A π-calculus Process Semantics of Concurrent Idealised ALGOL

Christine Röckl[1] and Davide Sangiorgi[2]

[1] Technische Universität München, D-80290 München, roeckl@in.tum.de
[2] INRIA–Sophia Antipolis, F-06902 Sophia Antipolis, davide.sangiorgi@inria.fr

Abstract. We study the use of the π-calculus for semantical descriptions of languages such as *Concurrent Idealised ALGOL* (CIA), combining imperative, functional and concurrent features. We first present an operational semantics for CIA, given by 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, demonstrating that there is a close operational correspondence between them and that the π-calculus semantics is sound. This allows for applying the π-calculus theory in proving behavioural properties of CIA phrases. We discuss laws and examples which have served as benchmarks to various semantics, and a more complex example involving procedures of higher order.

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 features 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, since it combines imperative, functional and concurrent features, and possesses an atomicity construct.

In this paper we study semantics of CIA given by a translation into the π-calculus. The main reasons for using the π-calculus are the following. It offers a well-developed theory that we wish to exploit, through the translation, to reason on CIA terms. We also intend to profit from the π-calculus being, syntactically, a first-order language, i.e., values only consist of names (in typed versions, there may also be basic values such as integers and booleans). In contrast, CIA is higher-order, thus values may be arbitrary terms. In higher-order languages, defining satisfactory notions of behavioural equivalences—not to mention proof techniques for them—may be hard. Proofs of process equivalences are complicated by universal quantifications over terms. Further, it is in general hard to establish that a notion of bisimilarity is a congruence. (For higher-order languages,

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 306-321, 1999.

this is usually proved using Howe's technique [How96]; attempts to extend this technique to languages with local state, however, 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; models representing states by functions—usually denotational models do so—suffer from snapback effects, which are usually removed by means of logical relations [OT97].

Our study is also motivated by the question of how appropriate the π-calculus is 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 to operational semantics, and in using π-calculus translations for proving properties of the source languages. Denotational approaches indicate a strong similarity between local names in the π-calculus and local references in imperative languages; note that 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. Hence it is unclear 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 handle these examples using types, especially I/O-types. It is unclear whether the type systems we propose already yield full abstraction (we conjecture they do not). Yet introducing more and more sophisticated types deteriorates the applicability to concrete proofs. However, our experiments have led us to the conclusion that in most cases I/O types suffice.

2 Concurrent Idealised ALGOL

Syntax, typing and notations for CIA closely 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):

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

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, with *domain* $\mathrm{dom}(\Gamma)$.
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**. Let P and P' be phrases of variable, expression or command type which do not contain free identifiers; σ and σ' are assignments closing up on all free variables of P and P'. We call a pair $\langle P, \sigma\rangle$ a *configuration*, and, if P is a command, we call it a *command configuration*. In the sublanguage without **await** (CIA-{**await**}), the SOS rules are of the form

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

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) action; and the tick $\surd$ denotes termination. If P is an expression, the tick carries the value resulting from its evaluation.
The command **await** guarantees for an atomic execution of a sequential composition of assignments and conditionals once its guard has been evaluated to true (an evaluation to false results in a repetition after some period of busy-waiting). During the evaluation of the guard and the execution of the body of an **await** statement, any other computation has to be stopped. We achieve this by introducing *locked configurations* $\langle P,\sigma\rangle_\ell$. The tag ℓ represents a lock. 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 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 is defined in two steps: We first apply the (standard) definition of bisimilarity in value-passing process calculi to CIA command configu-

rations (Definition 1); then, by closing it under all (closing) contexts, we obtain an *observational congruence* applicable to all phrase types (Definition 2).

Definition 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 (Observational congruence). Let P_1, 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 the incapability of expressions to change a given assignment.

Lemma 1. $\langle C, \sigma \rangle_\ell \xrightarrow{\tau} \langle C', \sigma' \rangle_\zeta$ *with* $\zeta \in \{\ell, \epsilon\}$ *implies* $\langle C, \sigma \rangle_\ell \approx \langle C', \sigma' \rangle_\zeta$.

Corollary 1. *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 supplied with a simple type system. This type system provides 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 $x, y, \ldots$. Together, channels and variables constitute the *names*, $p, q, \ldots$. *Integer* and *boolean constants* are denoted by $n, m, \ldots$. Channels and constants are the *values*, ranged over by v. $\otimes$ denotes basic operators like addition, subtraction, complement, etc.

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

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. In contrast to reduction semantics [Mil91], this allows us to use 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

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

$(\nu\, \tilde{b})\overline{a}\langle\tilde{v}\rangle$ denotes the output of the values $\tilde{v}$ on the name a, where $\tilde{b}$ are those channels among the names of $\tilde{v}$ which are private to the sender process; $a\langle\tilde{v}\rangle$ is the input of values $\tilde{v}$ over the channel a; finally, τ represents an internal action. We use the standard SOS 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 $\stackrel{\epsilon}{\Longrightarrow}$ 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.
Bisimilarity is defined in the usual way (cf. for instance [MPW89]):

Definition 3 (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

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

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, which suffices in our case. 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 CIA in the π-calculus

The π-calculus interpretation of CIA is given by the rules in Tables 2 and 3 at the end of this paper. The storage is modelled by *registers* of the form (in the π-calculus, recursive process definitions are derivable from replication [Mil91])

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

Processes in the scope of $\mathrm{fn}_\iota \stackrel{\text{def}}{=} \{\mathbf{get}_\iota, \mathbf{put}_\iota\}$ are allowed to read and modify the content of $\mathbf{Reg}_\iota$. Configurations $\langle C, \sigma\rangle$, with $\Gamma(\sigma) = \{\iota_i\}_i$, translate to

$$[\![\langle C, \sigma\rangle]\!]_p \stackrel{\text{def}}{=} (\nu\, \mathrm{fn}_{\iota_1}, \ldots, \mathrm{fn}_{\iota_n})(\prod_i \mathbf{Reg}_{\iota_i}[\sigma(\iota_i)] \mid [\![C]\!]_p).$$

CIA-{await}

A π-calculus interpretation of CIA 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 $[\![P]\!]_p$; the fresh name p is used to signal the termination of the execution of $[\![P]\!]_p$. The sequential composition of two commands, for instance, is written as

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

First only $[\![C_1]\!]_q$ is able to execute because of name q guarding $[\![C_2]\!]_p$. As soon as $[\![C_1]\!]_q$ terminates, it signals so on $\overline{q}$ thus releasing $[\![C_2]\!]_p$. Yet another example of sequentiality are declarations,

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

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

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

where $\approx_\pi$ is an application of some simple π-calculus laws (precisely the law $(\nu\, q)(\overline{q}\langle v\rangle.R \mid q(x).S) \approx_\pi (\nu\, q)(R \mid S\{v/x\})$ 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 fresh copies are available at every call (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

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

There is a close *operational correspondence* between configurations $\langle P, \sigma\rangle$ and their encodings $[\![\langle P, \sigma\rangle]\!]_p$. 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 $[\![\langle C,\sigma\rangle]\!]_p \succeq_\pi (\overline{p}.0)^{\sigma'}$;
- $[\![\langle C,\sigma\rangle]\!]_p \xrightarrow{\overline{p}} R$ implies $R \succeq_\pi 0$ and $\langle C,\sigma\rangle \xrightarrow{\surd} \sigma'$;
- $\langle C,\sigma\rangle \xrightarrow{\mathbf{out}\langle v\rangle} \langle C',\sigma'\rangle$ implies $[\![\langle C,\sigma\rangle]\!]_p \overset{\overline{\mathbf{out}}\langle v\rangle}{\Longrightarrow} [\![\langle C',\sigma'\rangle]\!]_p$;
- $[\![\langle C,\sigma\rangle]\!]_p \xrightarrow{\tau} R$ implies either $R \succeq_\pi [\![\langle C',\sigma'\rangle]\!]_p$ such that $\langle C,\sigma\rangle \overset{\epsilon}{\Longrightarrow} \langle C',\sigma'\rangle$, or $[\![\langle C,\sigma\rangle]\!]_p \approx_\pi R \xrightarrow{\overline{\mathbf{out}}\langle v\rangle} [\![\langle C',\sigma'\rangle]\!]_p$ 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 is obtained from $\mathcal{C}$ by removing some "administrative" silent steps.
Let $\approx_{oc}^{-}$ be the observational congruence on CIA-{**await**} defined analogously to $\approx_{oc}$ on full CIA.

Theorem 1 (Soundness). $[\![P_1]\!]_p \approx_\pi [\![P_2]\!]_p$ *implies* $P_1 \approx_{oc}^{-} P_2$ *for arbitrary CIA-*{**await**} *phrases* P_1 *and* P_2.

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

$$[\![C_1;C_2]\!]^\ell_p \stackrel{\text{def}}{=} (\nu\, q)([\![C_1]\!]^\ell_q \mid q.[\![C_2]\!]^\ell_p).$$

What is different wrt. the encoding $[\![.]\!]$ 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:

$$[\![!V]\!]^\ell_p \stackrel{\text{def}}{=} (\nu\, q)([\![V]\!]^\ell_q \mid q(gt,pt).\underbrace{\overline{\ell}}_{\text{Take lock}}.gt(x).(\underbrace{\ell.0}_{\text{Release lock}} \mid \overline{p}\langle x\rangle.0))$$

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

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

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 contain a lock ℓ. So, instead of $[\![P]\!]_p$ we now work with processes of the form $(\nu\,\ell)(\ell.0 \mid [\![P]\!]_p^\ell)$. (In the operational correspondence, the configurations themselves are not locked, as Corollary 1 allows us to abstract from those being locked.)

Theorem 2 (Soundness). $(\nu\,\ell)(\ell.0 \mid [\![P_1]\!]_p^\ell) \approx_\pi (\nu\,\ell)(\ell.0 \mid [\![P_2]\!]_p^\ell)$ *implies* $P_1 \approx_{oc} P_2$ *for arbitrary CIA phrases* P_1 *and* P_2.

The following result relates the two translations, $[\![.]\!]$ and $[\![.]\!]^\ell$, which allows us to use the simpler encoding in the absence of **await**.

Theorem 3. $[\![P_1]\!]_p \approx_\pi [\![P_2]\!]_p$ *implies* $(\nu\,\ell)(\ell.0 \mid [\![P_1]\!]_p^\ell) \approx_\pi (\nu\,\ell)(\ell.0 \mid [\![P_2]\!]_p^\ell)$, *and thus* $P_1 \approx_{oc} P_2$, *for arbitrary CIA-*{**await**} *phrases* P_1 *and* P_2.

5 Examples of reasoning

Considering benchmark laws and examples from [MS88,Bro96,MT90a,MT90b], we demonstrate that the π-calculus semantics yields simple proofs of these well-known equalities. Further we show by a more complex example how to tackle procedures of higher order.

• Basic properties of CIA operators, such as associativity of sequential composition, or associativity and commutativity of parallel composition, are straightforward consequences 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:

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

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 $\mathrm{fn}_\iota \stackrel{\mathrm{def}}{=} \{\mathbf{get}_\iota, \mathbf{put}_\iota\}$:

$$\begin{array}{lll}
[\![\mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,(P;P')]\!]_p & \approx_\pi (\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid (\nu\,q)([\![P]\!]_q \mid q.[\![P']\!]_p)) & (1) \\
& \approx_\pi (\nu\,q)((\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid ([\![P]\!]_q \mid q.[\![P']\!]_p))) & (2) \\
& \approx_\pi (\nu\,q)((\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid [\![P]\!]_q) \mid q.[\![P']\!]_p) & (3) \\
& \approx_\pi [\![(\mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,P);P')]\!]_p. &
\end{array}$$

Line (1) contains the encoding with v already written to $\mathbf{Reg}_\iota$; in Section 4 we have shown that this process is bisimilar to the original encoding. In (2) the restriction on q is moved to an outer level, and in (3) the restriction on fn_ι is removed from $[\![P']\!]_p$.

• The proof of the law $(\lambda(x:\theta).\,P)P' \;=\; P\{P'/x\}$ (validity of β-reduction) 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 wrt. the proof of the call-by-name λ-calculus, but the structure of the proof is similar).

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

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

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

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

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

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

where x denotes the formal parameter of P (owing to P being a free identifier, name x is provided by the observer).

• 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

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

• 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. 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.

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

Analogously one can define buffers with two, or even more, places. Buffer $\mathbf{B}_1$ below, e.g., is a two-place buffer. It possesses local variables ct_1 and ct_2, for storing values, and a counter ib to indicate how many values are currently stored.

$$\begin{aligned}
\mathbf{B}_1 \stackrel{\mathrm{def}}{=}\ & \lambda(x_p : \theta_c).\,\mathbf{new}\ [int]\,ib := 0,\, ct_1 := 0,\, ct_2 := 0\ \mathbf{in} \\
& \quad (x_p(\lambda(x_n : int).\,\mathbf{await}\ (!ib \le 1)\ \mathbf{then} \\
& \qquad\qquad (\mathbf{if}\ (!ib = 0)\ \mathbf{then}\ ct_1 := x_n\ \mathbf{else}\ ct_2 := x_n); \\
& \qquad\qquad ib :=!ib + 1)) \\
& \qquad (\lambda(x_r : \mathbf{var}[int]).\,\mathbf{await}\ (!ib \ge 1)\ \mathbf{then} \\
& \qquad\qquad x_r :=!ct_1; \\
& \qquad\qquad \mathbf{if}\ (!ib = 2)\ \mathbf{then}\ ct_1 :=!ct_2; \\
& \qquad\qquad ib :=!ib - 1)).
\end{aligned}$$

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:

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

Again, x_p represents the clients; 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

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

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

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

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, by "bisimulation up to expansion" (cf. Section 3), that the encodings of $\mathbf{B}_1^{\mathrm{body}}$ and $\mathbf{B}_2^{\mathrm{body}}$ are bisimilar. Due to the presence of **await** we have to use locks, hence the encoding $[\![.]\!]^{\ell}$, and, as required by Theorem 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):

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

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. values v, w) stored in it; similarly for $\mathbf{B}_2^{\mathrm{body}}(v)$ (resp. $\mathbf{B}_2^{\mathrm{body}}(v,w)$). Consider for instance the first pair of the relation; here the buffers are empty, i.e., $!ib = 0$ 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. With corresponding

sequences of transitions, s, the buffers accept a value v from their client and, after storing it, signal the termination of that activity, thus

$$(\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\mathrm{body}}]\!]_p^\ell) \stackrel{s}{\Longrightarrow} (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\mathrm{body}}(v)]\!]_p^\ell)$$
$$(\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\mathrm{body}}]\!]_p^\ell) \stackrel{s}{\Longrightarrow} (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\mathrm{body}}(v)]\!]_p^\ell).$$

Precisely s is a sequence of visible actions consisting of: the client requesting that a value be stored ($x_{pt}\langle r\rangle$, where r will be used to signal the termination, see below); the buffers asking for a value (action $(\nu\,q)\overline{x}_n\langle q\rangle$, where x_n is a previously agreed channel to be used for invoking **get**, and q is 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 buffers hold the lock; it is released at the same time the client is informed of the termination.
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

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

($\succeq_\pi$ denotes expansion as introduced in Section 3). Note that this application of the "up to" techniques is vital to the proof of the example (otherwise the relation would yield an extremely large number of pairs).
We do not know how to prove this or the previous examples directly in the operational semantics of ALGOL without going through a universal quantification over contexts (recall the problems with reasoning directly within the ALGOL semantics, discussed in the Introduction).

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. Refining types, however, makes behavioural equivalences coarser (more process equivalences can be established), simply because the number of well-typed observers decreases.
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)

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

which is operationally true in CIA, does not yield bisimilar π-calculus encodings; 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 $[\![A]\!]$ and $[\![B]\!]$ be bisimilar, we demand bisimilarity between $\prod \mathbf{Reg}_i[\sigma(x_i)] \mid [\![A]\!]$ and $\prod \mathbf{Reg}_i[\sigma(x_i)] \mid [\![B]\!]$ (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, both program and observer have unlimited access to registers.) To ensure that—apart from input and output—communication between

program and observer is only possible via these registers, we use a type system distinguishing between the capabilities of using a channel in input and output (I/O types, cf. [PS93,BS98]). So, 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 $\mathbf{put}_\iota$. The corresponding equivalence on π-calculus processes, for which soundness theorems similar to Theorems 1 and 2 hold, is closer to the observational congruence in CIA than the ordinary bisimilarity; it allows us to prove (1), as well as, e.g.,

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

Again, this equality is valid in CIA but not in the π-calculus applying its ordinary bisimilarity.
Yet full abstraction is not gained by introducing I/O types. Consider the following example, where P is a free identifier:

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

This example hinges on the unlimited access the observer has on fn_ι, in the π-calculus, once ι has been exported by calling P: 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. Yet, the access the observer has gained on ι at the time P was called, does not cease with the termination of the procedure (recall that in the π-calculus encoding, P is a free identifier). Hence, the observer can write on ι even after having signalled the termination of P. 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 refined typing would be necessary, which allows one to express linearity (the observer could use certain names only once) and sequentiality (the observer could use a given name only as long he/she does not use another given name) constraints on the use of names. Such a type system could 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 would allow us to validate equivalences like

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

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]; bisimilarity-based equivalences for them, as well as related algebraic properties, can be given by developing those for I/O types. However, even this further type refinement might not yield 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 variation of 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 are 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. The degree of parallelism in the presence of an active (i.e., currently running) **await** statement can be increased by, e.g., a simultaneous execution of phrases which do not access variables affected by the **await** statement. This can be modelled, in the SOS semantics, by locks carrying along information about the concerned variables; in the π-calculus semantics, multiple locks can be introduced. 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 as closed only such programs that do not possess open identifiers *nor* variables, using explicit input and output constructs An alternative approach is to provide the observer with direct access to the global variables:

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

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. 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 proofs conducted in these theories, with bisimulation-based proofs carried out in the π-calculus might clarify the relationship between these two notions.

Acknowledgements
We 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.

[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. In *thirteen LICS Conf.* IEEE Computer Society Press, 1998.

[FHJ95] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. Technical report, 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 *11th LICS*. IEEE Computer Society Press, 1996.

[Hon96] K. Honda. Composing processes. In *Proc. 23rd 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. IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98)*. North-Holland, 1998.

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

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report 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. Two volumes, also 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 *Proc. 15th POPL*, 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. Technical report, University of Stanford, 1990.

[MT90b] I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. Technical report, 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. Two 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, 1981. Also appeared in vol. 1 of [OT97].

[San95a] D. Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA-Sophia Antipolis, 1995. 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.

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash B : \mathbf{exp}[bool] \quad \Gamma \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{await}\, B\, \mathbf{then}\, C : \mathbf{comm}}\ C \text{ seq. comp. of ass., cond.}$$

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

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

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

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

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

Table 1: Syntax and typing of CIA

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

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

$$
\begin{aligned}
[\![!V]\!]^\ell_p &\stackrel{\text{def}}{=} (\nu\, q)([\![V]\!]^\ell_q \mid q(gt, pt).\overline{\ell}.gt(x).(\ell.0 \mid \overline{p}\langle x\rangle.0)) \\
[\![\mathbf{output}\, E]\!]^\ell_p &\stackrel{\text{def}}{=} (\nu\, q)([\![E]\!]^\ell_q \mid q(x).\overline{\ell}.\overline{\mathbf{out}}\langle x\rangle.(\ell.0 \mid \overline{p}.0)) \\
[\![\mathbf{input}\, \iota]\!]^\ell_p &\stackrel{\text{def}}{=} \overline{\ell}.\mathbf{in}(x).(\ell.0 \mid \overline{\ell}.\overline{\mathbf{put}}_\iota\langle x\rangle.(\ell.0 \mid \overline{p}.0)) \\
[\![V := E]\!]^\ell_p &\stackrel{\text{def}}{=} (\nu\, q, r)([\![V]\!]^\ell_q \mid [\![E]\!]^\ell_r \mid q(gt, pt).r(x).\overline{\ell}.\overline{pt}\langle x\rangle.(\ell.0 \mid \overline{p}.0)) \\
[\![\mathbf{await}\, B\, \mathbf{then}\, C]\!]^\ell_p &\stackrel{\text{def}}{=} (\nu\, a)(!a.(\nu\, q)(\overline{\ell}.[\![B]\!]_q \mid q(x).([x = \mathrm{tt}]\,(\nu\, r)([\![C]\!]_r \mid r.(\ell.0 \mid \overline{p}.0)) \mid \\
&\qquad\qquad [x = \mathrm{ff}]\,(\ell.0 \mid \overline{a}.0))) \mid \overline{a}.0)
\end{aligned}
$$

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