5) says that, for each $i \in \mathbb{N}$, if the environment is such that it returns i for its invocation at y, then P returns $2 \times i$.

Now consider the following process which slightly alters double_{xy}.

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

Note y is now bound. For this agent, the assertion corresponding to (5) becomes, under the typing $x : (\overline{\mathbb{N}^\circ}[\oplus_{i \in \mathbb{N}}]^{\uparrow})^!$:

$$\mathsf{double'}_x \vdash \ \textbf{rely} \ \mathsf{T} \ \textbf{guar} \ y \bullet \varepsilon = \mathtt{in}_i() \supset x \bullet y = \mathtt{in}_{2 \times i}().$$

$x \bullet y$ indicates x is now invoked with a single argument (together with an implicit return channel). Since y is auxiliary, we may rewrite the above rely formula as $\forall y, i. (y \bullet \varepsilon = \mathtt{in}_i() \supset x \bullet y = \mathtt{in}_{2 \times i}())$.

2.2.2 Terms

Terms in the logic are given by the following grammar. Below let τ be of mode ! or $\uparrow$, and ρ be of mode $\uparrow$.

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

The first set of terms $(e, e', \ldots)$ are *data expressions*, while the second set $(a, a', b, b', \ldots)$ *behaviour expressions*. $T, T', ..$ denotes atomic data sets ($\{*\}$, $\mathbb{B}$ and $\mathbb{N}$). Data expressions denote their elements, while behaviour expressions abstract process interactions. The well-typedness of data/behaviour expressions is naturally given, setting the type of $a \bullet \vec{b}$ to be $\tau^{\uparrow}$ assuming a has type $(\vec{\overline{\rho}}\tau)^{!}$ and $\vec{b}$ has type $\vec{\rho}$ (for the full typing rules, see Appendix C). We usually omit ρ from $\mathtt{in}_e^{\rho}(\vec{a})$, and write $(\vec{a})^{\uparrow}$ for $\mathtt{in}_*(\vec{a})$ for brevity.

2.2.3 Formulae

Formulae are those of the first-order logic with equality [42]. 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 T (which is, say, $1 = 1$) and the falsity F (which is $\neg\mathsf{T}$). The quantifications induce binding, for which we assume the standard bound name convention. $\mathsf{fn}(A)$ denotes free names in A. A is *well-typed* if whenever a variable/name occurs twice they own the same type and each pair of equated terms have the same type. *We only consider well-typed formulae from now on.*

2.2.4 Judgement

A judgement for a typed process $\vdash P \triangleright \Gamma$ has the following shape.

$$P^{\Gamma} \models \textbf{rely } A \textbf{ guar } B\,. \tag{6}$$

A (resp. B) is the *rely formula* (resp. the *guarantee formula*) of the judgement. Those names in A and B which are from Γ are called *primary names*, others *auxiliary names*. Assuming $\Gamma = \Delta_1, \Delta_2$ such that types in Δ_1 (resp. Δ_2) are negative (resp. positive/neutral), the free names in A (resp. in B) should be typed under $\overline{\Delta_1}$ (resp. under Δ_2), combined with a typing on auxiliary names, common to A and B. Note free names in formulae in each judgement are strictly typed by the underlying action type. Also note neutral names, i.e. those typed by $\updownarrow$, never occur in formulae (since a name in a formula is always either positive or negative). This indicates logical insignificance of neutral names (which reflects their behavioural insignificance: no visible action is possible at neutral names). An assertion is sometimes written $P \models \textbf{rely} A \textbf{ guar} B$, leaving the action type implicit. An *assertion* means either a judgement or formulae occurring in a judgement.

2.2.5 Total and Partial Correctness

In the following we present simple semantics of assertions. Before starting, however, a note is due on the treatment of divergence in specification. As in Hoare logic, an assertion in the present typed process logic can be either about total correctness (where non-trivial assertions always guarantee termination) or about partial correctness (where divergence allows arbitrary specifications). These two, together with a third one which subsumes both, are discussed in detail in [27]. In the present paper we focus on the logics for total correctness because of its simplicity in presentation.

Semantics of assertions uses an abstract notion of behaviour, defined from the standard typed congruence for affine processes [6]. The congruence, written $\cong_\pi$, is the maximum typed congruence satisfying: $\vdash P \cong_\pi Q \triangleright x\!:\!()^{\uparrow}$ iff $P \Downarrow_x \equiv Q \Downarrow_x$, where $\vdash P_1 \cong_\pi P_2 \triangleright \Gamma$ means $\vdash P_{1,2} \triangleright \Gamma$ are related by $\cong_\pi$ and $P \Downarrow_x$ means $P \longrightarrow^* \overline{x}|R$

2.2.6 Model

Let τ be positive. An *abstract process* p of type τ, often written p^{τ}, is a map from the names to the $\cong_\pi$-congruence classes such that: (1) $P \in p(x)$ implies $\vdash P \triangleright x\!:\!\tau$, 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* ξ is a finite map from names to abstract processes. ξ *has type* Γ iff $\xi = x_1\!:\!p_1^{\tau_1}, .., x_n\!:\!p_n^{\tau_n}$ and $\Gamma = x_1\!:\!\tau_1, .., x_n\!:\!\tau_n$. ξ^{Γ} indicates ξ has type Γ. A model ξ 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.

2.2.7 Satisfaction Relation

Fix auxiliary names in A and an interpretation I, the latter mapping the former to constants and abstract processes. The interpretation $[\![e]\!]I$ is standard, starting from $[\![x]\!]I = I(x)$. For behaviour expressions, the interpretation $[\![a]\!]\,I \cdot \xi$ is given by:

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

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

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

In these clauses we confuse an abstract process with a concrete process in its $\cong_\pi$-congruence class. This does not lead to ambiguity because of congruency and renaming closure.

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

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

Logical connectives are interpreted classically:

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

where, in the last line, $[\![\tau]\!]$ denotes the set of all abstract processes of type τ. The rest is de Morgan duality.

2.2.8 Semantics of Judgement

Let $\vdash P \triangleright \Gamma$. We write $P \models \xi^{\Gamma}$ with $\xi = x_1 : p_1, .., x_n : p_n$ to denote $\vdash P \cong_\pi \Pi_i P_i \Gamma$ such that $P_i \in p_i(x_i)$ for each i. Fixing auxiliary names in A and its interpretation I, let us write $P \models^I A$ when $P \models \xi$ and $\xi \models^I A$ for some total ξ (the restriction to total models is for ensuring total correctness: I is left unconstrained). We can now formally define the semantics of assertion.

Definition 1 *Assume $\vdash P \triangleright \Gamma, \Delta$ where Γ is negative and Δ is positive/neutral. Then we write $P^{\Gamma,\Delta} \models \textbf{rely} A \textbf{ guar} B$ when, for each I and $\vdash R \triangleright \overline{\Gamma}$, $R \models^I A$ implies $(\nu\,\mathsf{fn}(\Gamma))(P|R) \models^I B$.*

A basic property of the typed process logic follows, which says that assertions valid for each process precisely characterise its behaviour up to the contextual congruence. Below $P^{\Gamma} =_{\mathcal{L}} Q^{\Gamma}$ stands for $\forall A, B. (\ P^{\Gamma} \models \textbf{rely} A \textbf{ guar} B \ \equiv \ Q^{\Gamma} \models \textbf{rely} A \textbf{ guar} B\ )$.

Proposition 1 *$P^{\Gamma} =_{\mathcal{L}} Q^{\Gamma}$ iff $\vdash P \cong_\pi Q \triangleright \Gamma$.*

See [27, §3.8, §4.4] for the proof. Though this property does not directly pertain to expressiveness of assertions, it does confirm that specifications in the present logic speak no more and no less than observable properties of processes.

(Zero)

$$\dfrac{-}{\mathbf{0} \vdash \textbf{rely}\, A\, \textbf{guar}\, A}$$

(Par)

$$\dfrac{\begin{array}{l} P \vdash \textbf{rely}\, A_1\, \textbf{guar}\, B_1 \wedge C \\ Q \vdash \textbf{rely}\, A_2 \wedge C\, \textbf{guar}\, B_2 \end{array}}{P|Q \vdash \textbf{rely}\, A_1 \wedge A_2\, \textbf{guar}\, B_1 \wedge B_2}$$

(Rec)

$$\dfrac{\begin{array}{l} P \vdash \textbf{rely}\, A^{\text{-}\vec{y}}\, \textbf{guar}\, B(0)^{\vec{x}} \\ P \vdash \textbf{rely}\, A \wedge B(i)[\vec{y}/\vec{x}]\, \textbf{guar}\, B(i+1) \end{array}}{\mu \vec{y}^{\vec{\tau}} = \vec{x}.P \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

(Res)

$$\dfrac{P \vdash \textbf{rely}\, A\, \textbf{guar}\, B^{\text{-}x}}{(\nu x)P \vdash \textbf{rely}\, A\, \textbf{guar}\, B}$$

(Weak)

$$\dfrac{P^{\Gamma} \vdash \textbf{rely}\, A\, \textbf{guar}\, B}{P^{\Gamma, x:\tau} \vdash \textbf{rely}\, A\, \textbf{guar}\, B}$$

(Bra$^{\downarrow}$)

$$\dfrac{\forall i.\, P_i \vdash \textbf{rely}\, A[\mathtt{in}_i(\vec{y}_i)/x]\, \textbf{guar}\, B}{x[\&_i(\vec{y}_i).P_i] \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

(Sel$^{\uparrow}$)

$$\dfrac{P \vdash \textbf{rely}\ A\ \textbf{guar}\ B[\mathtt{in}_i(\vec{y})/x]}{\overline{x}\mathtt{in}_i(\vec{y})P \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

(In$^{!}$)

$$\dfrac{\begin{array}{l} \forall \vec{y}.\, (C_1 \supset C_2[x \bullet \vec{y}/z]) \supset B \\ P \vdash \textbf{rely}\ A^{\text{-}\vec{y}} \wedge C_1^{\vec{y}}\ \textbf{guar}\ C_2 \end{array}}{!x(\vec{y}z).P \vdash \textbf{rely}\ A\ \textbf{guar}\ B}$$

(Out$^{?}$)

$$\dfrac{\begin{array}{l} A \supset C_1 \supset C_2[x \bullet \vec{y}/z] \wedge x \bullet \vec{y} \Downarrow \\ P \vdash \textbf{rely}\ A^{\text{-}z} \wedge C_2^{z}\ \textbf{guar}\ C_1^{\vec{y}} \wedge B^{\text{-}\vec{y}} \end{array}}{\overline{x}(\vec{y}z)P \vdash \textbf{rely}\, A\ \textbf{guar}\ B}$$

Figure 1: Proof Rules for Processes

(Consequence)

$$\dfrac{\begin{array}{l} A \supset A',\ B' \supset B \\ P \vdash \textbf{rely}\, A'\, \textbf{guar}\, B' \end{array}}{P \vdash \textbf{rely}\, A\, \textbf{guar}\, B}$$

(Aux-$\forall$)

$$\dfrac{P \vdash \textbf{rely}\, A^{\text{-}i}\, \textbf{guar}\, B}{P \vdash \textbf{rely}\, A\, \textbf{guar}\, \forall i.B}$$

(Aux-$\exists$)

$$\dfrac{P \vdash \textbf{rely}\, A\, \textbf{guar}\, B^{\text{-}i}}{P \vdash \textbf{rely}\, \exists i.A\, \textbf{guar}\, B}$$

Figure 2: Structural Rules

2.3 Proof Rules for Process Logic

One of the significant consequences of the representation of environments in assertions in the present logic is the existence of simple compositional proof rules for valid assertions. We use the judgement of the form $P^{\Gamma} \vdash \textbf{rely}\, A\, \textbf{guar}\, B$ for provability, which should follow the same typing constraint on primary/auxiliary names in formulae as before (action types are often omitted). The proof rules are given in Figures 1 and 2. In each rule, we assume all occurring processes and assertions, including those in the conclusion, are well-typed. Further conventions:

- $A^{\text{-}\vec{y}}$ indicates no name in $\vec{y}$ occurs in A, while $A^{\vec{y}}$ indicates all *primary* names in A are in $\{\vec{y}\}$.
- $i, j, \ldots$ exclusively range over auxiliary names (in a given judgement).
- In each rule, no primary names in a judgement in its premise should overlap with auxiliary names in a judgement in either its premise or its conclusion.

Among the proof rules, (Zero) should need no illustration. (Res) is reminiscent of Hoare's rule for local name [24]. In (Weak), x can occur as an auxiliary name in A in the antecedent.

(Par) rule assumes non-circular parallel composition, where positive names in P are compensated by negative names in Q, but not vice versa (the precise condition is given in Appendix A). As a result, we obtain the simple proof rule, close to the cut rule in the sequent calculus. Since a parallel composition in affine processes can always be decomposed into a non-circular one and a recursion, this does not lose generality. 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) uses the expression for recursion given in §2.1, and combines mathematical induction and recursive behaviour, ensuring total correctness. In the rule, $A(e)$ denotes the result of substituting e for a hole in a formula, avoiding name capture. The rule is close to Harel's rule for the while command [19], which is known to be complete for strong models [4]. For tractable reasoning, one may extend the rule to well-founded induction [16].

(Bra$^{\downarrow}$) says that, if $x[\&_i(\vec{y}_i).P_i]$ 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 its dual, saying, for an output $\overline{x}\mathtt{in}_i(\vec{y})P$ to guarantee B relying on A, P itself should guarantee B relying on A, with x in B replaced by its carried value.

(In$^{!}$) and (Out$^{?}$) are another pair of mutually dual proof rules. For (In$^{!}$), suppose P guarantees C_2 at z, relying on A and C_1, with only the latter being about $\vec{y}$. Then $!x(\vec{y}z).P$ can guarantee, for all $\vec{y}$ satisfying C_1, that the result of invoking x with $\vec{y}$ satisfies C_2 which is now about x and $\vec{y}$ by substitution. B is a weaker property, so we safely place the formula in the conclusion.

In (Out$^{?}$), we first assume A as the global property of the environment, which in particular talks about x. Under A, suppose P can guarantee C_1 at $\vec{y}$. Further, suppose P can guarantee B *if* the environment satisfies C_2 at z. Finally suppose A and C_1 together ensure that invoking x with arguments $\vec{y}$ will result in z satisfying C_2. Then invoking x with arguments $\vec{y}$ and a return channel z will in effect guarantee B. The condition $x \bullet \vec{y} \Downarrow$ demands the result of invocation does not diverge, an essential requirement for the total correctness. Formally $a \Downarrow$ for a of type $\rho^{\uparrow}$ stands for: $\exists i \vec{x}.\ a = \mathtt{in}_i(\vec{x})$.

Moving to Figure 2, (Consequence) is similar to the corresponding rule in Hoare Logic, except that the entailment $\supset$ in the present logic is not only about number-theoretic facts but also about the universe of sequential behaviour. For the latter, we may use:

(extensionality) $\quad \forall \vec{x}.\, u \bullet \vec{x} = v \bullet \vec{x} \quad \supset \quad u = v.$

(data) $\quad \mathtt{in}_e(\vec{x}) = \mathtt{in}_{e'}(\vec{y}) \quad \supset \quad e = e' \wedge \vec{x} = \vec{y}.$

The remaining two structural rules 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). These two rules are essentially simple instances of Hoare's well-known adaptation rule [22], but are found to be useful in the present proof system (for other structural rules, see [27, §3.6]).

Many inference examples using these proof rules are given in [27]: some are also found in Section 3 in relationship with proof rules for higher-order functions. The following result is proved in [26, Sections 3 and 4] by rule induction of the proof rules.

Theorem 1 *(soundness) For each* $\vdash P \triangleright \Gamma$*, if* $P^{\Gamma} \vdash \textbf{rely}\ A\ \textbf{guar}\ B$ *then* $P^{\Gamma} \models \textbf{rely}\ A\ \textbf{guar}\ B$.

3. Program Logics via Process Logics

3.1 PCFv and its Process Encoding

In this section we show how we can derive a program logic for the call-by-value PCF (henceforth PCFv) from the affine process logic, leading to natural proof rules as well as their soundness. First we summarise the syntax of PCFv.

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

Binding etc. are standard. $\alpha,\beta,\ldots$ are types, whose grammar is to be given soon. $\mathbf{c}$ is a constant for which we consider booleans $\texttt{t},\texttt{f}$ and natural numbers n. $\mathbf{op}(M_0,...,M_{n-1})$ denotes an n-ary arithmetic or boolean operation (e.g. $\mathsf{succ}(M)$, $M\wedge N$). *Values* $(V,V',\ldots)$ are variables, constants, λ-abstractions and recursions. We use the standard weak call-by-value one-step reduction [15], written $M\longrightarrow M'$.

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

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

$\mathbb{B}$ and $\mathbb{N}$ are *atomic types*. A typed term has the form $\Gamma\vdash M:\alpha$, where Γ is a *base*, which is a finite map from variables to types. The typing rules are standard [15] and are omitted. Assuming $\Gamma\vdash M_{1,2}:\alpha$, we write $\Gamma\vdash M_1\cong_\lambda M_2:\alpha$ when, for each closing context $C[M_i]$ of type $\mathbb{N}$, we have $C[M_1]\Downarrow n$ iff $C[M_2]\Downarrow n$, where $\Downarrow$ denotes convergence to a value.

PCFv can be fully abstractly embedded into affine processes [6] using the encoding by Milner [44, 30]. The encoding of types is given by:[2]

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

The encoding of programs closely follows that of types. There are two kinds: $[\![V]\!]_m$ (for values) and $\langle\!\langle M\rangle\!\rangle_u$ (for all terms). Some of the main ones are:

$$\begin{array}{rcl}
[\![n]\!]_m & \stackrel{\text{def}}{=} & !m(c).\overline{c}\,\mathtt{in}_n\\
[\![x^\alpha]\!]_m & \stackrel{\text{def}}{=} & [m\to x]^{\alpha^\circ}\\
[\![\lambda x^\alpha.M]\!]_m & \stackrel{\text{def}}{=} & !m(xc).\langle\!\langle M\rangle\!\rangle_c\\
[\![\mu x^{\alpha\Rightarrow\beta}.V]\!]_m & \stackrel{\text{def}}{=} & (\nu x)([\![V]\!]_m|[x\to m]^{(\alpha\Rightarrow\beta)^\circ})\\
\langle\!\langle MN\rangle\!\rangle_u & \stackrel{\text{def}}{=} & (\nu m)(\langle\!\langle M\rangle\!\rangle_m|m(c).(\nu n)(\langle\!\langle N\rangle\!\rangle_n|n(e).\overline{c}\langle eu\rangle))\\
\langle\!\langle V\rangle\!\rangle_u & \stackrel{\text{def}}{=} & \overline{u}(m)[\![V]\!]_m.
\end{array}$$

The encoding is type-preserving in the following way:

$$\Gamma\vdash M:\alpha\,,\quad u \quad\mapsto\quad \vdash\langle\!\langle M\rangle\!\rangle_u \triangleright \overline{\Gamma^\circ},\, u:(\alpha^\circ)^{\uparrow} \qquad (7)$$

where $\overline{\Gamma^\circ}$ is the result of pointwise mapping and dualising each channel type.

Proposition 2 *(Berger, Honda, Yoshida [6])*

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

[2] To be precise, we use a truly linear mode [59] for $\uparrow$ in the encoding of each atomic type (we can also use a terser, but less uniform, encoding in [6]). See [27, §6.2] for detail.

3.2 Assertions for PCFv

The significance of type preservation in the schema (7) is that the translation guides us to develop a simple program logic for PCFv. Let us start from an assertion for the encoding of $\Gamma\vdash M:\alpha$ in the typed process logic. By type preservation in (7), we know it has the following form:

$$\langle\!\langle M\rangle\!\rangle_u\vdash \mathbf{rely}\,A\,\mathbf{guar}\,E \qquad (8)$$

where A is about $\overline{\Gamma^\circ}$ and E is about $u:(\alpha^\circ)^{\uparrow}$. A key observation which leads to the assertion language for PCFv is that, since the process logic under consideration is total, as far as A is satisfiable (and if not, i.e. if $A\equiv\mathsf{F}$, the assertion is vacuously valid for any E), we can set u of E as total, so that we can write:

$$E \quad\equiv\quad \exists m.(u=(m)^{\uparrow}\wedge B) \qquad (9)$$

We further observe each primary name, say x_i, in A is given a $(\cdot)^\circ$-mapped type, which is in accordance with the type α° of m. This allows us to uniformly "decode" this assertion back to M:

$$A'\vdash M:_u B' \quad\leftrightarrow\quad \langle\!\langle M\rangle\!\rangle_u\vdash\mathbf{rely}\,A\,\mathbf{guar}\,E \qquad (10)$$

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*. 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, assuming the property A' of 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 can now formally introduce syntax of assertions. Firstly, the grammar of terms and formulae of the logic follows. Below $\star\in\{\wedge,\vee,\supset\}$ and $Q\in\{\forall,\exists\}$.

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

$\bot^\alpha$ denotes divergence (which is seldom used in specifications but is needed for semantic completeness), often written $\bot$. The operator $\bullet$ can be understood as the call-by-value version of application in combinatory logic [12]. We write $e\Downarrow$ for $e\neq\bot$, which corresponds to $a\Downarrow$ in the process logic, cf. §2.3. We distinguish a term of a boolean type (say $\mathbf{t}$) from a formula (say T). This is necessary since a boolean term can take the value $\bot$. In practice, however, we can often infer a given boolean term is total, in which case it can be soundly identified with a formula, as is often done in Hoare Logic.

Terms are typed starting from constants and variables, requiring, in $e_1\bullet e_2$, that e_1 (resp. e_2) has type $\alpha\Rightarrow\beta$ (resp. α) for some α and β, in which case the whole term has type β. The well-typedness of formulae is defined as before.

We can now present the syntax of judgements.

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

In both, we assume $\Gamma\vdash M:\alpha$, $u\notin\mathsf{fn}(A)\cup\mathsf{fn}(\Gamma)$ and $\mathsf{fn}(\Gamma)\cap\mathsf{fn}(B)=\emptyset$, as well as well-typedness of formulae. Following Section 2, we call A (resp. B) *rely formula* (resp. *guarantee formula*) of the judgement. In the judgement, the primary names in A are those occurring in Γ, while the primary name of B is the anchor u. We often omit type annotation, writing e.g. $A\vdash M:_u B$. By an *assertion* we mean either a judgement or the rely/guarantee formula used in it.

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

$$\mathsf{T}\vdash\lambda x^\alpha.x\ :_u\ \forall y^\alpha.\,u\bullet y=y \qquad (11)$$

which says that the program, when applied to any well-typed argument y of type α, will return that argument as a result. This sequent has a precise analogue in the process logic via encoding. Since the program under consideration is a value, we can take $[\![\lambda x^{\alpha}.x]\!]_u$, for which we can state:

$$!u(xm).\overline{m}(c)[c \rightarrow x]^{\alpha^{\circ}} \vdash \textbf{rely}\, \top\, \textbf{guar}\, \forall y.\ u \bullet y = (y)^{\uparrow} \qquad (12)$$

which is easily derivable in the proof rules in §2.3. Note $(y)^{\uparrow}$ in (12) becomes simply y in (11).

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

$$\forall x^{\mathbb{N}}.Even(f \bullet x) \vdash (f3)+1 :_u Odd(u) \qquad (13)$$

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

$$\begin{array}{l} \overline{f}(xc)([\![3]\!]_x \,|\, c(y).\overline{y}(e)e[\&_n.\overline{u}(c)[\![n+1]\!]_c]) \ \vdash \\ \qquad \textbf{rely} \quad \forall x.\ \exists i.\ (\underline{f \bullet x \text{ gives } i} \wedge Even(i)) \\ \qquad \textbf{guar} \quad \exists j.\ (\underline{u \text{ gives } j} \wedge Odd(j)) \end{array}$$

The indirection via affine output complicates the formulae: yet we can precisely read out the high-level assertion in (13) from the low-level one. More examples of assertions will be given in §3.5.

3.3 Semantics of Judgement

Intuitively, a judgement $A \models M :_u B$ under the typing $\vec{x} : \vec{\tau} \vdash M : \beta$ means that, restricting our attention to total correctness as before:

> *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.[3]

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

Let $\kappa_1^{\alpha \Rightarrow \beta}$ and κ_2^{α}. Then $\kappa_1 \bullet \kappa_2$ is given as the congruence class including MN for $M \in \kappa_1$ and $N \in \kappa_2$. Arithmetic operations etc. are similarly defined. A *model* ξ maps names to total behaviours. An *interpretation* I maps names to both total and non-total behaviours. Given a model ξ and an interpretation I, the map $[\![e]\!]I \cdot \xi$ is defined by induction on e, starting from $[\![x]\!]I \cdot \xi = (I \cup \xi)(x)$. The satisfaction relation $\xi \models^I A$ is standard. We then write $\kappa \models^I_u A$ (read: κ *with anchor u satisfies A under I*) when $u : \kappa \models^I A$ (where $u : \kappa$ is the model that maps u to κ). Finally with M closed, $M \models^I_u A$ stands for $\exists \kappa.\ (M \in \kappa \wedge \kappa \models^I_u A)$. We also write $\vec{x} : \vec{V} \models^I A$ for $\vec{x} : \vec{\kappa} \models^I A$ with $V_i \in \kappa_i$. We can now define:

Definition 2 *(semantics of judgement) Assume $\vec{x} : \vec{\beta} \vdash M : \alpha$. Then $A \models M :_u B$ iff, for each well-typed I and each $\vec{V}$ of types $\vec{\beta}$ such that $\vec{x}:\vec{V} \models^I A$, we have $M[\vec{V}/\vec{x}] \models^I_u B$.*

The validity in the PCFv-logic is precisely embeddable into that of the process logic. The translation of formulae is written $\langle\!\langle A \rangle\!\rangle^{\vec{v}}$, where $\vec{v}$ indicate the primary names in A (which are to be treated as total behaviours). The embedding starts from (always choosing fresh names for newly introduced names):

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

The rest is compositional with respect to logical connectives and quantifiers. The map $\langle\!\langle e \rangle\!\rangle^{\vec{y}}_x$ encodes a term e to a formula[4], naming the term as x and indicating $\vec{y}$ are primary. The map is defined inductively as follows.

$$\begin{array}{rcl}
\langle\!\langle x \rangle\!\rangle^{\vec{v}}_u & \stackrel{\text{def}}{=} & \begin{cases} u = x & (x \notin \{\vec{v}\}) \\ u = (x)^{\uparrow} & (x \in \{\vec{v}\}) \end{cases} \\
\langle\!\langle \mathbf{op}(\vec{e}) \rangle\!\rangle^{\vec{v}}_u & \stackrel{\text{def}}{=} & \exists \vec{x}.\, (\wedge_i \langle\!\langle e_i \rangle\!\rangle^{\vec{v}}_{x_i} \ \wedge\ (u \Downarrow \supset \wedge_i x_i \Downarrow) \ \wedge \\
 & & \quad \forall \vec{y}.(\ \wedge_i x_i = \mathtt{in}_{y_i}() \supset u = \mathtt{in}_{\mathbf{op}(\vec{y})}()\ )\ ) \\
\langle\!\langle e_1 \bullet e_2 \rangle\!\rangle^{\vec{v}}_u & \stackrel{\text{def}}{=} & \exists x_1 x_2.\, (\wedge_i \langle\!\langle e_i \rangle\!\rangle^{\vec{v}}_{x_i} \ \wedge\ (u \Downarrow \supset \wedge_i x_i \Downarrow) \ \wedge \\
 & & \quad \forall y_1 y_2.(\ \wedge_i \underline{x_i \text{ gives } y_i} \supset \underline{u \text{ gives } y_1 \bullet y_2}\ )) \\
\langle\!\langle \perp \rangle\!\rangle^{\vec{v}}_u & \stackrel{\text{def}}{=} & \neg u \Downarrow
\end{array}$$

In the second line, an operation **op** is assumed to be total and strict in each argument (when **op** is nullary the map becomes that for a constant). The translation decomposes coalesced PCFv-types into their affine (partial) and replicated (total) parts. This decomposition makes the mapping complex. The resulting complexity in turn suggests abstraction and convenience in specification we obtain when we move from the fine-grained process logic to a logic tailored for a specific programming language.

A key result on the logical embedding follows. The property is called *logical full abstraction* by Longley and Plotkin [41], though in a different setting. The proof uses Proposition 2 (in particular definability), see [27, §6.4] for detail. Below we use the entailment $\exists m.(u = (m)^{\uparrow} \wedge \langle\!\langle B \rangle\!\rangle^{u}[m/u]) \supset \langle\!\langle B \rangle\!\rangle^{\varepsilon}$, which allows us to simplify the guarantee formula.

Theorem 2 *(logical full abstraction) Assume $\vec{x}:\vec{\alpha} \vdash M : \beta$. Then $A \models M :_m B$ iff $\langle\!\langle M \rangle\!\rangle_u \models$ **rely** $\langle\!\langle A \rangle\!\rangle^{\vec{x}}$ **guar** $\langle\!\langle B \rangle\!\rangle^{\varepsilon}$.*

3.4 Proof Rules for PCFv-Logic

The correspondence between assertions in the PCFv-logic and those in the process logic leads to simple proof rules for PCFv, which are given in Figure 3. As in §2.3, $A^{\vec{y}}$ means primary names in A are inside $\{\vec{y}\}$, $A^{-\vec{y}}$ means no name from $\vec{y}$ occurs in A, and $i, i', \ldots$ exclusively range over auxiliary names. Again as in §2.3, we assume that, in each rule, no primary names in a judgement in the premise should overlap with auxiliary names in any judgement in the rule, either in the premise or in the conclusion.

All rules in Figure 3 should naturally read 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. This axiom corresponds to, via encoding, the following *copycat laws*, derivable in the rules of the process logic (see [27, §3] for the derivation). Below we assume τ has mode ! and ρ has mode $\downarrow$.

(Copycat-!)	$[x \rightarrow x']^{\tau} \vdash \textbf{rely}\, A[x'/x]\, \textbf{guar}\, A$
(Copycat-$\downarrow$)	$[x' \rightarrow x]^{\rho} \vdash \textbf{rely}\, A[x'/x]\, \textbf{guar}\, A$

$[Const]$ is about identity of u and a constant **c**, which is reminiscent of the assignment rule in Hoare Logic. $[Op]$ is a generic rule for

[3] An alternative model construction is discussed in [27, Appendix B].

[4] Terms in the PCFv-logic include composite values, which in general cannot be expressed as terms in the process logic.

$$[Var]\ \frac{-}{A[x/u] \vdash x :_u A} \qquad [Const]\ \frac{-}{A[\mathbf{c}/u] \vdash \mathbf{c} :_u A}$$

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

$$[If]\ \frac{A \vdash M :_b C \quad A \wedge C[\mathbf{t}/b] \vdash N_1 :_u B \quad A \wedge C[\mathbf{f}/b] \vdash N_2 :_u B}{A \vdash \mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2 :_u B}$$

$$[Abs]\ \frac{A^{\text{-}x\vec{i}} \wedge C_1^x \vdash M :_m C_2 \quad \forall x\vec{i}.(C_1 \supset C_2[u \bullet x/m]) \supset B}{A \vdash \lambda x.M :_u B}$$

$$[App]\ \frac{A \vdash M :_m C_1 \quad A \vdash N :_x C_2 \quad C_1 \supset C_2 \supset B[m \bullet x/u] \wedge m \bullet x \Downarrow}{A \vdash MN :_u B}$$

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

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

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

Figure 3: Proof Rules for PCFv

operators (which in fact subsumes the rule for constant). Syntactically $[Op]$ assumes the corresponding operator is in the language of the logic; semantically it assumes the operator is total.

$[If]$ is close to the conditional rule in Hoare Logic, keeping clean symmetry like its predecessor. Note, in the rule, $A \vdash M :_b C$ indicates (as far as A is satisfiable) M terminates. This allows us to conclude the whole term terminates.

$[Abs]$ says that, if, under the assumption C_1 about x, the program M named as m satisfies C_2, then $\lambda x.M$ named as u satisfies the following property: for any x satisfying C_1, the result of applying x to u, named as m, satisfies C_2. The global assumption A should not mention x. The quantification of the auxiliary names $\vec{i}$ is in fact redundant in the presence of [Aux-$\forall$] discussed below, but having it together is almost always convenient in reasoning. Dually $[App]$ says that if M named as m satisfies C_1, and N named as x satisfies C_2, and C_1 and C_2 together imply B whose u is substituted for $m \bullet x$, as well as convergence of the invocation, then MN named as u satisfies B.

$[Rec]$ combines recursion with mathematical induction, in direct correspondence with (Rec). Again we may extend this rule to well-founded induction.

The final three rules are structural rules. $[Consequence]$ is just as in Hoare logic. $[Aux\text{-}\forall]$ and $[Aux\text{-}\exists]$ come from the semantics of auxiliary names, just as in §2.3. For applying the consequence rule, we need to calculate the validity of formulae in the domain of call-by-value sequential behaviour, in addition that of the standard number theory. Figure 4 lists three basic rules (rules similar to ($\bot$-right) are valid for the first-order operators).

Each of these proof rules can be cleanly decomposed into a few steps of inference in the process logic. As an example, we show the case of $[App]$. For brevity let M and N be values, in which case the

(ext)	$\forall y.\, e_1 \bullet y = e_2 \bullet y$	$\supset \quad e_1 = e_2$
($\bot$-left)	$\bot \bullet e = \bot$	
($\bot$-right)	$e \bullet \bot = \bot$	

Figure 4: Axioms for Strict $\bullet$

encoding becomes:

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

The following derivation starts from the antecedents of $[App]$ and finally reaches its conclusion, all in the encoded form. Ant 1,2,3 are three conditions in the antecedent, while IH stands for the induction hypothesis.

1. $\llbracket M \rrbracket_m \vdash \textbf{rely}\, \langle\!\langle A \rangle\!\rangle^{\vec{x}}\, \textbf{guar}\, \langle\!\langle C_1 \rangle\!\rangle^m$	(Ant 1, IH)
2. $\llbracket N \rrbracket_x \vdash \textbf{rely}\, \langle\!\langle A \rangle\!\rangle^{\vec{x}}\, \textbf{guar}\, \langle\!\langle C_2 \rangle\!\rangle^x$	(Ant 2, IH)
3. $[u' \to u] \vdash \textbf{rely}\, \langle\!\langle B \rangle\!\rangle^{\varepsilon}[u'/u]\, \textbf{guar}\, \langle\!\langle B \rangle\!\rangle^{\varepsilon}$	(copycat law)
4. $\overline{m}(xu')(\llbracket N \rrbracket_x | [u' \to u])) \vdash$ $\textbf{rely}\, \langle\!\langle C_1 \rangle\!\rangle^m \wedge \langle\!\langle A \rangle\!\rangle^{\vec{x}}\, \textbf{guar}\, \langle\!\langle B \rangle\!\rangle^{\varepsilon}$	(2, 3, Ant 3, (Out))
5. $\llbracket M \rrbracket_m | \overline{m}(xu')(\llbracket N \rrbracket_x | [u' \to u]) \vdash$ $\textbf{rely}\, \langle\!\langle A \rangle\!\rangle^{\vec{x}}\, \textbf{guar}\, \langle\!\langle B \rangle\!\rangle^{\varepsilon}$	(1, 4, (Par))
6. $\langle\!\langle MN \rangle\!\rangle_u \vdash \textbf{rely}\, \langle\!\langle A \rangle\!\rangle^{\vec{x}}\, \textbf{guar}\, \langle\!\langle B \rangle\!\rangle^{\varepsilon}$	(Res)

Decomposing each rule in the same way, we arrive at:

Proposition 3 *Let* $\Gamma \vdash M : \alpha$ *with* $\Gamma = \vec{x} : \vec{\alpha}$. *Then* $A \vdash M :_u B$ *implies* $\langle\!\langle M \rangle\!\rangle_u \vdash \textbf{rely}\, \langle\!\langle A \rangle\!\rangle^{\vec{x}}\, \textbf{guar}\, \langle\!\langle B \rangle\!\rangle^{\varepsilon}$.

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

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

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

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

Note the proof needs both directions of Theorem 2. Examples of inferences using these proof rules are listed in §3.5.

3.5 Examples of Inferences in PCFv-Logic

We show two simple examples of inferences in the PCFv-logic. Further examples are found in [27, §6]. First we derive a natural specification for the identity function.

1.	$\mathsf{T} \wedge x = i \vdash x :_m m = i$	(Var)
2.	$\mathsf{T} \vdash \lambda x.x :_u \forall x^{\mathbb{N}} i^{\mathbb{N}}.(x = i \supset u \bullet x = i)$	(Abs)
3.	$\mathsf{T} \vdash \lambda x.x :_u \forall x^{\mathbb{N}}.(u \bullet x = x)$	(Conseq)

Next we consider the standard recursive factorial:

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

The assertion which we wish to prove is:

$$\mathsf{T} \vdash \mathit{Fact} :_u \forall n^{\mathbb{N}}.\ u \bullet n = n!. \tag{14}$$

Being simple as a statement, the inferences to reach this assertion will use all proof rules of PCFv-logic given in Figure 3. By the shape of $[Rec]$, the proof is divided into two parts, the base case and the induction step. For brevity we let $B(f)(n) \stackrel{\text{def}}{=} f \bullet n = n!$ and $M \stackrel{\text{def}}{=} \mathtt{if}\ x=0\ \mathtt{then}\ 1\ \mathtt{else}\ x \times f(x-1)$. We first show the base case. Below (falsity) stands for the lemma $\mathsf{F} \vdash N :_u A$ for each N and A, which is easily derived by rule induction.

1.	$x=0 \vdash x=0 :_b b=\mathbf{t}$	(Var, Num, Eq)
2.	$x=0 \wedge \mathbf{t}=\mathbf{t} \vdash 1 :_v v=1$	(Num, Conseq)
3.	$x=0 \wedge \mathbf{f}=\mathbf{t} \vdash x \times f(x-1) :_v v=1$	(falsity)
4.	$x=0 \vdash \mathtt{if}\ x=0\ \mathtt{then}\ 1\ \mathtt{else}\ x \times f(x-1) :_v v=1$	(1, 2, 3, If)
5.	$\mathsf{T} \vdash \lambda x.M :_u \forall x^{\mathbb{N}}.(x=0 \supset u \bullet x = 1)$	(4, Abs)
6.	$\mathsf{T} \vdash \lambda x.M :_u B(u)(0)$	(Conseq)

where (Eq) is an instance of $[Op]$. For the induction step:

1.	$B(f)(n) \wedge x=n+1 \vdash x=0 :_b b=\mathbf{f}$	(Var, Num, Eq)
2.	$B(f)(n) \wedge x=n+1 \wedge \mathsf{F} \vdash 1 :_v v=(n+1)!$	(Num, Conseq)
3.	$B(f)(n) \wedge x=n+1 \wedge \mathsf{T} \vdash x :_m m=n+1$	(Var, Conseq)
4.	$B(f)(n) \wedge x=n+1 \wedge \mathsf{T} \vdash f :_l B(l)(n)$	(Var, Conseq)
5.	$B(f)(n) \wedge x=n+1 \wedge \mathsf{T} \vdash x-1 :_k k=n$	(3. Subt)
6.	$B(f)(n) \wedge x=n+1 \vdash f(x-1) :_v v=(n-1)!$	(4, 5, App)
7.	$B(f)(n) \wedge x=n+1 \vdash x \times f(x-1) :_v v=n!$	(3, 6, Mult)
8.	$B(f)(n) \wedge x=n+1 \vdash M :_v v=n!$	(1, 2, 6, If)
9.	$\mathsf{T} \wedge B(f)(n) \vdash \lambda x.M :_u B(u)(n+1)$	(7, Abs, Conseq)

where (Mult) and (Subt) are instances of $[Op]$. We can now combine the two conclusions, writing them as $(*)$ and $(**)$:

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

thus reaching the required statement (14).

3.6 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 β_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 [27]. For example, to treat sums and products, one only has to add the following terms to the PCFv-logic.

$$e ::= \ \ldots \ \mid \ () \ \mid \ \langle e_1, e_2 \rangle \ \mid \ \pi_i(e) \ \mid \ \mathtt{in}_i(e) \qquad (i=1,2)$$

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 [52]: we extend types with recursion $\mu \mathrm{X}.\alpha$ (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 $\mathtt{List}(\alpha) \stackrel{\text{def}}{=} \mu \mathrm{Y}.(\mathtt{Unit} + (\alpha \times \mathrm{Y}))$. 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.

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

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 $\forall l^{\mathtt{List}(\mathbb{N})}.(A(u.l) \wedge B(u,l))$ (see [27, §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, and data structure with destructive update (cf. [50, 56]). 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 λ-calculus, $\Lambda X.\lambda x^X.x : \forall X.X \Rightarrow X$, with anchor u, as follows.

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

The assertion says that a type-abstracted behaviour named as u, when it is applied to any type α and well-typed argument y of type α, 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 [27, 28] for the account of these and other experiments.

References

[1] Abadi, M., Cardelli, L., Curien, P.-L., Formal Parametric Polymorphism, *TCS* 121, 1-2, (Dec), 9-58. Elsevier, 1993.

[2] Abadi, M, Leino, R. A logic for object-oriented programs, pages 682–696, *TAPSOFT'97*, LNCS, 1997.

[3] Abramsky, S., Jagadeesan, R. and Malacaria, P., Full Abstraction for PCF. *Info. & Comp.* 163 (2000), 409-470.

[4] Apt, K.R. Ten Years of Hoare Logic: a survey. *TOPLAS.* 3:431-483, 1981.

[5] Baeten, J.C.M. and Weijland, W.P. *Process Algebra.* Cambridge University Press, 1990.

[6] Berger, M., Honda, K. and Yoshida, N., Sequentiality and the π-Calculus, *TLCA01*, LNCS 2044, 29–45, 2001.

[7] Berger, M., Honda, K. and Yoshida, N., Genericity and the π-Calculus, *FoSSaCs'03*, LNCS 2620, 103–119, 2003.

[8] Boudol, G., Asynchrony and the pi-calculus, INRIA Research Report 1702, 1992.

[9] Caires, L. and Cardelli, L., A Spatial Logic for Concurrency (Part I). *Info. & Comp.*, 2003.

[10] Clint, M. and Hoare C.A.R. Program proving: jumps and functions. Acta Inf. 1:214–224, 1971.

[11] Cousot, P. Methods and logics for proving programs. *Handbook of Theoretical Computer Science*, volume B, 843-993. Elsevier, 1999.

[12] Curry, B. and Feys, R. *Combinatory Logic*. North-Holland, 1958.

[13] Dam, M. Proof Systems for π-Calculus Logics. *Logic for Concurrency and Synchronization*. Studies in Logic and Computing, Oxford University Press, 2003.

[14] Gordon, M., Milner, A. and Wadsworth, C., Edinburgh LCF, LNCS 78, 1979.

[15] Gunter, C., *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992.

[16] Floyd, W. Assigning meaning to programs. *Prc. Symp. in Applied Mathematics*. 19:19-32, 1967.

[17] Francez N. and Pnueli, A. A proof method for cyclic programs. *Acta Inf.* 9, pp.133-157, 1978.

[18] Girard, J.-Y., Linear Logic, *TCS*, 50:1–102, 1987.

[19] Harel, D. Proving the correctness of regular deterministic programs. *TCS*, 12:61–81, 1980.

[20] Hennessy, M. and Milner, R. Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM*, 32:1, pp.137-161, 1985.

[21] Hoare, C.A.R. An axiomatic basis of computer programming. *CACM*, 12:576-580, 1969.

[22] Hoare, C.A.R. Procedures and Parameters: an axiomatic approach. Lecture Notes in Mathematics 188, 102–116, Semantics of Algorithmic Languages, Springer, 1971.

[23] Hoare, C.A.R. *Communicating Sequential Processes*, Prentice Hall, 1985.

[24] Hoare, C.A.R. and Wirth, N., An axiomatic definition of the programming language PASCAL. Acta Inf. 2:335–355, 1973.

[25] Honda, K., Composing Processes, *POPL'96*, 344-357, ACM, 1996.

[26] Honda, K. *Sequential Process Logics: Soundness Proofs*. Typescript, 50pp. November 2003/January 2004. Available at: www.dcs.qmul.ac.uk/˜kohei/logics.

[27] Honda, K. *Process Logic and Duality: Part (1) Sequential Processes*. Typescript, 210pp. March 2004. Available at: www.dcs.qmul.ac.uk/˜kohei/logics.

[28] Honda, K. and Yoshida, N., A Compositional Logic for Polymorphic Higher-Order Functions. *PPDP'04*, ACM, 2004.

[29] Honda, K. and Tokoro, M. An object calculus for asynchronous communication. ECOOP'91, LNCS 512, 133–147, 1991.

[30] Honda, K. and Yoshida, N. Game-theoretic analysis of call-by-value computation. *TCS*, 221:393–456, 1999.

[31] Honda, K. and Yoshida, N., A Uniform Type Structure for Secure Information Flow, *POPL'02*, 81–92, ACM, 2002.

[32] Honda, K. and Yoshida, N., Noninterference proofs from Flow Analysis. To appear in: *Journal of Functional Programming*, CUP, 2004.

[33] Honda, K., Yoshida, N. and Berger, M., Control in the π-Calculus, *Proc. CW'04*, ACM, 2004.

[34] Hyland, M. and Ong, L., "On Full Abstraction for PCF": I, II and III. *Info. & Comp.* 163 (2000), 285-408.

[35] Jacobs, B. et al. Reasoning about Java Classes (Preliminary Report). *OOPSLA'98*, ACM, 1998.

[36] Jones, C.B. Specification and Design of (Parallel) Programs. *Proc. IFIP 9th World Computer Congress*. North Holland, 321-332, 1983.

[37] Kobayashi, N., Pierce, B., and Turner, D., Linear types and the π-calculus, *TOPLAS*, 21(5):914–947, 1999.

[38] Kowaltowsky, T. Axiomatic approach to side effects and general jumps. *Acta Informatica 7, 357-360.*

[39] Laurent, O., Polarized games, LICS 2002, 265-274, IEEE, 2002.

[40] Leavens, G. and Baker, A. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. *FM'99: World Congress on Formal Methods*, Springer, 1999.

[41] Longley, J. and Plotkin, G. Logical Full Abstraction and PCF. *Tbilisi Symposium on Logic, Language and Information*. 333-352, Stanford, CSLI, 1998.

[42] Mendelson, E. *Introduction to Mathematical Logic (third edition)*. Wadsworth Inc., 1987.

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

[44] Milner, R., Functions as Processes, *MSCS.* 2(2):119–141, 1992,

[45] Milner, R., Speech by Robin Milner on receiving an Honorary Degree from the University of Bologna, ICALP'97, http://www.cs.unibo.it/icalp/Lauree_milner.html.

[46] Milner, R., Parrow, J. and Walker, D., A Calculus of Mobile Processes, *Info. & Comp.* 100(1):1–77, 1992.

[47] Milner, R., Parrow, J.G. and Walker, D.J., Modal logics for mobile processes, *TCS*, 114:149–171, 1993.

[48] Milner, R., Polyadic π-Calculus: a tutorial. *Proceedings of the International Summer School on Logic Algebra of Specification*, Marktoberdorf, 1992.

[49] Naur, P. Proof of algorithms by general snapshots. *BIT 6*, 310–316, 1966.

[50] O'Hearn, P., Yang, H. and Reynolds, J., Separation and Information Hiding, *POPL'04*, 268–280, 2004.

[51] Oheimb, D.V., Hoare Logic for Java in Isabelle/HOL. *Concurrency: Practice and Experience*. John Wiley, 2002.

[52] Pierce, B.C., *Types and Programming Languages*, MIT Press, 2002.

[53] Pierce, B.C. and Sangiorgi. D, Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.

[54] Plotkin, G. and Abadi, M., A Logic for Parametric Polymorphism, *LICS'98*, 42–53, IEEE Press, 1998.

[55] Poetzsh-Heffter, A. and Muller, P. A programming logic for sequential Java. ESOP'99, LNCS 1576, 162-176, 1999,

[56] Reynolds, J. Intuitionistic Reasoning about Shared Mutable Data Structure, Millennial Perspectives in Computer Science, Oxford, 1999.

[57] Sangiorgi, D., π-calculus, internal mobility, and agent-passing calculi. *TCS*, 167(2):235–271, 1996.

[58] Walker, D., Objects in the π-calculus. *Info. Comp.*, 116(2):253-271, 1995.

[59] Yoshida, N., Berger, M. and Honda, K., Strong Normalisation in the π-Calculus, *LICS'01*, 311–322, IEEE, 2001. Full version in: *Info. & Comp.*, 191:145–202, 2004.

[60] Yoshida, N., Honda, K. and Berger, M. Linearity and Bisimulation, FoSSaCs 2002, LNCS 2303, 417–433, 2002.

(Zero) $\dfrac{-}{\vdash \mathbf{0} \triangleright \_}$ (Par) $(\Gamma_1 \asymp \Gamma_2)$ $\dfrac{\vdash P_i \triangleright \Gamma_i \;\; (i=1,2)}{\vdash P_1 | P_2 \triangleright \Gamma_1 \odot \Gamma_2}$ (Res) $\dfrac{\vdash P \triangleright \Gamma, x : \tau^{!,\updownarrow}}{\vdash (\nu x) P \triangleright \Gamma}$ (Weak) $\dfrac{\vdash P \triangleright \Gamma^{-x}}{\vdash P \triangleright \Gamma, x : \tau^{?,\updownarrow}}$

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

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

Figure 5: Sequential Affine Typing

Appendix

A. Sequential Affine Typing

In the following, we present a concise, but self-contained, illustration of the affine type discipline used in the present paper. For further detail, see [27, §2.2].

Firstly, it suffices to consider the channel types of the shape given below [34, 3, 6] ($\vec{\tau}^p$ indicates that each type in $\vec{\tau}$ has mode p).

- $(\vec{\tau}^{?}\rho^{\uparrow})^{!}$ and, dually, $(\vec{\tau}^{!}\rho^{\downarrow})^{?}$.
- $[\&_{i \in I} \vec{\tau}_i^{?}]^{\downarrow}$ and, dually, $[\oplus_{i \in I} \vec{\tau}_i^{!}]^{\uparrow}$.

Note an input type only carries output types, dually for an output type. A replicated input type carries zero or more ?-types (which may be considered as arguments it may receive); and a unique affine output (which may be considered as a return channel). Dually for a ?-type.

The typing rules are presented in Figure 5, which are a variant of the sequential affine typing introduced in [6]. They differ from those in [6] in that it does not use IO-modes for ensuring sequentiality. This is made possible by the use of sequential action types. Some notations: "Γ, Δ" denotes the union of Γ and Δ, indicating at the same time disjointness of their domains. $?\Gamma$ indicates Γ only use types of mode ?, similarly for Γ^{-x} says x does not occur in Γ. For parallel composition, we use the partial operation $\odot$, given by:

(1) $\Gamma \odot \Delta = \Gamma, \Delta \quad (\mathsf{fn}(\Gamma) \cap \mathsf{fn}(\Delta) = \emptyset)$.

(2) $\Gamma \cdot x : \tau \odot \Delta \cdot x : \tau' = \Theta \cdot x : \rho \quad (\tau \odot \tau' = \rho \text{ and } \Gamma \odot \Delta = \Theta)$.

In other cases $\Gamma \odot \Delta$ is undefined. In (2), $\tau \odot \rho$ is a commutative partial operation which satisfies:

$$\text{(a-1) } \tau^{?} \odot \tau^{?} = \tau^{?}, \quad \text{(a-2) } \tau^{!} \odot \overline{\tau}^{?} = \tau^{!}, \quad \text{(b) } \tau^{\downarrow} \odot \overline{\tau}^{\uparrow} = \updownarrow.$$

(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 linear output compensate each other to become the uncomposable type.

We briefly illustrate the typing rules. In each rule, we assume all occurring types are well-formed and all occurring action types are sequential (including those in conclusion). (Zero) assigns the empty type to $\mathbf{0}$. (Par) composes two processes using $\odot$: in the antecedent, we write $\Gamma \asymp \Delta$ when $\Gamma \odot \Delta$ is defined. We can show, as far as $\Gamma \odot \Delta$ is sequential, $\odot$ ensures both determinacy and single-threadedness of a process.

(Par-nc) $\dfrac{\begin{array}{l}\vdash P_1 \triangleright ?{\downarrow}\Gamma_1,\ !{\uparrow}\Delta_1,\ !{\uparrow}\Theta \\ \vdash P_2 \triangleright \overline{\Theta},\ ?{\downarrow}\Gamma_2,\ !{\uparrow}\Delta_2 \\ \Gamma_1 \asymp \Gamma_2, \mathsf{fn}(\Delta_i) \cap \mathsf{fn}(\Gamma_j, \Delta_j) = \emptyset \ (i \neq j)\end{array}}{\vdash P_1 | P_2 \triangleright \Gamma_1 \odot \Gamma_2,\ \Delta_1, \Delta_2,\ \Theta \odot \overline{\Theta}}$ (Rec) $\dfrac{\vdash P \triangleright ?{\downarrow}\Gamma,\ \vec{y} : \overline{\vec{\tau}},\ \vec{x} : \vec{\tau}}{\vdash \mu\vec{y} = \vec{x}.P \triangleright \Delta,\ \vec{x} : \vec{\tau}^{!}}$

Figure 6: Rules for Non-Circular Composition and Recursion

(Res) hides a channel of mode ! and $\updownarrow$ (thus channels of mode ?, $\downarrow$ and $\uparrow$ need their duals before hiding). (Weak) weakens ?- as well as $\updownarrow$-channels. (Bra) says that if, for each i, a process has only output types as specified, then we can prefix it with a branching input (note each type in $\vec{\tau}_i$ has mode ? by well-formedness of types). (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 ρ, which is a $\uparrow$-type) to construct a replicated input, which suppresses free outputs Γ. Note Γ 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.

Processes typable with these rules cover essentially all sequential affine processes in [6] modulo $\equiv$. which allows fully abstract embeddings of, among others, both call-by-value and call-by-name PCF with products and sums. The typability is closed under reduction in the way: $\vdash P \triangleright \Gamma$ and $P \longrightarrow P'$ implies $P' \equiv P''$ such that $\vdash P'' \triangleright A$ for some P''.

Finally we list the non-circular version of (Par) rule and the recursion rule in Figure 6. $?{\downarrow}\Gamma$ etc. indicate the modes of types in Γ, just like $?\Gamma$. These rules are used as the basis of the proof rules in §2.3. The non-circular (Par) rule is equipotent to the original (Par) rule when combined with the recursion rule up to simple syntactic conversion.

B. Copycat

We define a copycat $[x \to y]^{\tau}$ for an arbitrary τ of mode $\downarrow$ or ! by the following induction.

$$[x \to y]^{\tau} \stackrel{\text{def}}{=} \begin{cases} !x(\vec{a}).\overline{y}(\vec{b})\Pi_i[b_i \to a_i]^{\overline{\rho_i}} & (\tau = (\vec{\rho})^{!}) \\ x[\&_i(\vec{a}_i).\overline{y}\mathtt{in}_i(\vec{b}_i)\Pi_{ij}[b_{ij} \to a_{ij}]^{\overline{\rho_{ij}}}] & (\tau = [\&_i \vec{\rho}_i]^{\downarrow}) \end{cases}$$

where, in $[\&_i \vec{\rho}_i]^{\downarrow}$, we set, for each i, $\vec{\rho}_i \stackrel{\text{def}}{=} \rho_{i1}..\rho_{ij}..\rho_{im_i}$ for some m_i.

C. Well-typed Behaviour Expressions

We say *a has type* τ by the following induction.

(1) x^{τ} has type τ;

(2) $a \bullet \vec{b}$ has type $\tau^{\uparrow}$ iff a has type $(\overline{\vec{\rho}}\tau)^{!}$ and $\vec{b}$ has type $\vec{\rho}$;

(3) $\mathtt{in}_e^{\rho}(\vec{a})$ has type $\rho = [\oplus_{i \in T} \vec{\tau}_i]^{\uparrow}$ iff e has type T and that, for each valuation, if $e = i \in T$ then $\vec{a}$ has type $\vec{\tau}_i$.

Above a *valuation* is a well-typed map from names to constants. Note each well-typed behaviour expression has a unique type. It usually (as in this paper) suffices, for various embeddings of programming languages, to consider only those terms whose well-typedness is feasibly checkable.