Applying π:
Towards a Basis for Concurrent Imperative Programming

Martin Odersky

Universität Karlsruhe
76128 Karlsruhe, Germany
odersky@ira.uka.de

Proc. ACM SIGPLAN Workshop on State in Programming Languages, January 1995

Abstract

We study an extension of asynchronous π-calculus where names can be returned from processes. We show that with this simple extension an extensive range of functional, state-based and control-based programming constructs can be expressed by macro expansions, similar to Church-encodings in lambda calculus.

1 Introduction

Most programming languages in use today have some way to express concurrent execution of processes – either in the language itself (e.g. Ada [20], Modula-3 [5], Facile [7], CML [23]) or by means of a library (e.g. Modula-2's Process module [28], C++'s thread library [25]). This paper proposes a formal basis for reasoning about such languages.

Traditionally, formal foundations for languages with concurrency constructs come in one of two styles. Most commonly, one combines a semantic description for the sequential base language with another one for the concurrency primitives. For instance, semantic descriptions of Facile [7] or CML [2] define a structured operational semantics for the base language as a special case of a larger labeled transition system that also models the concurrent aspects of the language. This style of description has the advantage that a semantics of the sequential part of the language can be obtained by subsetting. However, the resulting formal systems tend to be large.

Alternatively, one can use a standard process calculus such as CCS [13] or π-calculus [16] to reason about both the base language and the concurrency primitives. An example of this approach is the PICT programming language [21] that was designed with asynchronous π-calculus [4] as a basis. PICT stays fairly close to the underlying calculus and consequently does not fully support sequential programming constructs such as functions or sequential composition. Representing these constructs in traditional process calculi requires a *global* encoding, not unlike a conversion to continuation passing style in functional programming[1]. Examples of such encodings are found in work by Milner

[1]In fact, PICT takes an intermediate approach: There is a notation for function definition, which leaves the result channel implicit, but there is no corresponding notation for function calls, so that an explicit result channel argument always needs to be passed. In effect, this leads to function definitions in PICT being CPS-converted one at a time.

[15] for the case of functions and by Walker [27] or Jones [11] for the case of objects. If our aim is to reason about source programs such encodings are undesirable since they are all-or-nothing propositions: To reason about one part of a program one must encode everything.

We would prefer a relationship between programming language and foundation that is similar to the relationship between functional languages and λ-calculus. There, one is transformed to the other via *Church-encodings*, which are pure macro expansions. In this paper we show that a modest change to a standard process calculus is sufficient to capture both call-by-value functional programming and imperative programming via similar encodings. The *applied π calculus* augments asynchronous π calculus [4] (which is essentially equivalent to ν-calculus [10]) with the ability to return a name from a process. Together with standard name restriction this gives us a way to model anonymous values in the calculus. It turns out that this is all that is needed to encode essentially all sequential programming constructs in a concise and straightforward manner.

Interestingly, with just seven term formation rules and one reduction rule, applied π is more compact than calculi for sequential state-based languages [12, 6, 19]. This comparison is not completely fair, however, since the encoding into applied π gives us only an operational understanding of functional and imperative constructs. Much less is known at present about the observational properties of the encodings. In general, process contexts discriminate more terms than sequential contexts. Hence, source language constructs would need to be encapsulated in some way in order to preserve their observational properties, but such an encapsulation is not discussed here. Nevertheless, we believe that applied π can be useful for gaining semantic intuition about how familiar functional and state-based programming constructs should behave when extended to a concurrent setting.

Related work. We have already mentioned the work on PICT and the encodings by Milner, Walker, and Jones. Sangiorgi has argued that the higher-order π calculus improves on first-order π calculus as a foundation for functional programming [24]. In a sense, applied π's ability to return a name from a process is an alternative to higher-order processes, since λ-abstractions can be represented. Boudol's γ-calculus [3] tries to generalize both CCS and λ-calculus. Like in λ — and unlike in applied π — communicating agents are matched by position rather than just channel name.

Our process equivalence relation is based on Milner's and Sangiorgi's barbed bisimulation [17]. We adapt their definitions in a straightforward way to the asynchronous and applied case. Honda and Yoshida [9] have shown for an asynchronous calculus that barbed bisimulation has a tractable characterization that does not depend on a quantification over contexts.

The rest of this paper is organized as follows. Section 2 presents an operational semantics for applied π. Section 3 defines a notion of process equivalence for the calculus. Section 4 shows how functional programming constructs can be encoded in applied π. Section 5 does the same for imperative programming, giving encodings for the essential constructions of state and control. Section 6 presents an encoding of applied π in asynchronous π. Section 7 concludes.

2 The Core Calculus

Syntactic Domains

Variables	x, y, z			
Preterms	M, N, P	$=$	x	Variable
		$\mid$	$\nu x.M$	Restriction
		$\mid$	$x^? y.M$	Abstraction (Input)
		$\mid$	xM	Application (Output)
		$\mid$	$M \mid N$	Parallel Composition
		$\mid$	$!M$	Replication
		$\mid$	0	Identity

We build on asynchronous π-calculus [4], modulo some minor notational modifications that are introduced for making the treatment of function application smoother. There is one extension: Processes may evaluate to names, and an arbitrary term instead of a single name may appear as the argument of an application. Roughly speaking, an application xM is evaluated by evaluating M to some number of names which are all passed in parallel to the channel x.

Hence, applied π is a rather small variation of a standard process calculus. However, it can also be seen as a generalization of λ-calculus where the concept of a λ-abstraction is generalized in two ways. First, applied π's abstractions can be used only once, unless they are prefixed by a (!) replicator. This is similar to the role of abstractions in linear λ-calculus [1]. Second, an abstraction and its argument are matched by name rather than by position. Fresh local names are introduced by a restriction prefix νx (this has also been studied in the context of λ-calculus [22, 18]).

Notational Conventions. $\mathrm{fn}(M)$, the set of free names of a term M, is given by

$$
\begin{array}{lcllcl}
\mathrm{fn}(x) & = & \{x\} & \mathrm{fn}(\nu x.M) & = & \mathrm{fn}(M)\backslash\{x\} \\
\mathrm{fn}(x^? y.M) & = & \{x\} \cup (\mathrm{fn}(M)\backslash\{y\}) & \mathrm{fn}(xM) & = & \{x\} \cup \mathrm{fn}(M) \\
\mathrm{fn}(M \mid N) & = & \mathrm{fn}(M) \cup \mathrm{fn}(N) & \mathrm{fn}(!M) & = & \mathrm{fn}(M) \\
\mathrm{fn}(0) & = & \{\}. & & &
\end{array}
$$

$[x/y]M$ denotes substitution of x for all free occurrences of y in M. To avoid name capture problems in substitutions, we assume everywhere that the free and bound variables of a term and all its subterms are distinct. This can always be achieved by α-renaming (see below).

A note on precedence: Application binds tightest, followed by replication (!) and the binding prefixes νx and $x^? y$, followed by parallel composition (|). Application is left-associative and parallel composition is associative. Grouping can be changed by using parentheses.

We also use the words *channel* and *agent* interchangeably for name and term, respectively. We sometimes contract multiple input or restriction prefixes, using the abbreviations

$$
\begin{array}{lcl}
\nu x_1 \ldots x_n.M & \stackrel{\mathrm{def}}{=} & \nu x_1. \;\ldots\; \nu x_n.M \\
x^? y_1 \ldots y_n.M & \stackrel{\mathrm{def}}{=} & x^? y_1. \;\ldots\; x^? y_n.M \;\; .
\end{array}
$$

Equivalences

Terms are equivalence classes of preterms. We take *syntactic equivalence* ($\equiv$) to be the smallest congruence that satisfies the laws below.

1. Variables can be α-renamed.

$$
\begin{array}{llcll}
(\alpha_1) & \nu x.M & \equiv & \nu y.[y/x]M & (y \notin \mathrm{fn}(M)) \\
(\alpha_2) & z^? x.M & \equiv & z^? y.[y/x]M & (y \notin \mathrm{fn}(M))
\end{array}
$$

2. Replication composes arbitrarily many copies of a term in parallel.

$$
\begin{array}{llcl}
(\mathrm{Repl}) & !M & \equiv & M \mid !M
\end{array}
$$

3. Parallel composition is commutative and associative, with identity 0.

$$
\begin{array}{llcl}
(\mathrm{Comm}) & M \mid N & \equiv & N \mid M \\
(\mathrm{Assoc}) & (M \mid N) \mid P & \equiv & M \mid (N \mid P) \\
(\mathrm{Id}) & M \mid 0 & \equiv & M
\end{array}
$$

4. The scope of a restricted variable x can be extended over parallel composition and application, provided x is not captured. Restriction with an unused name has no effect.

$$
\begin{array}{llcll}
(\nu\text{-Par}) & M \mid \nu x.N & \equiv & \nu x.(M \mid N) & (x \notin \mathrm{fn}(M)) \\
(\nu\text{-Apply}) & y(\nu x.M) & \equiv & \nu x.yM & (x \neq y) \\
(\nu\text{-Garbage}) & \nu x.M & \equiv & M & (x \notin \mathrm{fn}(M))
\end{array}
$$

5. Application distributes over parallel composition. Application has no effect on abstraction arguments.

$$
\begin{array}{llcl}
(\mathrm{Dist}) & x(M \mid N) & \equiv & xM \mid xN \\
(\mathrm{Absorb}) & x(y^? z.M) & \equiv & y^? z.M
\end{array}
$$

Except for (ν-Apply), equalities (1)–(4) all have counterparts in π-calculus. Equalities (Dist) and (Absorb) are perhaps surprising at first. Essentially, they introduce a fundamental asymmetry between applications and abstractions. Abstractions are *volatile*, in that they can move freely into and out-of applications. By contrast, applications are *stationary*, they appear only a single fixed context. Note the similarity to first order functional programming, where abstractions correspond to function definitions (where the location of the definition does not matter) and applications correspond to function calls (where the point of call does matter). However, unlike in functional programming, an abstraction can be used only once if it is not replicated. We will see that this resource-consciousness is the essential ingredient that allows applied π to model side-effects in expressions.

Reduction

There is a single reduction rule.

$$
\begin{array}{llcl}
(\mathrm{Reaction}) & x^? y.M \mid xz & \rightarrow & [z/y]M
\end{array}
$$

Reduction is considered modulo syntactic equivalence. Reduction can be applied anywhere in a term except under an abstraction or a replication. That is, a binary reduction relation ($\rightarrow$) between terms is given by the axiom (Reaction) and the inference rule

$$\text{(Context)} \qquad \frac{M' \equiv M \qquad M \rightarrow N \qquad N \equiv N'}{E[M'] \rightarrow E[N']}$$

where E is an arbitrary *evaluation context* that can be generated by the grammar

$$E \;=\; [\,] \quad | \quad \nu x.E \quad | \quad xE \quad | \quad E|M \;.$$

Let $\twoheadrightarrow$ be the reflexive transitive closure of reduction.

3 A Process Equivalence

Our notion of equivalence of applied π terms is based on bisimulation. The central intuition of bisimulation is that an experiment which tests whether two processes are equivalent can be constructed from two basic actions: One can observe the interaction of a running process, and one can freeze a process in a given state and let in run repeatedly starting from this state. The latter distinguishes bisimulation from trace equivalence.

For processes whose operational semantics is defined by means of a reduction relation, a particularly simple form of bisimulation can be devised, which tests only the possibility of interacting on a channel, but disregards what is communicated over it. This relation is called *barbed bisimulation* [17]. For applied π, barbed bisimulation can be simplified further in that only the action of returning a name, but not input or output actions, can be observed. This is formalized in the following definitions.

Definition. A symmetric relation $\mathcal{R}$ on terms is *reduction-closed* iff $M\mathcal{R}N$ and $M \twoheadrightarrow M'$ implies the existence of a term N' such that $N \twoheadrightarrow N'$ and $M'\mathcal{R}N'$.

Definition. A term M *converges*, written $M \Downarrow$, if $M \twoheadrightarrow (x \mid N)$, or $M \twoheadrightarrow \nu x.(x \mid N)$ for some name x and term N.

Definition. A symmetric relation $\mathcal{R}$ between terms is a *(weak) barbed bisimulation* for applied π iff $\mathcal{R}$ is reduction-closed and $M\mathcal{R}N$ and $M \Downarrow$ implies $N \Downarrow$. $M \dot{\approx} N$ iff there is a bisimulation $\mathcal{R}$ such that $M\mathcal{R}N$.

$\dot{\approx}$ is not a congruence, for instance it is not preserved by parallel composition: $x^?y.0 \dot{\approx} x^?y.y$, but not $x^?y.0 \mid xz \dot{\approx} x^?y.y \mid xz$. We therefore define:

Definition. Let $\approx$ be the largest congruence such that $\approx \subseteq \dot{\approx}$.

In the following, whenever we say that two terms M, N are equivalent (written $M = N$) we mean that they are barbed congruent, *i.e.* $M \approx N$.

Proposition 3.1 (a) The following are bisimulation equivalences in applied π.

$$\begin{aligned} x(!M) &= !xM && (1)\\ !M &= !M \mid !M && (2)\\ \nu x.x^? y.M &= 0 && (3)\\ \nu x.(xy \mid x^? z.M) &= \nu x.[y/z]M && (4)\\ \nu x.(xy \mid !x^? z.M) &= \nu x.[y/z]M && (5) \end{aligned}$$

(b) If $x \notin \mathrm{fn}(M, N, P)$ then the following are also bisimulation equivalences.

$$\begin{aligned} \nu x.(xM \mid x^? y.zy) &= zM && (6)\\ \nu x.(xM \mid xN \mid !x^? y.P) &= \nu x.(xM \mid !x^? y.P) \mid \nu x.(xN \mid !x^? y.P) && (7)\\ \nu x.(!xM \mid !x^? y.P) &= !\nu x.(xM \mid !x^? y.P) && (8) \end{aligned}$$

Equation (1) is the analogue of the (Absorb) equivalence for replicated abstraction. Equation (2) says that parallel composition is idempotent on replicated terms. Equation (3) says that any term that reads from a freshly allocated variable is an identity for parallel composition. We also call such terms *inert*. Equations (4) and (5) say that reduction via a local variable is an equivalence. Equation (6) says that forwarding a term via a local variable is equivalent to sending the term directly to its final destination. Finally, equations (7) and (8) are factoring laws for a parallel composition or replication of output terms in a local computation.

4 Encoding Functions

We now encode functional programming constructs in applied π, using just macro expansions. We define an affine λ-abstraction $\lambda_1 x.M$, which can be applied at most once, and an unrestricted call-by-value abstraction $\lambda x.M$.

$$\begin{aligned} \lambda_1 x.M &\stackrel{\text{def}}{=} \nu f.(f \mid f^? x.M) && (f \text{ fresh})\\ \lambda x.M &\stackrel{\text{def}}{=} \nu f.(f \mid !f^? x.M) && (f \text{ fresh}) \end{aligned}$$

General function application can be simulated by using a local name for the function part of the application. Here we have a choice, whether function and argument part should be evaluated concurrently or in sequence. We start with sequential application, which is expressed by juxtaposition of function and argument and is encoded as follows:

$$M\,N \stackrel{\text{def}}{=} \nu x.(xM \mid !x^? f.fN) \qquad (x, f \text{ fresh})$$

Application of a channel x is (modulo $=$) a special case of sequential function application, as is seen by looking at the expanded form of xM, *i.e.* $\nu y.(yx \mid !y^? f.fM)$, where y and f are fresh names.

$$\begin{aligned} &\nu y.(yx \mid y^? f.fM)\\ &\quad = \nu y.xM && \text{by (5)}\\ &\quad \equiv xM && \text{by } (\nu\text{-GC}) \end{aligned}$$

This explains why we have chosen to use $x^?$ for abstraction and plain x for application, whereas in original π calculus plain x is an input prefix amd $\overline{x}$ is an output prefix.

Example 4.1

$$
\begin{array}{lll}
& (\lambda_1 x.x)\,(\lambda_1 y.y) & \\
\stackrel{\text{def}}{=} & \nu a.(a(\nu g.(g \mid g^? x.x)) \mid a^? g.gH) & \text{where } H \stackrel{\text{def}}{=} \nu h.(h \mid h^? y.y) \\
\equiv & \nu a.\nu g.(a(g \mid g^? x.x) \mid a^? g.gH) & \text{by } (\nu\text{-*}) \\
\equiv & \nu a.\nu g.((ag \mid g^? x.x) \mid a^? g.gH) & \text{by (Dist), (Absorb)} \\
\equiv & \nu a.\nu g.((ag \mid a^? g.gH) \mid g^? x.x) & \text{by (Assoc), (Comm)} \\
\rightarrow & \nu a.\nu g.(gH \mid g^? x.x) & \text{by reduction} \\
\equiv & \nu a.\nu g.(g(\nu h.(h \mid h^? y.y)) \mid g^? x.x) & \text{substituting the definition of } H \\
\equiv & \nu a.\nu g.\nu h.(g(h \mid h^? y.y) \mid g^? x.x) & \text{by various } (\nu) \text{ equivalences} \\
\equiv & \nu a.\nu g.\nu h.((gh \mid h^? y.y) \mid g^? x.x) & \text{by (Dist), (Absorb)} \\
\equiv & \nu a.\nu g.\nu h.(gh \mid g^? x.x \mid h^? y.y) & \text{by (Assoc), (Comm)} \\
\rightarrow & \nu a.\nu g.\nu h.(h \mid h^? y.y) & \text{reducing via } g \\
\equiv & \nu h.(h \mid h^? y.y) & \text{by } (\nu\text{-*}), \text{(GC)} \\
\stackrel{\text{def}}{=} & \lambda_1 y.y & \text{by sugaring}
\end{array}
$$

Proposition 4.2 The following are observational equivalences for applied π.

$$
\begin{array}{rcll}
(M \mid N)\,P & = & M\,P \mid N\,P & (9) \\
M\,(N \mid P) & = & M\,N \mid M\,P & (10) \\
(x^? y.M)\,N & = & x^? y.M & (11) \\
(!M)\,N & = & !(M\,N) & (12)
\end{array}
$$

The proofs are all simple equivalence chains. Two examples are: (9):

$$
\begin{array}{lll}
& (M \mid N)\,P & \\
\stackrel{\text{def}}{=} & \nu x.(x(M \mid N) \mid !x^? y.yP) & \text{desugaring the application} \\
\equiv & \nu x.(xM \mid xN \mid !x^? y.yP) & \text{by (Dist)} \\
= & \nu x.(xM \mid !x^? y.yP) \mid \nu x.(xN \mid !x^? y.yP) & \text{by (7)} \\
\stackrel{\text{def}}{=} & M\,P \mid N\,P & \text{resugaring}
\end{array}
$$

(12):

$$
\begin{array}{lll}
& (!M)\,N & \\
\stackrel{\text{def}}{=} & \nu x.(x(!M) \mid !x^? f.fN) & \text{desugaring the application} \\
= & \nu x.(!xM \mid !x^? f.fN) & \text{by (1)} \\
= & !\nu x.(xM \mid !x^? f.fN) & \text{by (8)} \\
\stackrel{\text{def}}{=} & !(M\,N) & \text{resugaring}
\end{array}
$$

Note that symmetric versions of (11) and (12) do not hold; *e.g.* in $M\,(x^? y.N)$, the abstraction becomes available only after M reduces to a name.

Parallel application ($\bullet$) imposes no sequencing constraints on the evaluation of a function and its argument. It is encoded as follows.

$$M \bullet N \quad = \quad \nu x.\nu y.(xM \mid yN \mid x^? f.y^? a.fa)$$

Local Definitions

Using lambda abstraction and application, we can define a let-construct $\mathsf{let}\ x = M\ \mathsf{in}\ N$ to be sugar for $(\lambda x.N)\,M$. Expanding this and simplifying yields:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{let}\ x = M\ \mathsf{in}\ N} \\ \stackrel{\text{def}}{=} & (\lambda x.N)\,M \\ \stackrel{\text{def}}{=} & \nu z.(z(\nu y.(y \mid !y^? x.N)) \mid !z^? w.wM) \\ \equiv & \nu y.(\nu z.(zy \mid !z^? w.wM) \mid !y^? x.N) \\ = & \nu y.(yM \mid !y^? x.N) \qquad\qquad\qquad \text{by (5)}\ . \end{array}$$

As in the λ-calculus, this gives us a non-recursive local definition where the variable x cannot appear in the body of its defining term, M. Recursive (function) definitions are also possible. They can be defined as follows:

$$\mathsf{letrec}\ f\,x = M\ \mathsf{in}\ N \quad \stackrel{\text{def}}{=} \quad \nu f.(N \mid !f^? x.M)\ .$$

This extends naturally to mutual recursion:

$$\begin{array}{l} \mathsf{letrec}\ f_1\,x_1 = M_1, \ldots, f_n\,x_n = M_n\ \mathsf{in}\ N \\ \quad \stackrel{\text{def}}{=} \quad \nu f_1 \ldots f_n.(N \mid !f_1^? x_1.M_1 \mid \ldots \mid !f_n^? x_n.M_n)\ . \end{array}$$

5 Encoding Imperative Programs

Sequential Composition

We can define the sequential composition of a value-producing term M and a term N by

$$M\ ;\ N \quad \stackrel{\text{def}}{=} \quad \nu x.(xM \mid x^? y.N) \qquad (x, y\ \text{fresh})\ .$$

This evaluates M until a value is produced, and then continues with N. The value produced by M is discarded. We use the convention that (;) has higher precedence than (|) but lower precedence than the unary operators.

If in $(M_1 \mid \ldots \mid M_n)\ ;\ P$ each M_i produces a value then P will be enabled as soon as one of the M_i produces its result. We can force a wait for all M_i's by defining a blocking parallel composition (|||) of independent subcomputations — this is essentially Hoare's interleave operator [8]. Interleave is expressed as follows:

$$M_1 \mathbin{|||} \ldots \mathbin{|||} M_n \quad \stackrel{\text{def}}{=} \quad \nu x_1\ \ldots\ x_n.(x_1 M_1 \mid \ldots \mid x_n M_n \mid x_1^? y_1\ \ldots\ x_n^? y_n.())$$

Here, the empty tuple () is a shorthand that stands for some arbitrary reserved name, whose identity is unimportant.

Dereferencing

One sometimes wants to use the result of a read operation as an argument in an application. Writing $x(y^?z.z)$ would not do, as this expression is equivalent to just $y^?z.z$. Instead, one can use $x(y\!\uparrow)$ where the read operator $(\uparrow)$ is given by:

$$x\!\uparrow \;\stackrel{\text{def}}{=}\; \nu a.(a() \mid x^?y.a^?z.y) \ .$$

Note the role of the *acknowledgment* channel a. Its purpose can be explained as follows. Clearly, to read from a channel x, we need a term of the form $x^?y.M$. The problem is that this term is volatile, and hence will reduce in the context of the corresponding output operation. But when writing $z(x\!\uparrow)$, for instance, we want the read value to be passed to z. This is accomplished by the pair of the output action $a()$ in the parallel composition and the input action $a^?z$ in the reader term. Figuratively an interaction via a "pulls back" the abstraction $a^?z.y$ into the context of the output term $a()$. A similar technique is used below in the modeling of mutable variables.

Mutable Variables

We now encode mutable variables with an allocation operation $\mathsf{newref}\ x$, where M computes the initial value of the allocated result variable, an assignment operation $r := x$, and a dereferencing operation $r\!\uparrow$.

$$\begin{array}{lcl}
\mathsf{newref}\ x & \stackrel{\text{def}}{=} & \nu r.(r \mid rx) \\
r := x & \stackrel{\text{def}}{=} & \nu a.(a() \mid r^?y.(rx \mid a^?z.x)) \\
r\!\uparrow & \stackrel{\text{def}}{=} & \nu a.(a() \mid r^?y.(ry \mid a^?z.y))
\end{array}$$

These constructs model a mutable variable by a name r that always has a pending output operation rx, where x denotes the current value of the variable. Consequently, assignment to a mutable variable involves reading out the old value before the new value is written. Likewise, dereferencing a mutable a variable involves reading out its value and then writing it back. Note that this makes assignment and read symmetric operations, which is reflected in the similarity of their encodings.

Initializations and assignments with structured terms are derived from these encodings as in the case of functions. That is,

$$\mathsf{newref}\ M \;\stackrel{\text{def}}{=}\; (\lambda x.\mathsf{newref}\ x)\ M \;=\; \nu y.(y^?x.\mathsf{newref}\ x \mid yM) \ ,$$

and, analogously,

$$r := M \;\stackrel{\text{def}}{=}\; (\lambda x.r := x)\ M \;=\; \nu y.(y^?x.r := x \mid yM) \ .$$

Multiple assignments can be expressed by interleaving.

$$r_1, \ldots, r_n := M_1, \ldots, M_n \;\stackrel{\text{def}}{=}\; r_1 := M_1 \,|||\, \ldots \,|||\, r_n := M_n \ .$$

Example 5.1 The following reduction shows that (;) enforces sequential execution of assignments. Consider the sequence of assignments $r := 1 \,;\, r := r\!\uparrow +1$ with initial value 0 of r:

$(r := 1 \,;\, r := r\!\uparrow +1) \mid r0$		
$\stackrel{\text{def}}{=}$	$(\nu a.(a() \mid r^?y.(r1 \mid a^?z.1)) \,;\, r := r\!\uparrow +1) \mid r0$	by desugaring the first assignment
$\stackrel{\text{def}}{=}$	$\nu s.(s(\nu a.(a() \mid r^?y.(r1 \mid a^?z.1))) \mid s^?d.r := r\!\uparrow +1) \mid r0$	by expanding the sequential composition
$\equiv$	$\nu s.\nu a.(s(a()) \mid r^?y.(r1 \mid a^?z.1) \mid s^?d.r := r\!\uparrow +1 \mid r0)$	by various equivalences
$\rightarrow$	$\nu s.\nu a.(s(a()) \mid r1 \mid a^?z.1 \mid s^?d.r := r\!\uparrow +1)$	reducing via r
$\equiv$	$\nu s.\nu a.(s(a() \mid a^?z.1) \mid r1 \mid s^?d.r := r\!\uparrow +1)$	by (Dist), (Absorb)
$\rightarrow$	$\nu s.\nu a.(s1 \mid r1 \mid s^?d.r := r\!\uparrow +1)$	reducing via a
$\rightarrow$	$\nu s.\nu a.(r1 \mid r := r\!\uparrow +1)$	reducing via s
$\equiv$	$r1 \mid r := r\!\uparrow +1$	by (GC)

Control

We conclude our overview of sequential programming constructs with an encoding of control operators *abort* and *call/cc* in applied π. To make a program M abortable, embed it in the context

$$\nu e.(M \mid e()) \ .$$

where e is some fresh name. Then *abort* is given by

$$\mathit{abort}\ x \ \stackrel{\text{def}}{=}\ e^?y.x \ .$$

Note the *reverse trigger*, $e()$, that gets replaced by the argument x of *abort* by creating the agent $e^?y.x$. Since abstractions are volatile, an occurrence of *abort* inside an application chain will thus react with the top-level trigger $e()$, thereby returning a result from the program. A similar trick is used in the encoding of *call/cc*:

$$\mathit{call/cc}\ f \ \stackrel{\text{def}}{=}\ \nu e.\nu k.(fk \mid !k^?x.e^?y.x \mid !e())$$

This passes a continuation k that captures the current context to the function f. Again, $e()$ acts as a reverse trigger that injects the argument of the continuation variable k into the context of the *call/cc*.

6 Encoding Applied π in Asynchronous π

Applied π has close relations to asynchronous π calculus. We now formalize this statement by giving an encoding of applied π in asynchronous π. We use a slight variation of Boudol's definition. In our version, π_{async}, terms are given by

$$M \ = \ \nu x.M \ \mid\ x^?y.M \ \mid\ xy \ \mid\ M|N \ \mid\ !M \ \mid\ 0 \ ,$$

modulo syntactic equivalences (α), (Repl), (Comm), (Assoc), (Id), (ν-Par), (ν-Garbage) and reduction is as in applied π.

As an equivalence theory for asynchronous π terms we also use barbed bisimulation, which now takes the following form.

Definition. A term $M \in \pi_{\text{async}}$ *outputs* on a channel x, written $M \Downarrow_x$, if there is a name y and a term N such that either $M \twoheadrightarrow xy \mid N$ or $M \twoheadrightarrow \nu y.(xy \mid N)$.

That is, we take as observables output actions, but not input actions. Based on this notion of observation, barbed bisimulation and barbed congruence are then defined as usual:

Definition. A symmetric relation $\mathcal{R}$ between terms in π_{async} is a (weak) asynchronous barbed bisimulation iff $\mathcal{R}$ is reduction-closed and $M\mathcal{R}N$ and $M \Downarrow_x$ implies $N \Downarrow_x$. $M \dot{\approx}_{\text{async}} N$ iff there is an asynchronous bisimulation $\mathcal{R}$ such that $M\mathcal{R}N$. let $\approx_{\text{async}}$ be the largest congruence contained in $\dot{\approx}$.

We now define a mapping $[\![\cdot]\!]$ that takes as arguments an applied π term M and a name r and yields a term in π_{async}. The name r represents a channel where the result of the translated term should be sent to. The translation is given by:

$$\begin{array}{lcl}
[\![x]\!]r & = & rx \\
[\![\nu x.M]\!]r & = & \nu x.[\![M]\!]r \\
[\![x^? y.M]\!]r & = & x^?(y,s).[\![M]\!]s \\
[\![xM]\!]r & = & \nu s.([\![M]\!]s \mid !\nu t.s^? y.(x(y,t) \mid !t^? z.rz)) \\
[\![M \mid N]\!]r & = & [\![M]\!]r \mid [\![N]\!]r \\
[\![!M]\!]r & = & ![\![M]\!]r \ .
\end{array}$$

We use for brevity polyadic inputs $x^?(y,z).M$ and outputs $x(y,z)$ which can be expanded with Honda and Tokoro's "zip-lock" technique[2] [10]:

$$\begin{array}{lcl}
x^?(y,z).M & \stackrel{\text{def}}{=} & x^? u.\nu v.(uv \mid v^? y.\nu w.(uw \mid w^? z.M)) \\
x(y,z) & \stackrel{\text{def}}{=} & \nu u.(xu \mid u^? v.(vy \mid u^? w.wz)) \ .
\end{array}$$

To show that this encoding is well-defined, have have to verify that it is insensitive to the preterm chosen to represent a term.

Proposition 6.1 Let r be a name. Let M, N be preterms such that $M \equiv N$. Then $[\![M]\!]r \approx [\![N]\!]r$.

Proof Sketch: Verify that the translations of all syntactic equivalence rules are barbed asynchronous bisimulations. $\Box$

The following lemma shows that forwarding of a result via an intermediary is indistinguishable from passing the result directly:

Lemma 6.2 Assume $s, t \notin \text{fn}(M)$. Then $[\![M]\!]r \;\approx\; \nu s.([\![M]\!]s \mid s^? x.rx)$.

[2] Note how parallel compositions in the input term correspond to input prefixes in the output term and vice versa.

We now show that the encoding preserves the reduction semantics of applied π, in the following sense:

Definition. Let $M, N \in \pi_{\text{async}}$. $M \rightarrow^{\approx}_{\text{async}} N$ iff there are terms $M' \approx_{\text{async}} M$ and $N' \approx_{\text{async}} N$ such that $M' \rightarrow_{\text{async}} N'$.

Proposition 6.3 Let M, N be terms in applied π and let $r \notin \text{fn}(M,N)$. If $M \rightarrow N$ then $[\![M]\!]r \rightarrow^{\approx}_{\text{async}} [\![N]\!]r$.

Proof: Assume $M \rightarrow N$ and $r \notin \text{fn}(M,N)$. Then we have:

$$\begin{array}{lll}
\multicolumn{2}{l}{[\![x^? z.M \mid xy]\!]r} & \\
\quad \equiv & x^?(z,s).[\![M]\!]s \mid \nu s.(sy \mid s^? z.\nu t.(x(z,t) \mid t^? u.ru)) & \\
\quad = & x^?(z,s).[\![M]\!]s \mid \nu t.(x(y,t) \mid t^? u.ru)) & \text{by local reduction} \\
\quad \rightarrow & \nu t.([y/z,t/s][\![M]\!]s \mid t^? u.ru) & \\
\quad \equiv & \nu t.([\![[y/z]M]\!]t \mid t^? u.ru) & \\
\quad = & [\![[y/z]M]\!]r & \text{by Lemma 6.2}
\end{array}$$

□

Proposition 6.4 Let M, N be terms in applied π. If, for all $r \notin \text{fn}(M,N)$, $[\![M]\!]r \approx_{\text{async}} [\![N]\!]r$ then $M \approx N$.

Proof: Assume $M \not\approx N$. Then there is a context C such that one of $C[M]$, $C[N]$ converges but the other does not. W.l.o.g. assume that $C[M] \Downarrow$, $C[N] \not\Downarrow$. Let a be a fresh name. Then, because of Proposition 6.3, $[\![C[M]]\!]a \Downarrow_a$ but $[\![C[N]]\!]a \not\Downarrow_a$. Since the encoding $[\![\cdot]\!]$ is compositional on terms, there is a context D in π_{async} and a name $r \notin \text{fn}(P)$ such that $[\![C[P]a]\!] \equiv D[[\![P]\!]r]$, for all terms P, names a. Hence, $D[[\![M]\!]r] \Downarrow_a$ but $D[[\![N]\!]r] \not\Downarrow_a$. It follows that $[\![M]\!]r \not\approx_{\text{async}} [\![N]\!]r$. □

Unfortunately, the other direction of Proposition 6.4 seems to be much harder to prove. Proposition 6.4 requires that reductions in applied π can be simulated by reductions in asynchonous π, which is guaranteed by Proposition 6.3. The reverse direction would require that every possible asynchronous reduction sequence that starts and ends in an encoded applied term simulates a reduction seuquence in applied π. This appears credible, but a formal proof is still missing. We therefore can only conjecture that $[\![\cdot]\!]$ takes equivalences in applied π to equivalences in $\pi_a sync$.

Conjecture Let M, N be terms in applied π. Let $r \notin \text{fn}(M,N)$. If $M \approx N$ then $[\![M]\!]r \approx_{\text{async}} [\![N]\!]r$.

7 Conclusion

We have presented a modification of asynchronous π calculus that allows us to model sequential programming constructs in a simple way, using just macro expansions. We believe that this proposal might evolve into a formal foundation for programming languages that can express concurrent execution of processes but at the same time retain their sequential programming heritage. However, more work needs to be done until this goal is achieved.

In particular, we would like to get process equivalence criteria that are more tractable than the barbed congruence we have used. Another open question concerns the relationship between the process equivalence theory of applied π and the corresponding theory of the pure asynchronous calculus. Finally, it should be possible to define a typed version of applied π by generalizing Milner's sorting approach for π calculus [14].

Acknowledgments I'd like to thank John Maraist, for reading and commenting on previous drafts of this work, and Benjamin Pierce, for his thorough review, which was a great help in improving the paper.

References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–129, January 1992.

[3] Gérard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In J. Díaz and F. Orejas, editors, *Proceedings TAPSOFT '1989*, pages 149–161, New York, March 1989. Springer-Verlag. Lecture Notes in Computer Science 351.

[4] Gérard Boudol. Asynchrony and the pi-calculus. Research Report 1702, INRIA, May 1992.

[5] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[6] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[7] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[8] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[9] Keiho Honda and Nobuko Yoshida. On reduction-based process semantics. In *Proc. 13th Conf. on Foundations of Softawre Technology and Theoretical Computer Science*, pages 373–387, December 1993.

[10] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proc. 5th European Conference on Object-Oriented Programming*, pages 133–147, July 1991. Springer LNCS 512.

[11] C.B. Jones. Process-algebraic foundations for an object-based design notation. Technical Report UMCS-93-10-1, University of Manchester, 1993.

[12] Ian Mason and Carolyn Talcott. Equivalence in functional languages with side effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[13] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[14] Robin Milner. The polyadic π-calculus: A tutorial. Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Edinburgh University, October 1991.

[15] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[16] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I + II. *Information and Computation*, 100:1–77, 1992.

[17] Robin Milner and D. Sangiorgi. Barbed bisimulation. In *Automata, Languages, and Programming, 19th International Colloquium*, 1992. Lecture Notes in Computer Science 623.

[18] Martin Odersky. A functional theory of local names. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 48–59, January 1994.

[19] Martin Odersky, Dan Rabin, and Paul Hudak. Call-by-name, assignment, and the lambda calculus. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 43–56, January 1993.

[20] United States Department of Defense. *The Programming Language Ada Reference Manual.* Springer-Verlag, 1980.

[21] Benjamin C. Pierce, Didier Rémy, and David N. Turner. A typed higher-order programming language based on the Pi-calculus. Draft report; available in the PICT distribution, July 1993.

[22] Andrew Pitts and Ian Stark. On the observable properties of higher order functions that dynamically create local names. In *SIPL '93 ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*, pages 31–45, June 1993. Yale University Research Report YALEU/DCS/RR-968.

[23] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[24] Davide Sangiorgi. An investigation into functions as processes. In *Proc. 9th International Conference on the Mathematical Foundation of Programming Semantics, New Orleans, Lousiana*, pages 143–159, April 1993.

[25] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[26] Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 192–214. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

[27] David Walker. π-calculus semantics of object-oriented programming languages. In Takayasu Ito and Albert R. Meyer, editors, *Proc. Theoretical Aspects of Computer Software*, pages 532–547. Springer-Verlag, September 1991. LNCS 526.

[28] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, 2nd edition, 1983.