A π-calculus Semantics of Concurrent Idealised ALGOL

Christine Röckl[*] Davide Sangiorgi[†]

September 28, 1998

Abstract

We study the use of the π-calculus as a metalanguage for the semantics of languages such as *Concurrent Idealised ALGOL* (CIA), which combines imperative, functional and concurrent features. We first present an operational semantics for CIA, defined in terms of SOS rules and a contextual form of behavioural equivalence; then a π-calculus semantics. As behavioural equivalence on π-calculus processes we choose the standard (weak early) bisimilarity. We compare the two semantics, showing that there is a close operational correspondence between them and that the π-calculus semantics is sound. This allows us to apply the π-calculus theory in proving behavioural properties of CIA phrases. We discuss laws and examples from [MS88, Bro96, MT90a, MT90b], as well as a more complex example involving procedures of higher order. By means of counterexamples we show that the π-calculus semantics is not complete, and discuss how these counterexamples can be dealt with by adopting refined forms of π-calculus types and, correspondingly, adopting typed versions of bisimilarity.

1 Introduction

Reynolds formalised Idealised ALGOL (IA) as a *simple imperative language* enriched with a procedural mechanism provided by a *typed call-by-name λ-calculus* [Rey81]. IA combines in an elegant way imperative and functional features, and since its introduction has been the object of extensive study (cf. [OT97]). *Concurrent Idealised ALGOL* (CIA) was introduced by Brookes as an extension of IA with shared variable parallelism [Bro96]. CIA allows parallel composition of commands and includes an **await** operator for imposing atomicity. Brookes [Bro96] has presented an elegant denotational model for CIA, extending a Kripke-style Possible Worlds semantics. From a semantical point of view CIA is a challenging language, because it combines imperative, functional and concurrent features, and because of its atomicity construct.

In this paper we study semantics of CIA given by translation into the π-calculus. The main reasons for using π-calculus are the following. The π-calculus offers a well-developed theory that we wish to exploit, through the translation, to reason on CIA terms. We also prefer the π-calculus semantics to a direct operational semantics of CIA because π-calculus is, syntactically, a first-order language, whereas CIA is higher order: that is, values in CIA may contain arbitrary terms, while values in π-calculus contain just names (in typed versions, possibly also basic values such as integers

---

[*] Technische Universität München, Arcisstr. 21, D-80290 München, `Email: roeckl@in.tum.de`

[†] INRIA – Sophia Antipolis, 2004, Rue des Lucioles, B.P.93, F-06902 Sophia Antipolis Cedex `Email: davide.sangiorgi@inria.sophia.fr`

and booleans) but not, for instance, process terms. In higher-order languages, defining satisfactory notions of behavioural equivalences, and proof techniques for them, may be hard. For instance, definitions of equivalence usually require forms of universal quantification on terms, which makes proofs of process equivalences complex. Also it may be very hard to prove that a bisimilarity is a congruence relation. (Typically congruence of bisimilarity in higher-order languages is proved using Howe's technique [How96]; but attempts to extend the technique to languages with local state have been unsuccessful so far; see discussions in [FHJ95].) A further advantage of the π-calculus semantics is that, as states are represented by processes, no *snapback effects* (reversibility of state changes, cf. [AM96, AM97, OT97]) can occur; by contrast, models representing states by functions—as in general denotational models—suffer of snapback effects, which are usually removed by means of logical relations [OT97].

Our study has also π-calculus motivations: we wish to understand to what extent the π-calculus is appropriate for giving semantics to languages such as CIA. Previous work gives evidence that the π-calculus can model references, functions and various forms of (non atomic) parallelism [Wal95, Jon93, KS98, Mil92], but so far only limited forms of combinations of these have been considered. In the case of imperative languages, little effort has been spent in comparing π-calculus semantics to operational semantics of the source languages, as well as in experimenting with the π-calculus semantics to prove properties of the source languages. There are decisive similarities between local names in the π-calculus and local references in imperative languages. This becomes evident in the denotational semantics of these languages: the mathematical techniques employed in modelling the π-calculus [Sta96, FMS96] were originally developed for the semantic description of local references. Yet names and references behave rather differently: receiving from a channel is destructive—it consumes a value—whereas reading from a reference is not; a reference has a unique location, whereas a channel may be used by several processes for both reading and writing; etc. Due to these differences, it is not clear if and how interesting properties of imperative languages can be proved via a translation into the π-calculus.

Section 2 briefly introduces an SOS-style operational semantics for CIA along with a contextual form of behavioural equivalence. Then a π-calculus semantics is presented, together with soundness results for the encoding (Sections 3 and 4). The main part of this paper is devoted to the discussion of concrete examples (Section 5). We prove laws and examples from [MS88, Bro96, MT90a, MT90b], as well as a more complex example involving procedures of higher order, namely the equivalence between two CIA descriptions of two-places buffers (n-place buffers could be dealt with similarly). Then we show that our semantics is not fully abstract (Section 6). We present equivalent CIA phrases, the translations of which are not bisimilar. We show how to remedy the deficiency using I/O types, which refine the ordinary channel types by distinguishing between the capabilities of receiving and sending on a channel. However, as we demonstrate by another example, I/O-types do not suffice for achieving full abstraction. We hint at an even more refined type system which we conjecture does away with the counterexample. It is not clear if such a semantics is already fully abstract (we conjecture it is not) or if further typing constraints would be necessary. Yet we do not pursue full abstraction any further, for two reasons: First, there is a considerable tradeoff between the introduction of more and more sophisticated types on the one hand (leading towards full abstraction), and the applicability to concrete proofs on the other. And, second, our experiments have led us to the conclusion that for the large majority of the cases I/O types are sufficient to prove the desired properties.

2 Concurrent Idealised ALGOL

Concerning syntax, typing and notations for CIA we follow [Rey81, Bro96]. *Data types* consist of integers and booleans; *phrase types* are constructible from variables, expressions and commands using arrow type (for simplicity we omit tupling):

$$\begin{array}{lll} \tau ::= \mathit{int} \mid \mathit{bool} & \qquad & \text{Data Types} \\ \sigma ::= \mathbf{var}[\tau] \mid \mathbf{exp}[\tau] \mid \mathbf{comm} \mid (\sigma \rightarrow \sigma') & \qquad & \text{Phrase Types} \end{array}$$

Data and variable types are lifted up to expression types via the rules

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash v : \mathbf{exp}[\tau]} \qquad \text{and} \qquad \frac{\Gamma \vdash \iota : \mathbf{var}[\tau]}{\Gamma \vdash !\iota : \mathbf{exp}[\tau]}.$$

Variables can be declared on data types only, whereas *procedure definition*, *recursion* and *conditional* are uniformly applicable to all phrase types. An *environment* Γ is a partial function from *identifiers* to types; $\mathrm{dom}(\Gamma)$ denotes the *domain* of Γ.

The syntax is defined according to [Bro96]. However, for defining behavioural equivalences we find it convenient to have explicit constructs for input (on variables) and output (of expressions); alternatively, we could have allowed the observer direct access to the variables (we shall come back to this in Section 7). Further we allow for the use of conditionals in the body of **await** statements. The body of an **await** statement therefore consists of assignments, sequential composition and conditionals. Syntax and typing rules are presented in Table 1 at the end of this paper.

We define an SOS-style operational semantics of CIA, using small-step transition rules (as opposed to a big-step or natural semantics) in order to capture the nondeterministic behaviour resulting from the interaction of phrases via shared variables. The rules are quite standard, with the exception of those needed for modelling the atomicity required by **await**. In the sublanguage without **await** (CIA-{**await**}), the SOS rules are of the form

$$\langle P, \sigma \rangle \xrightarrow{\mathbf{out}\langle v \rangle} \langle P', \sigma' \rangle, \qquad \langle P, \sigma \rangle \xrightarrow{\mathbf{in}\langle v \rangle} \langle P', \sigma' \rangle, \qquad \langle P, \sigma \rangle \xrightarrow{\tau} \langle P', \sigma' \rangle, \qquad \langle P, \sigma \rangle \xrightarrow{\surd} \sigma',$$

where $\mathbf{out}\langle v \rangle$ is the output of value v, $\mathbf{in}\langle v \rangle$ the input of value v, τ is an invisible (internal) actions; and the tick $\surd$ denotes termination (in case of expressions, the tick carries the value resulting by the evaluation). Moreover, P and P' are phrases of variable, expression or command type; σ and σ' are assignments closing up on all free variables of P and P'; finally, P and P' do not contain free identifiers[1]. We call a pair $\langle P, \sigma \rangle$ a *configuration*, and, if P is a command, we call it a *command configuration*.

The command **await** restrains the whole context from running as long as either its condition is being tested, which happens in a busy-wait manner, or, after the condition has been tested positive, during the execution of its body. Therefore the SOS semantics has to provide a means of preventing the application of the transition rules to all commands but **await**. To model this, we introduce *locked configurations* $\langle P, \sigma \rangle_\ell$. The tag ℓ represents a lock; it is to signify that an atomic sequence of steps is being executed. Whenever an **await** statement is executed, the configuration is marked with the lock ℓ, and all but the **await** component are prevented from running (this component is marked itself so to be distinguishable from its context). The lock is released either if the guarding boolean expression has been evaluated to false, or otherwise after the command has been completed. The rules for locked configurations are of the form $\langle P, \sigma \rangle_\ell \xrightarrow{\tau} \langle P', \sigma' \rangle_\ell$; further there are rules for

---

[1] To distinguish variables from identifiers of variable type, we write $\iota, \kappa, \ldots, a, b, \ldots$ for the former, and $x, y, \ldots$ for the latter.

introducing and eliminating the lock from the configurations. Relation $\stackrel{\epsilon}{\Longrightarrow}$ is the reflexive and transitive closure of $\xrightarrow{\tau}$, and $\stackrel{\mu}{\Longrightarrow}$ is given by $\stackrel{\epsilon}{\Longrightarrow}\xrightarrow{\mu}\stackrel{\epsilon}{\Longrightarrow}$ (arbitrarily many invisible steps before and after the μ transition).
Behavioural equality for CIA phrases is defined in two steps: We first apply the (standard) definition of bisimilarity in value-passing process calculi to CIA command configurations (Definition 2.1); then we close this bisimilarity under all (closing) contexts, so to obtain a behavioural equality, *observational congruence*, which is applicable to open phrases as well and is a congruence (Definition 2.2).

Definition 2.1 (Configuration bisimulation) A binary relation $\mathcal{R}$ upon command configurations is a *configuration bisimulation* if it is symmetric and $E_1 \mathcal{R} E_2$ implies,

1. if $E_1 \xrightarrow{\surd} \sigma_1$ then there is σ_2 s.t. $E_2 \stackrel{\surd}{\Longrightarrow} \sigma_2$,
2. if $E_1 \xrightarrow{\tau} E_1'$ then there is E_2' s.t. $E_2 \stackrel{\epsilon}{\Longrightarrow} E_2'$ and $E_1' \mathcal{R} E_2'$,
3. if $E_1 \xrightarrow{\mu} E_1'$ and μ is an output or an input, then there is E_2' s.t. $E_2 \stackrel{\mu}{\Longrightarrow} E_2'$ and $E_1' \mathcal{R} E_2'$.

We write $E_1 \approx E_2$ if there is a configuration bisimulation $\mathcal{R}$ with $E_1 \mathcal{R} E_2$.

We say that a context Con is *closed wrt.* a phrase P if $\emptyset \vdash Con[P] : \mathbf{comm}$ (i.e., $Con[P]$ does not contain free identifiers nor variables).

Definition 2.2 (Observational congruence) Let P_1 and P_2 be arbitrary phrases. Then P_1 and P_2 are *observationally congruent*, written $P_1 \approx_{oc} P_2$, if for every context Con which is closed wrt. P_1 and P_2, $\langle Con[P_1], \emptyset \rangle \approx \langle Con[P_2], \emptyset \rangle$.

Observational congruence is the notion of behavioural equality on CIA phrases we are interested in. It is however hard to prove equalities following its definition, due to the universal quantification over the contexts.
We conclude the section with a useful fact about locked configurations. The behaviour of an **await** statement is deterministic, both due to the absence of parallel composition within its body and to the incapability of expressions to change a given assignment. Therefore locked configurations either diverge or yield bisimilar configurations without a lock. This allows us to abstract from locked configurations in some of the proofs.

Lemma 2.3 *$\langle C, \sigma \rangle_\ell \xrightarrow{\tau} \langle C', \sigma' \rangle_\zeta$ with $\zeta \in \{\ell, \epsilon\}$ implies $\langle C, \sigma \rangle_\ell \approx \langle C', \sigma' \rangle_\zeta$.*

Corollary 2.4 *For every configuration $\langle C, \sigma \rangle_\ell$ the following holds: Either it diverges (i.e., there is an infinite computation of silent steps starting from $\langle C, \sigma \rangle_\ell$) or there is another configuration $\langle C', \sigma' \rangle$ such that $\langle C, \sigma \rangle_\ell \stackrel{\epsilon}{\Longrightarrow} \langle C', \sigma' \rangle$ and $\langle C, \sigma \rangle_\ell \approx \langle C', \sigma' \rangle$.*

3 The π-calculus

We translate CIA into a π-calculus language equipped with a simple type system. The type system has integer, boolean, product and channel types; we omit the typing rules which are quite standard, assuming that all processes and expressions we write are well-typed. *Channels* are used to transmit values; they are ranged over by $a, b, \ldots$. *Variables* are ranged over by $x, y, \ldots$. Channels and variables constitute the *names*, ranged over by $p, q, \ldots$. The symbols $n, m, \ldots$ range over the set of *integer* and *boolean constants*. Channels and constants are the *values*, ranged over by v. An *expression* is either a value, or a variable, or is built up from constants and variables using basic integer and boolean operators like addition, subtraction, complement, etc.; we use $\otimes$ to range over these basic operators. Below we present the grammars for expressions, processes and prefixes; the process constructs are inaction, prefixing, summation, parallel composition, restriction, matching, mismatching, and replication. By abuse of notation ! is not only applied for dereferencing CIA variables (cf. Section 2) but also for replication in the π-calculus. As the meaning of the operator ! will always be clear from the context, we stick to both of these standard notations.

$$\begin{array}{lcll}
e & ::= & v \mid x \mid\ \otimes e \mid e \otimes e & \text{Expressions} \\
\pi & ::= & \overline{p}\langle\tilde{e}\rangle \mid p(\tilde{y}) \mid \tau & \text{Prefix} \\
R & ::= & 0 \mid \pi.R \mid R+R \mid R|R \mid (\nu\, p)R \mid [x=n]R \mid [x \neq n]R \mid\ !p(\tilde{y}).P & \text{Processes.}
\end{array}$$

A process is *closed* if it does not contain free variables. Otherwise it is *open*. For the semantics of the π-calculus we adopt a labelled transition system. The advantage of a labelled semantics, compared to a reduction semantics [Mil91] is that it allows us to define labelled forms of bisimulation and to use the associated proof techniques [MS92]. Process transitions (in the *early style*) are of the form $P \xrightarrow{\mu} P'$, where μ is given by the following syntax:

$$\mu ::= (\nu\,\tilde{b})\overline{a}\langle\tilde{v}\rangle \mid a\langle\tilde{v}\rangle \mid \tau.$$

The label $(\nu\,\tilde{b})\overline{a}\langle\tilde{v}\rangle$ denotes the output of the values $\tilde{v}$ on the name a, where $\tilde{b}$ consists of those channels appearing among the names of $\tilde{v}$ which are private to the sender process. The label $a\langle\tilde{v}\rangle$ denotes the input of values $\tilde{v}$ over the channel a. The action τ represents an internal action.

The rules for the operational semantics are standard rules of the π-calculus. As in typed π-calculi (such as in [Wal95]), there are rules for evaluating an expression to a value, so to be able to infer transitions like $\overline{a}\langle 2+3\rangle.P \xrightarrow{\overline{a}\langle 5\rangle} P$. Weak transitions can be obtained by adding arbitrarily many silent steps before and after a strong transition. We write $\xRightarrow{\epsilon}$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, adopting the standard convention that $\hat{\tau} \stackrel{\text{def}}{=} \epsilon$, and $\hat{\mu} \stackrel{\text{def}}{=} \mu$ for all visible labels μ.

On top of these transitions, *bisimilarity* is defined in the usual way (cf. for instance [MPW89]):

Definition 3.1 (Early bisimulation) A binary relation $\mathcal{R}$ upon closed processes is a *(weak early) bisimulation* if it is symmetric and $R\mathcal{R}S$ implies

$$\text{if } R \xrightarrow{\mu} R' \text{ then there is } S' \text{ s.t. } S \xRightarrow{\hat{\mu}} S' \text{ and } R'\mathcal{R}S'.$$

Two processes R and S are *(weakly early) bisimilar*, written $R \approx_\pi S$, if there is a (weak early) bisimulation $\mathcal{R}$ with $R\mathcal{R}S$.

The definition extends to open processes by closing over all substitutions. In the case of channel variables, however, one can often establish syntactic conditions to avoid the substitution of all channels for a variable, but simply substitute *one* fresh channel for the variable instead [San95a]. This also holds for those processes which we obtain by translating CIA, in Section 4 (we shall not discuss this further in this extended abstract). Also, even though early bisimilarity is not preserved by arbitrary summation, it is preserved by guarded summation; for the semantics of CIA, guarded summation will suffice.

The bisimulation proof technique can be made more powerful by combining it with up-to techniques, like "up to expansion" and "up to injective substitutions" [Mil89, MS92, San95b] (expansion is an asymmetric variant of bisimulation taking into account the number of internal steps performed by the processes [AKH92]).

4 Interpreting Concurrent Idealised ALGOL into the π-calculus

The interpretation of CIA into the π-calculus is defined by the rules of Tables 2 and 3, at the end of this paper. The rules themselves are not surprising: they follow and combine the translations of call-by-name λ-calculus and command compositions into CCS and π-calculus proposed by Milner and colleagues [Mil89, Mil92, MPW89], and extend these to accommodate a blocking construct such as **await**.

The storage is modelled by *registers* of the form (in the π-calculus, recursive process definitions are derivable from replication [Mil91])

$$\mathbf{Reg}_\iota[v] \stackrel{\text{def}}{=} \overline{\mathbf{get}}_\iota\langle v\rangle.\mathbf{Reg}_\iota[v] + \mathbf{put}_\iota(w).\mathbf{Reg}_\iota[w].$$

The pair of names $\mathbf{get}_\iota, \mathbf{put}_\iota$ is abbreviated to fn_ι. The channels $\mathbf{get}_\iota$ and $\mathbf{put}_\iota$ allow processes in the scope of fn_ι to read and modify the content of $\mathbf{Reg}_\iota$. For a concise representation of configurations we write

$$R^\sigma \stackrel{\text{def}}{=} (\nu\,\mathrm{fn}_{\iota_1}, \ldots, \mathrm{fn}_{\iota_n})(\prod_i \mathbf{Reg}_{\iota_i}[\sigma(\iota_i)] \,|\, R),$$

for processes R and finite assignments σ with $\{\iota_i\}_i$ as domain of σ. We present an interpretation into the π-calculus of CIA-$\{$**await**$\}$ before we proceed with full CIA; the latter encoding extends the former with locks to implement the mutual exclusion imposed by **await**.

CIA-$\{$await$\}$

An interpretation of CIA into the π-calculus necessitates certain care even in the absence of **await**, due to the language combining imperative features with higher order. We translate all phrases P into parameterised processes $\mathcal{C}[\![P]\!]_p$, where p is used to signal the termination of the execution of $\mathcal{C}[\![P]\!]_p$. The sequential composition of two commands, for instance, is interpreted as

$$\mathcal{C}[\![C_1; C_2]\!]_p \stackrel{\text{def}}{=} (\nu\, q)(\mathcal{C}[\![C_1]\!]_q \,|\, q.\mathcal{C}[\![C_2]\!]_p).$$

First only $\mathcal{C}[\![C_1]\!]_q$ is able to execute because of name q guarding $\mathcal{C}[\![C_2]\!]_p$. As soon as $\mathcal{C}[\![C_1]\!]_q$ terminates, it signals so on $\overline{q}$ thus releasing $\mathcal{C}[\![C_2]\!]_p$, which is in hold of the termination signal p. Yet another example of sequentiality are declarations. Here first an expression is evaluated, the result of which is then stored in a new location to be accessed by the command in its scope:

$$\mathcal{C}[\![\mathbf{new}\ [\tau]\ \iota := E\ \mathbf{in}\ C]\!]_p \stackrel{\text{def}}{=} (\nu\, q)(\mathcal{C}[\![E]\!]_q \,|\, q(x).(\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[x] \,|\, \mathcal{C}[\![C]\!]_p)).$$

Here not only does parameter q guard $\mathcal{C}[\![C]\!]_p$, but it is also used to transmit the result of the evaluation of $\mathcal{C}[\![E]\!]_q$ to register $\mathbf{Reg}_\iota$. In case E is a value v, we have:

$$\begin{aligned}\mathcal{C}[\![\mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ C]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)(\overline{q}\langle v\rangle.0 \,|\, q(x).(\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[x] \,|\, \mathcal{C}[\![C]\!]_p)) \\ &\approx_\pi (\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \,|\, \mathcal{C}[\![P]\!]_p),\end{aligned}$$

where $\approx_\pi$ is an application of some simple π-calculus laws (precisely the law $(\nu\, q)(\overline{q}\langle v\rangle.R \,|\, q(x).S) \approx_\pi (\nu\, q)(R \,|\, S\{v/x\})$ and the garbage-collection law $(\nu\, q)R \approx_\pi R$ if q is not free in R). Identifiers are modelled by processes sending along a specified channel which is used to invoke a copy of the argument they represent. Both procedural arguments and recursion are translated using replication, so that fresh copies are available each time they are called (recall that CIA is a call-by-name language). For instance, if P is a free identifier, called x_P in the π-calculus translation, then

$$\begin{aligned}&\mathcal{C}[\![\mathbf{new}\ [\mathit{int}]\ \iota := 1\ \mathbf{in}\ P(!\iota)]\!]_p\\ &\qquad\approx_\pi \underbrace{(\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[1]}_{\text{Declaration}} \mid \underbrace{(\nu\,q)(\overline{x}_P\langle q\rangle.0 \mid q(v)}_{\substack{\text{Invoking a copy}\\ \text{of procedure } P}}.\ \underbrace{(\nu\,x)(\overline{v}\langle x,p\rangle}_{\substack{\text{Communicating}\\ \text{argument and}\\ \text{termination}\\ \text{signal}}}\ .\underbrace{!x(r).\mathbf{get}_\iota(z).\overline{r}\langle z\rangle.0)))}_{\text{Procedural argument}}.\end{aligned}$$

There is a close *operational correspondence* between configurations $\langle P, \sigma\rangle$ and their encodings $\mathcal{C}[\![P]\!]_p^\sigma$. For the proof that the interpretation is sound, command configurations $\langle C, \sigma\rangle$ are of particular interest (recall that $\approx$ is defined exactly upon these). Using $\succeq_\pi$ for the expansion relation (cf. Section 3), we give some of the correspondences (the others are similar):

- $\langle C, \sigma\rangle \xrightarrow{\surd} \sigma'$ implies $\mathcal{C}[\![C]\!]_p^\sigma \succeq_\pi (\overline{p}.0)^{\sigma'}$
- $\mathcal{C}[\![C]\!]_p^\sigma \xrightarrow{\overline{p}} R$ implies $R \preceq_\pi 0^{\sigma'}$ and $\langle C, \sigma\rangle \xrightarrow{\surd} \sigma'$
- $\langle C, \sigma\rangle \xrightarrow{\mathbf{out}\langle v\rangle} \langle C', \sigma'\rangle$ implies $\mathcal{C}[\![C]\!]_p^\sigma \xRightarrow{\overline{\mathbf{out}}\langle v\rangle} \mathcal{C}[\![C']\!]_p^{\sigma'}$
- $\mathcal{C}[\![C]\!]_p^\sigma \xrightarrow{\tau} R$ implies either $R \succeq_\pi \mathcal{C}[\![C']\!]_p^{\sigma'}$ such that $\langle C, \sigma\rangle \xRightarrow{\epsilon} \langle C', \sigma'\rangle$,
  or $\mathcal{C}[\![C]\!]_p^\sigma \approx_\pi R \xrightarrow{\overline{\mathbf{out}}\langle v\rangle} \mathcal{C}[\![C']\!]_p^{\sigma'}$ such that $\langle C, \sigma\rangle \xrightarrow{\mathbf{out}\langle v\rangle} \langle C', \sigma'\rangle$.

The operational correspondence relates every possible transition of a configuration and of its encoding. A similar operational correspondence result holds for weak transitions. Exploiting the congruence properties of $\approx_\pi$, the compositionality of the encoding, and the operational correspondence results, we can prove that the encoding is sound. In the proof we also make use of an auxiliary encoding $\mathcal{C}'$ which yields an even closer operational correspondence with CIA, and obtained from $\mathcal{C}$ by removing some "administrative" silent steps. Let $\approx_{oc}^{-}$ be the observational congruence defined on CIA-{**await**} analogously to $\approx_{oc}$ on full CIA.

Theorem 4.1 (Soundness) *$\mathcal{C}[\![P_1]\!]_p \approx_\pi \mathcal{C}[\![P_2]\!]_p$ implies $P_1 \approx_{oc}^{-} P_2$ for arbitrary phrases P_1 and P_2 not containing* **await**.

The converse (completeness) holds in the case of closed commands, but does not extend to arbitrary phrases, as we shall discuss in Section 6.

Full CIA

The encoding $\mathcal{C}_\ell[\![.]\!]$ of phrases in full CIA follows the same compositional scheme as for CIA-{**await**}, for instance

$$\mathcal{C}_\ell[\![C_1; C_2]\!]_p \stackrel{\text{def}}{=} (\nu\,q)(\mathcal{C}_\ell[\![C_1]\!]_q \mid q.\mathcal{C}_\ell[\![C_2]\!]_p).$$

What is different w.r.t. the encoding $\mathcal{C}[\![.]\!]$ is the use of a lock to impose mutual exclusion on input, output, reading from and writing to a variable, and on **await**. Before any of these commands can be executed, the lock has to be acquired; it is released upon their termination. The lock is implemented by a process $\ell.0$. At any time at most one copy of the lock is available to the whole program. Acquiring the lock and continuing as R is modelled by $\overline{\ell}.R$ (the input "requires" the lock);

releasing the lock and continuing as R is translated by $\ell.0 \mid R$ (a new copy of the lock is released). Reading from a variable, for instance, now becomes:

$$\mathcal{C}_\ell[\![!V]\!]_p \overset{\text{def}}{=} (\nu\, q)(\mathcal{C}_\ell[\![V]\!]_q \mid q(gt,pt).\underbrace{\overline{\ell}}_{\text{Take lock}}.gt(x).(\underbrace{\ell.0}_{\text{Release lock}} \mid \overline{p}\langle x\rangle.0))$$

The command **await** is translated following a busy-wait strategy (cf. Table 3). In fact, its encoding is similar to that of the **while** loop (modulo the lock, cf. Table 2), only that a and p change their roles in the bodies of the conditionals.
Our previous example translates to (again we abstract from evaluation and storage of value 1)

$$\begin{aligned}&\mathcal{C}_\ell[\![\mathbf{new}\ [int]\ \iota := 1\ \mathbf{in}\ P(!\iota)]\!]_p\\ &\qquad\approx_\pi (\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[1] \mid (\nu\, q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\, x)(\overline{v}\langle x,p\rangle.!x(r).\overline{\ell}.\mathbf{get}_\iota(z).(\ell.0 \mid \overline{r}\langle z\rangle.0)))).\end{aligned}$$

The differing compilation rules are given in Table 3. The results of operational correspondence and soundness are similar to those in CIA-$\{$**await**$\}$ except that now the π-calculus terms are endowed with and restricted on a lock ℓ. So instead of $\mathcal{C}[\![P]\!]_p$ we now work with processes of the form $(\nu\,\ell)(\ell.0 \mid \mathcal{C}_\ell[\![P]\!]_p)$. (In the operational correspondence, the configurations themselves are not locked, as Corollary 2.4 allows us to abstract from those being locked.) The presence of the lock and its manipulations complicate, however, the inductive reasoning in the previous proofs.

Theorem 4.2 (Soundness) $(\nu\,\ell)(\ell.0 \mid \mathcal{C}_\ell[\![P_1]\!]_p) \approx_\pi (\nu\,\ell)(\ell.0 \mid \mathcal{C}_\ell[\![P_2]\!]_p)$ *implies* $P_1 \approx_{oc} P_2$ *for arbitrary CIA phrases* P_1 *and* P_2.

The following result relates the two translations, $\mathcal{C}[\![.]\!]$ and $\mathcal{C}_\ell[\![.]\!]$.

Theorem 4.3 *In CIA-*$\{$**await**$\}$, $\mathcal{C}[\![P_1]\!]_p \approx_\pi \mathcal{C}[\![P_2]\!]_p$ *implies* $(\nu\,\ell)(\ell.0 \mid \mathcal{C}_\ell[\![P_1]\!]_p) \approx_\pi (\nu\,\ell)(\ell.0 \mid \mathcal{C}_\ell[\![P_2]\!]_p)$ *for arbitrary phrases* P_1 *and* P_2.

As a consequence of Theorems 4.3 and 4.2, on CIA-$\{$**await**$\}$ $\mathcal{C}[\![P_1]\!]_p \approx_\pi \mathcal{C}[\![P_2]\!]_p$ implies $P_1 \approx_{oc} P_2$; this allow us to use the simpler interpretation $\mathcal{C}[\![.]\!]$ for reasoning about phrases that do not contain **await**.

5 Examples of reasoning

We illustrate the use of the π-calculus semantics to validate laws, as well as concrete examples, of CIA. Most of the laws are proved in a purely algebraic way, using some simple well-known π-calculus laws. We first consider laws and examples from [MS88, Bro96, MT90a, MT90b]; one of the goals of this part is to show that the π-calculus semantics yields simple proofs of these well-known equalities. Then we consider a more complex example, involving procedures of higher order.

- Basic properties of CIA operators, such as associativity of sequential composition, or associativity and commutativity of parallel composition, are straightforward consequence of analogous π-calculus laws (like associativity and commutativity of parallel composition in the π-calculus).

- Suppose that ι does not occur free in P', and consider the following laws:

$$\begin{array}{ll}(\mathrm{L1}) & \mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ P' \ = \ P'\\ (\mathrm{L2}) & \mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ (P; P') \ = \ (\mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ P); P'\\ (\mathrm{L3}) & \mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ (P'; P) \ = \ P'; (\mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ P)\\ (\mathrm{L4}) & \mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ (P \parallel P') \ = \ (\mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ P) \parallel P'.\end{array}$$

The π-calculus proofs of these laws are all similar, and purely algebraic. As an example, we present the proof of L2; recall from Section 4 that fn_ι stands for the names $\mathbf{get}_\iota, \mathbf{put}_\iota$:

$$\begin{aligned}
\mathcal{C}\llbracket \mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ (P; P') \rrbracket_p &\approx_\pi (\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid (\nu\, q)(\mathcal{C}\llbracket P \rrbracket_q \mid q.\mathcal{C}\llbracket P' \rrbracket_p)) && (1)\\
&\approx_\pi (\nu\, q)((\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid (\mathcal{C}\llbracket P \rrbracket_q \mid q.\mathcal{C}\llbracket P' \rrbracket_p))) && (2)\\
&\approx_\pi (\nu\, q)((\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid \mathcal{C}\llbracket P \rrbracket_q) \mid q.\mathcal{C}\llbracket P' \rrbracket_p) && (3)\\
&\approx_\pi \mathcal{C}\llbracket (\mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ P); P') \rrbracket_p .
\end{aligned}$$

Line (1) contains the encoding with v already written to $\mathbf{Reg}_\iota$; recall from Section 4 that this process is bisimilar to the original encoding. In (2) the restriction on q is moved to an outer level, whereas in (3) the restriction on fn_ι is removed from $\mathcal{C}\llbracket P' \rrbracket_p$.

- The proof of the following law (validity of β-reduction)

  $$(\lambda(x : \theta).\, P)P' \;=\; P\{P'/x\}$$

  is an extension of the proof of the validity of β-reduction in the π-calculus encoding of the call-by-name λ-calculus [Mil92]; it uses distributivity properties of private replications and structural induction (in this induction, there are more cases to consider w.r.t. the proof of the call-by-name λ-calculus, but the structure of the proof is similar).

- The law $\mathbf{new}\ [\mathit{int}]\ \iota := 1\ \mathbf{in}\ P(!\iota) \;=\; P(1)$ (where P is a free identifier of appropriate type) is proved algebraically:

  $$\begin{aligned}
  &\mathcal{C}\llbracket \mathbf{new}\ [\mathit{int}]\ \iota := 1\ \mathbf{in}\ P(!\iota) \rrbracket_p\\
  &\quad\approx_\pi (\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[1] \mid (\nu\, q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\, x)(\overline{v}\langle x, p\rangle.!x(r).\mathbf{get}_\iota(z).\overline{r}\langle z\rangle.0))) && (1)\\
  &\quad\approx_\pi (\nu\, q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\, x)(\overline{v}\langle x, p\rangle.(\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[1] \mid !x(r).\mathbf{get}_\iota(z).\overline{r}\langle z\rangle.0))) && (2)\\
  &\quad\approx_\pi (\nu\, q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\, x)(\overline{v}\langle x, p\rangle.!x(r).\overline{r}\langle 1\rangle.0)) && (3)\\
  &\quad= \mathcal{C}\llbracket P(1) \rrbracket_p .
  \end{aligned}$$

  Line (1) again contains the encoding with the declaration already performed (cf. Section 4). In the following two lines, $\mathbf{Reg}_\iota$ is eliminated owing to $\mathbf{get}_\iota$ being the only name in fn_ι to be used. Therefore $\mathbf{Reg}_\iota$ is moved into the only subprocess accessing the register; there $\mathbf{Reg}_\iota$ can be eliminated as, in the absence of $\mathbf{put}_\iota$, reading and returning a value is equivalent to just returning the value itself.

- Suppose again that P is a free identifier of appropriate type. Proving the law

  $$\mathbf{new}\ [\mathit{int}]\ \iota := 0\ \mathbf{in}\ P(\iota := !\iota + 1) \;=\; P(\mathbf{skip})$$

  essentially consists in showing that for arbitrary non-negative integer values v,

  $$(\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid !x(r).\mathcal{C}\llbracket \iota := !\iota + 1 \rrbracket_r) \quad \approx_\pi \quad !x(r).\mathcal{C}\llbracket \mathbf{skip} \rrbracket_r ,$$

  where x denotes the formal parameter of P (owing to P being a free identifier, name x is provided by the observer). We sketch the argument of why the two processes are bisimilar. The observer can invoke the argument via x, forking copies of $\mathcal{C}\llbracket \iota := !\iota + 1 \rrbracket$ and $\mathcal{C}\llbracket \mathbf{skip} \rrbracket$ respectively and providing them with a fresh name q on which to signal termination (it suffices to consider *fresh* names, cf. Section 3):

  $$\begin{gathered}
  (\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid !x(r).\mathcal{C}\llbracket \iota := !\iota + 1 \rrbracket_r) \xrightarrow{x\langle q\rangle} (\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid !x(r).\mathcal{C}\llbracket \iota := !\iota + 1 \rrbracket_r \mid \mathcal{C}\llbracket \iota := !\iota + 1 \rrbracket_q)\\
  !x(r).\mathcal{C}\llbracket \mathbf{skip} \rrbracket_r \xrightarrow{x\langle q\rangle} !x(r).\mathcal{C}\llbracket \mathbf{skip} \rrbracket_r \mid \mathcal{C}\llbracket \mathbf{skip} \rrbracket_q ,
  \end{gathered}$$

Assuming that no further procedure calls occur while the present call is being handled (the general case is proved similarly), we have:

$$\begin{array}{lll}
(\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid !x(r).\mathcal{C}\llbracket \iota :=!\iota + 1 \rrbracket_r \mid \mathcal{C}\llbracket \iota :=!\iota + 1 \rrbracket_q) & & \\
 & \overset{\epsilon}{\Longrightarrow} \; (\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v+1] \mid !x(r).\mathcal{C}\llbracket \iota :=!\iota + 1 \rrbracket_r \mid \mathcal{C}\llbracket \mathbf{skip} \rrbracket_q) & (1) \\
 & \xrightarrow{\overline{q}} \; (\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v+1] \mid !x(r).\mathcal{C}\llbracket \iota :=!\iota + 1 \rrbracket_r) & (2) \\
!x(r).\mathcal{C}\llbracket \mathbf{skip} \rrbracket_r \mid \mathcal{C}\llbracket \mathbf{skip} \rrbracket_q & \xrightarrow{\overline{q}} \; !x(r).\mathcal{C}\llbracket \mathbf{skip} \rrbracket_r, & (3)
\end{array}$$

where (1) represents the internal actions performed when incrementing the value stored in $\mathbf{Reg}_\iota$, and (2) and (3) show the signals of termination of the execution of the respective arguments, leaving the processes in the same shape as before.

- A simple π-calculus bisimulation relation can be used to prove that iteration is expressible via recursion, i.e., if x is not free in B and C then

$$\mathbf{while}\ B\ \mathbf{do}\ C \;=\; \mathbf{rec}\, x.\, \mathbf{if}\ B\ \mathbf{then}\ (C;\, x)\ \mathbf{else\ skip}.$$

In our last, more substantial, example we show that two implementations of a *two-place buffer* are equivalent (the example can be generalised to n-place buffers). For simplicity we assume that all buffers store integer values. The example involves both procedures of higher order and the **await** statement. A buffer is a procedure of higher order whose argument x_p intuitively represents the clients of the buffer. This argument x_p is, in turn, a higher-order procedure whose arguments are two procedures: the first (which we shall call **put**) is used by the client to store values in the buffer; the second (which we shall call **get**) is used for retrieving values from the buffer according to a FIFO strategy. Procedure **B** below defines a one-place buffer; x_p represents the clients, x_n a value stored by a client, and x_r is a client location where a value retrieved from the buffer is to be stored. We use sugared notation for the declarations and conditionals.

$$\begin{array}{lll}
\mathbf{B} \overset{\mathrm{def}}{=} & \lambda(x_p : \theta_c).\, \mathbf{new}\ [\mathit{bool}]\, \mathit{fl} := \mathrm{ff}, \mathit{ct} := 0\ \mathbf{in} & \\
 & \quad (x_p\, (\lambda(x_n : \mathit{int}).\, \mathbf{await}\ (!\mathit{fl} = \mathrm{ff})\ \mathbf{then}\ (\mathit{ct} := x_n;\, \mathit{fl} := \mathrm{tt}))) & /*\ \mathbf{put}\ */ \\
 & \qquad (\lambda(x_r : \mathbf{var}[\mathit{int}]).\, \mathbf{await}\ (!\mathit{fl} = \mathrm{tt})\ \mathbf{then}\ (x_r :=!\mathit{ct};\, \mathit{fl} := \mathrm{ff})). & /*\ \mathbf{get}\ */
\end{array}$$

Analogously one can define buffers with two, or even more, places. For instance, the following buffer, $\mathbf{B}_1$, is a two-place buffer. It possesses two local variables, ct_1 and ct_2, for storing values (with priority to ct_1; i.e., ct_2 is to be used only if ct_1 is already full, and the content of ct_1 should be returned before that of ct_2). The counter ib indicates how many values are currently stored.

$$\begin{array}{ll}
\mathbf{B}_1 \overset{\mathrm{def}}{=} & \lambda(x_p : \theta_c).\, \mathbf{new}\ [\mathit{int}]\ \mathit{ib} := 0, \mathit{ct}_1 := 0, \mathit{ct}_2 := 0\ \mathbf{in} \\
 & \quad (x_p\, (\lambda(x_n : \mathit{int}).\, \mathbf{await}\ (!\mathit{ib} \le 1)\ \mathbf{then} \\
 & \qquad\qquad (\mathbf{if}\ (!\mathit{ib} = 0)\ \mathbf{then}\ \mathit{ct}_1 := x_n\ \mathbf{else}\ \mathit{ct}_2 := x_n); \\
 & \qquad\qquad \mathit{ib} :=!\mathit{ib} + 1)) \\
 & \qquad (\lambda(x_r : \mathbf{var}[\mathit{int}]).\, \mathbf{await}\ (!\mathit{ib} \ge 1)\ \mathbf{then} \\
 & \qquad\qquad x_r :=!\mathit{ct}_1; \\
 & \qquad\qquad \mathbf{if}\ (!\mathit{ib} = 2)\ \mathbf{then}\ \mathit{ct}_1 :=!\mathit{ct}_2; \\
 & \qquad\qquad \mathit{ib} :=!\mathit{ib} - 1)).
\end{array}$$

n-place buffers defined like $\mathbf{B}_1$ are single monolithic terms. Yet we can also define n-place buffers in a modular way, by connecting n one-place buffers. In this case, however, it is necessary to distinguish the first $n-1$ buffers from the last, which acts as a barrier buffer. For the barrier buffer, we take the term $\mathbf{B}$ from above; the head buffers $\mathbf{HB}$ are defined as follows:

$$\begin{array}{rl}
\mathbf{HB} \stackrel{\mathrm{def}}{=} & \lambda(x_p : \theta_c).\, \lambda(x_{pt} : \mathit{int} \rightarrow \mathbf{comm}).\, \lambda(x_{gt} : \mathbf{var}[\mathit{int}] \rightarrow \mathbf{comm}). \\
& \quad \mathbf{new}\ [\mathit{bool}]\, \mathit{fl}_h := \mathrm{ff},\ [\mathit{int}]\, \mathit{ct}_h := 0\ \mathbf{in} \\
& \qquad (x_p(\lambda(x_n : \mathit{int}).\, \mathbf{await}\ (\neg ! \mathit{fl}_h)\ \mathbf{then}\ (\mathit{ct}_h := x_n;\, \mathit{fl}_h := \mathrm{tt}))) \\
& \qquad\quad x_{gt} \\
& \quad \parallel \mathbf{rec}\, x.\, (\mathbf{if}\ (!\mathit{fl}_h)\ \mathbf{then}\ (x_{pt}(!\mathit{ct}_h);\, \mathit{fl}_h := \mathrm{ff};\, x)\ \mathbf{else}\ x)
\end{array}$$

Argument x_p represents the clients; arguments x_{pt} and x_{gt} represent the **put** and **get** procedures of the server buffer which $\mathbf{HB}$ is connected to. The arguments for the clients x_p are a **put** procedure defined in $\mathbf{HB}$ itself, and the **get** procedure of the server buffer. The boolean variable fl_h indicates whether $\mathbf{HB}$ is full (in which case a value is currently stored in ct_h). Whenever $\mathbf{HB}$ is full, it attempts to transmit its content to the server buffer, using the **put** procedure of the server. We can then define a 2-place buffer by

$$\mathbf{B}_2 \stackrel{\mathrm{def}}{=} \lambda(x_p : \theta_c).\, \mathbf{B}\, (\mathbf{HB}\, x_p).$$

We now wish to prove that $\mathbf{B}_1$ and $\mathbf{B}_2$ are observationally congruent, i.e., $\mathbf{B}_1 \approx_{oc} \mathbf{B}_2$. We do so by translating them into the π-calculus so to be able to exploit the proof techniques developed for it. First, however, we apply the previously-validated CIA law for β-reduction (F1) and infer $\mathbf{B}_2 \approx_{oc} \mathbf{B}'_2$, where

$$\begin{array}{rl}
\mathbf{B}'_2 \stackrel{\mathrm{def}}{=} & \lambda(x_p : \theta_c).\, \mathbf{new}\ [\mathit{bool}]\, \mathit{fl} := \mathrm{ff},\ \mathit{fl}_h := \mathrm{ff},\ [\mathit{int}]\, \mathit{ct} := 0,\ \mathit{ct}_h := 0\ \mathbf{in} \\
& \qquad (x_p\, (\lambda(x_n : \mathit{int}).\, \mathbf{await}\ (\neg ! \mathit{fl}_h)\ \mathbf{then}\ (\mathit{ct}_h := x_n;\, \mathit{fl}_h := \mathrm{tt}))) \\
& \qquad\quad (\lambda(x_r : \mathbf{var}[\mathit{int}]).\, \mathbf{await}\ (!\mathit{fl})\ \mathbf{then}\ (x_r := !\mathit{ct};\, \mathit{fl} := \mathrm{ff})) \\
& \quad \parallel \mathbf{rec}\, x.\, (\mathbf{if}\ (!\mathit{fl}_h)\ \mathbf{then}\ ((\mathbf{await}\ (\neg ! \mathit{fl})\ \mathbf{then}\ (\mathit{ct} := !\mathit{ct}_h;\, \mathit{fl} := \mathrm{tt}));\, \mathit{fl}_h := \mathrm{ff});\, x).
\end{array}$$

It remains to show that $\mathbf{B}_1 \approx_{oc} \mathbf{B}'_2$. Let $\mathbf{B}_1^{\mathrm{body}}$ and $\mathbf{B}_2^{\mathrm{body}}$ be the bodies of the procedures $\mathbf{B}_1$ and $\mathbf{B}'_2$; they are obtained by stripping off the leading λ. It suffices to prove that the encodings of $\mathbf{B}_1^{\mathrm{body}}$ and $\mathbf{B}_2^{\mathrm{body}}$ are bisimilar. We do so exploiting "bisimulation up to expansion" (cf. Section 3). Due to the presence of **await** we have to use locks, hence the encoding $\mathcal{C}_\ell[\![.]\!]$, and, as required by Theorem 4.2, close the encoding processes under the lock ℓ. Roughly, the bisimulation up to expansion $\mathcal{R}$ which we use for the proof is of the following form (we omit those processes resulting from calls from clients that have not been served immediately):

$$\begin{array}{rll}
\mathcal{R} \stackrel{\mathrm{def}}{=} & \{((\nu\, \ell)(\ell.0 \,|\, \mathcal{C}_\ell[\![\mathbf{B}_1^{\mathrm{body}}]\!]_p),\ (\nu\, \ell)(\ell.0 \,|\, \mathcal{C}_\ell[\![\mathbf{B}_2^{\mathrm{body}}]\!]_p)), & \text{empty buffers} \\
& ((\nu\, \ell)(\ell.0 \,|\, \mathcal{C}_\ell[\![\mathbf{B}_1^{\mathrm{body}}(v)]\!]_p),\ (\nu\, \ell)(\ell.0 \,|\, \mathcal{C}_\ell[\![\mathbf{B}_2^{\mathrm{body}}(v)]\!]_p)), & \text{one value stored} \\
& ((\nu\, \ell)(\ell.0 \,|\, \mathcal{C}_\ell[\![\mathbf{B}_1^{\mathrm{body}}(v,w)]\!]_p),\ (\nu\, \ell)(\ell.0 \,|\, \mathcal{C}_\ell[\![\mathbf{B}_2^{\mathrm{body}}(v,w)]\!]_p)) & \text{two values stored} \\
& \mid v, w : \mathit{int}\}, &
\end{array}$$

where (informally) $\mathbf{B}_1^{\mathrm{body}}(v)$ (resp. $\mathbf{B}_1^{\mathrm{body}}(v,w)$) is like $\mathbf{B}_1^{\mathrm{body}}$ but with a value v (resp. two values v, w) stored in it; similarly for $\mathbf{B}_2^{\mathrm{body}}(v)$ (resp. $\mathbf{B}_2^{\mathrm{body}}(v,w)$).

To see that $\mathcal{R}$ is a bisimulation up to expansion, consider for instance the first pair of the relation, when the buffers are empty, i.e. $!\mathit{ib} = 0$ in $\mathbf{B}_1^{\mathrm{body}}$ and $!\mathit{fl} = !\mathit{fl}_h = \mathrm{ff}$ in $\mathbf{B}_2^{\mathrm{body}}$. In that state the values of ct_1, ct_2, ct and ct_h do not matter, as they cannot be read. Performing corresponding sequences of transitions, the buffers accept a value v from their client and signal the termination of that activity once they have stored it, thus

$$\begin{array}{rcl}(\nu\,\ell)(\ell.0\,|\,\mathcal{C}_\ell[\![\mathbf{B}_1^{\mathrm{body}}]\!]_p) & \stackrel{s}{\Longrightarrow} & (\nu\,\ell)(\ell.0\,|\,\mathcal{C}_\ell[\![\mathbf{B}_1^{\mathrm{body}}(v)]\!]_p)\\ (\nu\,\ell)(\ell.0\,|\,\mathcal{C}_\ell[\![\mathbf{B}_2^{\mathrm{body}}]\!]_p) & \stackrel{s}{\Longrightarrow} & (\nu\,\ell)(\ell.0\,|\,\mathcal{C}_\ell[\![\mathbf{B}_2^{\mathrm{body}}(v)]\!]_p),\end{array}$$

where s is a sequence of visible actions described below. Now $!ib = 1$ in $\mathbf{B}_1^{\mathrm{body}}$ and $!fl = \mathrm{tt}$ in $\mathbf{B}_2^{\mathrm{body}}$; value v is assigned to ct_1 and ct respectively. We can assume this, despite $\mathbf{B}_2^{\mathrm{body}}$ first storing v in ct_h, as

$$(\nu\,\ell)(\ell.0\,|\,\mathcal{C}_\ell[\![\mathbf{B}_2^{\mathrm{body}}(ct_h := v)]\!]_p) \quad \succeq_\pi \quad (\nu\,\ell)(\ell.0\,|\,\mathcal{C}_\ell[\![\mathbf{B}_2^{\mathrm{body}}(ct := v)]\!]_p)$$

($\succeq_\pi$ denotes expansion as introduced in Section 3). Note that this application of the "up to" techniques, which were introduced in Section 3, is vital to the proof of the example (otherwise the relation would yield an extremely large number of pairs). The visible actions determining sequence s consist of: the client requesting that a value be stored (action $x_{pt}\langle r\rangle$, where r is the channel on which the buffers are to signal the termination); the buffers asking for a value to be provided (action $(\nu\,q)\overline{x}_n\langle q\rangle$, where x_n has been agreed upon previously as that channel to be used for invoking **get**, and q a newly created one); the client providing a value (action $q\langle v\rangle$); and, finally, the buffer signalling that v has been stored (action $\overline{r}$). During this execution, the lock is in hold of the buffers. It is released at the same time the client is informed about the termination.

6 Refinements

For certain open CIA phrases, the ordinary π-calculus (weak early) bisimilarity turns out to be too discriminating, i.e., there exist observationally congruent CIA phrases whose translations into the π-calculus yield processes which are not bisimilar. In this section we present two such cases, hinging on the treatment of global variables, and show how to handle them using *types*.

In CIA, reading from a global variable does not influence the overall behaviour of a term as long as the value is not used in future interactions. This is not captured by the usual π-calculus bisimilarity, where all visible actions are treated identically. As a consequence, the equality (where κ is an integer variable)

$$\mathbf{new}\ [int]\ \iota := 0\ \mathbf{in}\ (\iota :=\,!\kappa\ \parallel\ \mathbf{output}\,5) \quad = \quad \mathbf{output}\,5, \tag{1}$$

which is operationally true in CIA, does not yield bisimilar translations into the π-calculus; only the translation of the left-hand term may perform a $\mathbf{get}_\kappa$ transition.

To overcome this problem, we have adopted two measures. If A and B are open CIA phrases with free variables $\{x_i\}_i$, then instead of requiring that $\mathcal{C}[\![A]\!]$ and $\mathcal{C}[\![B]\!]$ be bisimilar, we require bisimilarity between $\prod \mathbf{Reg}_i[\sigma(x_i)]\,|\,\mathcal{C}[\![A]\!]$ and $\prod \mathbf{Reg}_i[\sigma(x_i)]\,|\,\mathcal{C}[\![B]\!]$ (notice that in contrast to Section 4 the registers are not made local by a restriction), where σ is a function mapping all x_i's to some fixed initial value, e.g., 0 and "false". (Using some fixed initial value is possible because, intuitively, program and observer have unlimited access to registers.) To ensure that—aside input and output—communication between program and observer is only possible via these registers we refine the π-calculus types. Refining types makes behavioural equivalences coarser (more process equivalences can be established) because the number of well-typed observers decreases. Precisely, we adopt a type system distinguishing between the capabilities of using a channel in input and output (I/O types, cf. [PS93, BS98]). This way, if ι is a free register, we can assign an external observer only the input capability on $\mathbf{get}_\iota$ and the output capability on $\mathbf{put}_\iota$. This ensures us that a process attempts of reading and writing on register ι are not visible to the external observer.

This type refinement gives us an equivalence on π-calculus processes for which soundness theorems similar to Theorems 4.1 and 4.2 hold, but that is closer to the observational congruence in CIA than the ordinary bisimilarity; the new equivalence allows us to prove (1), as well as equalities like

$$\textbf{while}\ \mathrm{tt}\ \textbf{do}\ (\iota := 0\,;\ \iota := 1) \quad = \quad \textbf{while}\ \mathrm{tt}\ \textbf{do}\ (\iota := 1\,;\ \iota := 0).$$

Again, this equality is valid in CIA but not in the π-calculus for its ordinary bisimilarity. We have proved both this equality and (1) using the π-calculus bisimilarity for I/O types [BS98].

Yet full abstraction is not gained by introducing I/O types. Consider the following example, where P is a free identifier:

$$\begin{array}{lcl} \textbf{new}\ [\mathit{int}]\ \iota := 0\ \textbf{in} & & \textbf{new}\ [\mathit{int}]\ \iota := 0\ \textbf{in} \\ \quad P(\iota); & & \quad P(\iota); \\ \quad \textbf{if}\ (!\iota = 0)\ \textbf{then skip} & = & \quad \textbf{if}\ (!\iota = 0)\ \textbf{then}\ (\textbf{if}\ (!\iota = 1)\ \textbf{then diverge else skip}) \\ \qquad\qquad \textbf{else diverge}. & & \qquad\qquad \textbf{else diverge} \end{array}$$

Suppose the phrases have been signalled the termination of P, and ι is assigned a 0. One would naturally conclude that both phrases should terminate. When translating the phrases into the π-calculus, however, the observer gains access to ι, because P is a free identifier. As a consequence, the observer, after signalling the termination of P, can still write on ι. Now, suppose the variable has already positively been tested for 0. In this case the left-hand phrase is bound to terminate, whereas the right-hand one may still diverge (if the observer sets ι to 1 before the second test).

For validating this example, a yet more refined typing is necessary, namely one which allows us to express linearity (the observer can use certain names only once) and sequentiality (the observer can use a given name only as long he/she does not use another given name) constraints on the use of names. In this way, we can impose a linearity constraint on the return channel of P, as well as a sequentiality constraint that the observer cannot access ι after communicating on the return channel of P. Such a type system can also be used to force the observer to respect the atomicity of **await** statements (before accessing a register, the observer should grab the lock; and release it afterwards). This allows us to validate equivalences such as

$$\textbf{await}\ \mathrm{tt}\ \textbf{then}\ (\kappa := !\kappa + 1;\ \kappa := !\kappa + 1) \quad = \quad \textbf{await}\ \mathrm{tt}\ \textbf{then}\ (\kappa := !\kappa + 2).$$

We see no technical difficulties in adopting such a type system, as we have done with the I/O types. Indeed, type systems for the π-calculus of this kind already exist [Hon96, KPT96, Kob97]; and bisimilarity equivalences for them, and related algebraic properties, can be given by developing those for I/O types. However, even this further type refinement might not give completeness of the interpretation. Moreover, our experiments have led us to the conviction that the I/O types are usually sufficient for reasoning, and that further typing would just make concrete proofs too complex.

7 Further results and discussion

The approach presented in this paper is applicable to other languages with state. We have, e.g., modelled a language that modifies CIA by using call-by-value, instead of call-by-name, and by extending variables to higher order (this implies that not only values but also references and commands be stored in the registers); some of these modifications have been made following the languages in [MT90a, MT90b].

During the execution of an **await** statement, only one thread of computation is active (cf. Section 2 and [Bro96]), yielding a purely sequential behaviour. In order to increase the degree of parallelism in the presence of an active (i.e., currently running) **await** statement, one may, for instance, allow for a simultaneous execution of phrases which do not access variables affected by the **await** statement. In the SOS semantics, this would mean that locks carry along information about the variables involved in the **await** statement currently executed. In the π-calculus semantics, it can be modelled by imposing multiple locks, instead of a single one. The necessary information on the access to variables can be gained by some simple preliminary static analysis. Of course, such an increase in parallelism changes the overall semantics; nevertheless there are behavioural correspondences between the more sequential and the more parallel version: First, if two phrases are bisimilar in the more parallel version, then they are also bisimilar in the sequential one (cutting off branches from the transition systems). Second, a phrase may yield a divergent computation (transition trace) in the sequential semantics if and only if it does so in the parallel one (transitions occurring interleaved in the parallel semantics are *causally independent*, so they can be interchanged resulting in a computation of the sequential semantics). We have proved both these results by reasoning on the π-calculus translations.

We have considered closed only those programs that do not possess open identifiers *nor* variables, and have thus introduced explicit input and output constructs to CIA. An alternative approach is to provide the observer with direct access to the global variables via **read** and **write** constructs:

$$\begin{array}{ll}\langle P,\sigma\rangle \xrightarrow{\mathbf{read}_\iota(v)} \langle P,\sigma\rangle & \text{if } \Gamma(\iota)=\mathbf{var}[\tau] \text{ and } \sigma(\iota)=v,\\ \langle P,\sigma\rangle \xrightarrow{\mathbf{write}_\iota(v)} \langle P,\sigma\{\iota\leftarrow v\}\rangle & \text{if } \Gamma(\iota)=\mathbf{var}[\tau].\end{array}$$

To obtain operational correspondence and soundness (cf. Section 4), the translation into the π-calculus would have to take into account I/O types (cf. discussion in Section 6).

Semantics given to IA and CIA by O'Hearn and Tennent [OT95], Pitts [Pit96] and Brookes [Bro96], make use of relational parametricity. Comparing the proofs of laws and examples in IA and CIA conducted in these theories, with bisimulation-based proofs carried out in the π-calculus might clarify the relationship between these two notions.

Acknowledgements

We would like to thank J. Esparza, P. W. O'Hearn and U. S. Reddy for helpful discussions.
This work was partly supported by Teilprojekt A3 SAM of SFB 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", and by the PROCOPE project 9723064.

References

[AKH92] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. *Acta Informatica*, 29:737–760, 1992.

[AM96] S. Abramsky and G. McCusker. Linearity, sharing and state : a fully abstract game semantics for idealized algol with active expressions. *Electronic Notes in Theoretical Computer Science*, 3, 1996. `http://www.elsevier.nl/locate/entcs/volume3.html`.

[AM97] S. Abramsky and G. McCusker. Full abstraction for idealized algol with passive expressions. Submitted for Publication, 1997.

[Bro96] S. Brookes. The essence of parallel algol. In *Proc. LICS'96*. IEEE, 1996. App. in vol. 2 of [OT97].

[BS98] M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. To appear in *Proc. LICS '98*, 1998.

[FHJ95] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. Tr 95:05, School of Cognitive and Computing Sciences, University of Sussex, 1995.

[FMS96] M. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the π-calculus. In *eleventh LICS Conf.* IEEE Computer Society Press, 1996.

[Hon96] K. Honda. Composing processes. In *Proc. 23th POPL*. ACM Press, 1996.

[How96] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[Jon93] C. B. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *LNCS*, pages 158–172. Springer Verlag, 1993.

[Kob97] N. Kobayashi. A partially deadlock-free typed process calculus. In *Proc. 12th LICS Conf.* IEEE Computer Society Press., 1997.

[KPT96] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the π-calculus. In *Proc. 23rd POPL*. ACM Press, 1996.

[KS98] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *Proc. PROCOMET'98*, North-Holland, 1998.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Tech. Rep. ECS–LFCS–91–180, LFCS, 1991.

[Mil92] R. Milner. Functions as processes. *J. of Math. Struct. in Computer Science*, 17:119–141, 1992.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Report ECS-LFCS-89-85,86, Dept. of Computer Science, University of Edinburgh, 1989. 2 volumes, appeared in Information and Computation 100:1–77,1992.

[MS88] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In *Conf. Record 15th ACM Symposium on Principles of Programming Languages*, 1988. Also appeared in vol. 2 of [OT97].

[MS92] R. Milner and D. Sangiorgi. The problem of weak bisimulation up-to. In *Proc. CONCUR'92*, volume 630 of *LNCS*. Springer, 1992.

[MT90a] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. Tech. Rep., University of Stanford, 1990.

[MT90b] I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. Tech. Rep., University of Stanford, 1990.

[OT95] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995. Also appeared in [OT97].

[OT97] P. W. O'Hearn and R. D. Tennent, editors. *ALGOL-like Languages*. Progress in Theoretical Computer Science. Birkhäuser, 1997. 2 volumes.

[Pit96] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *Proc. LICS'96*. IEEE, 1996. Also appeared in vol. 2 of [OT97].

[PS93] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Proc. LICS'93*. IEEE, 1993. Also appeared in Mathematical Structures in Computer Science 6:5(1996) pp. 409–453.

[Rey81] J. C. Reynolds. The essence of ALGOL. In *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981. Also appeared in vol. 1 of [OT97].

[San95a] Davide Sangiorgi. Lazy functions and mobile processes. To appear in "Festschrift volume in honor of Robin Milner's 60th birthday", MIT Press.

[San95b] Davide Sangiorgi. On the proof method for bisimulation (extended abstract). In *Proc. MFCS'95*, volume 969 of *LNCS*, pages 479–488. Springer Verlag, 1995. Full version available electronically.

[Sta96] Ian Stark. A fully abstract domain model for the π-calculus. In *Proc. LICS'96*, pages 36–42. IEEE Computer Society Press, 1996.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.

$$\Gamma \vdash v : \tau$$

$$\Gamma \vdash \iota : \mathbf{var}[\tau] \quad \text{when } \Gamma(\iota) = \mathbf{var}[\tau]$$

$$\frac{\Gamma \vdash E_1 : \mathbf{exp}[\tau] \quad \Gamma \vdash E_2 : \mathbf{exp}[\tau]}{\Gamma \vdash E_1 \otimes E_2 : \mathbf{exp}[\tau]} \; \otimes : \tau \times \tau \to \tau$$

$$\Gamma \vdash \mathbf{skip} : \mathbf{comm}$$

$$\frac{\Gamma \vdash E : \mathbf{exp}[\tau]}{\Gamma \vdash \mathbf{output}\, E : \mathbf{comm}}$$

$$\frac{\Gamma \vdash \iota : \mathbf{var}[\tau]}{\Gamma \vdash \mathbf{input}\, \iota : \mathbf{comm}}$$

$$\frac{\Gamma \vdash V : \mathbf{var}[\tau] \quad \Gamma \vdash E : \mathbf{exp}[\tau]}{\Gamma \vdash V := E : \mathbf{comm}}$$

$$\frac{\Gamma \vdash C_1 : \mathbf{comm} \quad \Gamma \vdash C_2 : \mathbf{comm}}{\Gamma \vdash C_1 ; C_2 : \mathbf{comm}}$$

$$\frac{\Gamma \vdash C_1 : \mathbf{comm} \quad \Gamma \vdash C_2 : \mathbf{comm}}{\Gamma \vdash C_1 \parallel C_2 : \mathbf{comm}}$$

$$\frac{\Gamma \vdash B : \mathbf{exp}[\mathit{bool}] \quad \Gamma \vdash P_1 : \theta \quad \Gamma \vdash P_2 : \theta}{\Gamma \vdash \mathbf{if}\, B\, \mathbf{then}\, P_1\, \mathbf{else}\, P_2 : \theta}$$

$$\frac{\Gamma \vdash B : \mathbf{exp}[\mathit{bool}] \quad \Gamma \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{while}\, B\, \mathbf{do}\, C : \mathbf{comm}}$$

$$\frac{\Gamma \vdash B : \mathbf{exp}[\mathit{bool}] \quad \Gamma \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{await}\, B\, \mathbf{then}\, C : \mathbf{comm}} \; C \text{ atomic with conditionals}$$

$$\frac{\Gamma \vdash E : \mathbf{exp}[\tau] \quad \Gamma, \iota : \mathbf{var}[\tau] \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{new}\, [\tau]\iota := E\, \mathbf{in}\, C : \mathbf{comm}}$$

$$\Gamma \vdash x : \theta \quad \text{when } \Gamma(x) = \theta$$

$$\frac{\Gamma, x : \theta \vdash P : \theta}{\Gamma \vdash \mathbf{rec}\, x.P : \theta}$$

$$\frac{\Gamma, x : \theta \vdash P : \theta'}{\Gamma \vdash \lambda(x : \theta).P : (\theta \to \theta')}$$

$$\frac{\Gamma \vdash P_1 : (\theta \to \theta') \quad \Gamma \vdash P_2 : \theta}{\Gamma \vdash P_1(P_2) : \theta'}$$

Table 1: Syntax and typing of CIA

$$
\begin{aligned}
\mathcal{C}\llbracket \iota \rrbracket_p &\overset{\mathrm{def}}{=} \overline{p}\langle \mathbf{get}_\iota, \mathbf{put}_\iota \rangle.0 \\
\mathcal{C}\llbracket v \rrbracket_p &\overset{\mathrm{def}}{=} \overline{p}\langle v \rangle.0 \\
\mathcal{C}\llbracket !V \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q)(\mathcal{C}\llbracket V \rrbracket_q \,|\, q(gt, pt).gt(x).\overline{p}\langle x \rangle.0) \\
\mathcal{C}\llbracket E_1 \otimes E_2 \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q, r)(\mathcal{C}\llbracket E_1 \rrbracket_q \,|\, \mathcal{C}\llbracket E_2 \rrbracket_r \,|\, q(x).r(y).\overline{p}\langle x \otimes y \rangle.0) \\
\mathcal{C}\llbracket \mathbf{skip} \rrbracket_p &\overset{\mathrm{def}}{=} \overline{p}.0 \\
\mathcal{C}\llbracket \mathbf{output}\, E \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q)(\mathcal{C}\llbracket E \rrbracket_q \,|\, q(x).\overline{\mathbf{out}}\langle x \rangle.\overline{p}.0) \\
\mathcal{C}\llbracket \mathbf{input}\, \iota \rrbracket_p &\overset{\mathrm{def}}{=} \mathbf{in}(x).\overline{\mathbf{put}}_\iota\langle x \rangle.\overline{p}.0 \\
\mathcal{C}\llbracket V := E \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q, r)(\mathcal{C}\llbracket V \rrbracket_q \,|\, \mathcal{C}\llbracket E \rrbracket_r \,|\, q(gt, pt).r(x).\overline{pt}\langle x \rangle.\overline{p}.0) \\
\mathcal{C}\llbracket C_1;\, C_2 \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q)(\mathcal{C}\llbracket C_1 \rrbracket_q \,|\, q.\mathcal{C}\llbracket C_2 \rrbracket_p) \\
\mathcal{C}\llbracket C_1 \parallel C_2 \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q, r)(\mathcal{C}\llbracket C_1 \rrbracket_q \,|\, \mathcal{C}\llbracket C_2 \rrbracket_r \,|\, q.r.\overline{p}.0) \\
\mathcal{C}\llbracket \mathbf{if}\, B\, \mathbf{then}\, P_1\, \mathbf{else}\, P_2 \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q)(\mathcal{C}\llbracket B \rrbracket_q \,|\, q(x).([x = \mathrm{tt}]\, \mathcal{C}\llbracket P_1 \rrbracket_p \,|\, [x = \mathrm{ff}]\, \mathcal{C}\llbracket P_2 \rrbracket_p)) \\
\mathcal{C}\llbracket \mathbf{while}\, B\, \mathbf{do}\, C \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, a)(!a.\, (\nu\, q)(\mathcal{C}\llbracket B \rrbracket_q \,|\, q(x).\, (\, [x = \mathrm{tt}]\, (\nu\, r)(\mathcal{C}\llbracket C \rrbracket_r \,|\, r.\overline{a}.0) \,| \\
&\qquad\qquad\qquad\qquad\qquad\qquad [x = \mathrm{ff}]\, \overline{p}.0)) \,|\, \overline{a}.0) \\
\mathcal{C}\llbracket \mathbf{new}\, [\tau]\, \iota := E\, \mathbf{in}\, C \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q)(\mathcal{C}\llbracket E \rrbracket_q \,|\, q(x).(\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[x] \,|\, \mathcal{C}\llbracket C \rrbracket_p)) \\
\mathcal{C}\llbracket x \rrbracket_p &\overset{\mathrm{def}}{=} \overline{x}\langle p \rangle.0 \\
\mathcal{C}\llbracket \mathbf{rec}\, x.\, P \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, x)(!x(r).\mathcal{C}\llbracket P \rrbracket_r \,|\, \overline{x}\langle p \rangle.0) \\
\mathcal{C}\llbracket \lambda(x : \theta).\, P \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, v)(\overline{p}\langle v \rangle.v(x, q).\mathcal{C}\llbracket P \rrbracket_q) \\
\mathcal{C}\llbracket P_1 P_2 \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q)(\mathcal{C}\llbracket P_1 \rrbracket_q \,|\, q(v).(\nu\, x)(\overline{v}\langle x, p \rangle.!x(r).\mathcal{C}\llbracket P_2 \rrbracket_r))
\end{aligned}
$$

Table 2: Encoding CIA-{**await**} in the π-calculus

$$
\begin{aligned}
\mathcal{C}_\ell\llbracket !V \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q)(\mathcal{C}_\ell\llbracket V \rrbracket_q \,|\, q(gt, pt).\overline{\ell}.gt(x).(\ell.0 \,|\, \overline{p}\langle x \rangle.0)) \\
\mathcal{C}_\ell\llbracket \mathbf{output}\, E \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q)(\mathcal{C}_\ell\llbracket E \rrbracket_q \,|\, q(x).\overline{\ell}.\overline{\mathbf{out}}\langle x \rangle.(\ell.0 \,|\, \overline{p}.0)) \\
\mathcal{C}_\ell\llbracket \mathbf{input}\, \iota \rrbracket_p &\overset{\mathrm{def}}{=} \overline{\ell}.\mathbf{in}(x).(\ell.0 \,|\, \overline{\ell}.\overline{\mathbf{put}}_\iota\langle x \rangle.(\ell.0 \,|\, \overline{p}.0)) \\
\mathcal{C}_\ell\llbracket V := E \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, q, r)(\mathcal{C}_\ell\llbracket V \rrbracket_q \,|\, \mathcal{C}_\ell\llbracket E \rrbracket_r \,|\, q(gt, pt).r(x).\overline{\ell}.\overline{pt}\langle x \rangle.(\ell.0 \,|\, \overline{p}.0)) \\
\mathcal{C}_\ell\llbracket \mathbf{await}\, B\, \mathbf{then}\, C \rrbracket_p &\overset{\mathrm{def}}{=} (\nu\, a)(!a.\, (\nu\, q)(\overline{\ell}.\mathcal{C}\llbracket B \rrbracket_q \,|\, q(x).\, (\, [x = \mathrm{tt}]\, (\nu\, r)(\mathcal{C}\llbracket C \rrbracket_r \,|\, r.\, (\ell.0 \,|\, \overline{p}.0)) \,| \\
&\qquad\qquad\qquad\qquad\qquad\qquad [x = \mathrm{ff}]\, (\ell.0 \,|\, \overline{a}.0))) \,|\, \overline{a}.0)
\end{aligned}
$$

Table 3: Encoding Full CIA in the π-calculus – Modifications to Table 2