Combinatory Formulations of Concurrent Languages

N. RAJA and R.K. SHYAMASUNDAR
Tata Institute of Fundamental Research

We design a system with six *Basic Combinators* and prove that it is powerful enough to embed the full asynchronous π-calculus, including replication. Our theory for constructing *Combinatory Versions* of concurrent languages is based on a method, used by Quine and Bernays, for the general elimination of variables in linguistic formalisms. Our combinators are designed to eliminate the requirement of *names* that are *bound* by an *input prefix*. They also eliminate the need for input prefix, output prefix, and the accompanying mechanism of *substitution*. We define a notion of *bisimulation* for the combinatory version and show that the combinatory version preserves the semantics of the original calculus. One of the distinctive features of the approach is that it can be used to rework several process algebras in order to derive equivalent combinatory versions.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda calculus and related systems*; I.1.3 [**Algebraic Manipulation**]: Languages and Systems—*substitution mechanisms*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*

General Terms: Theory, Languages

Additional Key Words and Phrases: Functional completeness, Quine-Bernays combinators

1. INTRODUCTION

The discipline of *combinatory logic* [Curry 1930; Schönfinkel 1924] began in the study of foundations of mathematics. It was proposed as an approach which could overcome the drawbacks of complex primitives, such as *substitution*, in formulations of mathematical logic [Russell and Whitehead 1912]. Much later, computer science provided impetus to research on *combinators* [Turner 1979a; 1979b]. The study of *combinators* has led to deep insights in the theory of sequential programming [Hindley and Seldin 1986] and has had a great influence in the implementation of functional programming languages [Peyton Jones 1987].

In the field of concurrency there has been very little research in the pursuit of combinators. A major reason for this could be the fact that during the initial period of research on foundational models of concurrency [Hoare 1985; Milner 1989], little attention was paid to the communication of data between processes. Value passing was modeled in an indirect way, by encoding data values in the names of ports and

A preliminary version of this paper appeared in Proceedings of the Asian Computing Science Conference, 1995.
Authors' address: Computer Science Group, Tata Institute of Fundamental Research, Mumbai 400 005, India; email: raja@tifrvax.tifr.res.in; shyam@tcs.tifr.res.in.

then by using infinite disjunctions of pure synchronization. The next generation of process algebras started focusing on the exchange of values. In the last decade many new process algebras that employ similar mechanisms for communication of data have been designed [Boudol 1989; Milner et al. 1992; Thomsen 1990]. Influenced by Milner's ideas [Milner 1989], in these process algebras, communication consists in sending and receiving a value synchronously through a shared port. Consider the following parallel ("|") composition in the π-calculus [Milner et al. 1992]:

$$x(y).P \mid \overline{x}z.Q \;\mapsto\; P\{y \leftarrow z\} \mid Q$$

In this expression $x(y).P$ and $\overline{x}z.Q$ are processes which communicate through the common port x. Process $\overline{x}z.Q$ sends the value z on port x and then activates Q. Process $x(y).P$ receives the value z on port x, *substitutes* z for y in P, and then triggers P. The expression $x(y)$ is called an *input prefix*, and it denotes "name x *binds* name y." Thus, once again we encounter the complex mechanism of *substitution*, with its attendant paraphernalia of binding mechanisms and bound entities.

At first sight, the problem of eliminating substitution in process algebras appears to be simple. In π-calculus, the values we substitute are always *names*, rather than *processes* (however it is vice versa for certain other process algebras like CHOCS [Thomsen 1990]). So, eliminating bound names seems to be easy. However, in comparison with λ-calculus [Barendregt 1984], the problem of eliminating *substitution* is much more difficult in the setting of concurrent processes. Let us look at some of the reasons for these difficulties. The process calculi for concurrency are syntactically very different from the λ-calculus. Most such calculi do not possess the operator-to-operand kind of applicative structure found in the λ-calculus. Hence, the flow of information is not just confined to syntactically adjacent terms. λ-calculus is a single-sorted theory (everything is a term), but most concurrent calculi are inherently two-sorted, the two sorts being *processes* and *channels.* There is only one "abstractor" (λ) in λ-calculus, while there are innumerable distinct "abstractors" (infinitely many distinct names) in process algebras. Each of the *names* is distinct as an "abstractor" because the identity of an "abstractor" in a term should be accessible even after eliminating the corresponding bound name. This information is crucial in determining the subsequent reductions of the term under consideration. Apart from the mechanism of abstraction, there is only one other "operation" in λ-calculus (namely application), while in process algebras there is a rich set of other "operations" (viz., $|$, $+$, ν, and $!$ in the π-calculus). Further, there is a plethora of process algebras (which handle *value passing*), each of them as useful and powerful as the other.

The aim of this article is to design a system of *combinators*, which completely eliminates the need for substitution in process algebras. The *combinators* should explicitly handle all the operational details of the flow of data across processes, without relying on a metalevel operation such as substitution. Such a combinatory reformulation of any process algebra, would not only provide an alternative semantics in terms of *combinators*, but would also prove to be a valuable tool in the implementation of the process algebra.

In this article, we shall work in the setting of the asynchronous π-calculus [Boudol 1992; Honda and Tokoro 1991] with replication. The *combinators* we design arise

from a technique that was formulated independently by Bernays [1959] and Quine [1959] for the general elimination of variables in linguistic formalisms. We design a system of six *Basic Combinators* and prove that it is powerful enough to embed the asynchronous π-calculus (including process replication). We define a notion of bisimulation for the combinatory version and show that the combinatory version preserves the semantics of the original calculus. Further, the same approach can be used to rework several process algebras [Boudol 1989; Milner 1991; Milner et al. 1992; Thomsen 1990] in order to derive equivalent combinatory versions.

The organization of this article is as follows: Section 2 briefly reviews some background material; Section 3 motivates and introduces the combinatory version of the asynchronous π-calculus; Section 4 presents a formal definition of the combinatory version; Section 5 provides a formal embedding of the asynchronous π-calculus into the combinators and shows that the combinatory version preserves the semantics of the original calculus; Section 6 examines related work; and finally Section 7 concludes the article with directions for further research.

2. BACKGROUND

2.1 Quine's Technique in Logic

As mentioned earlier, the combinators that we design arise from a technique that was formulated independently by Bernays [1959] and Quine [1959]. However, there are slight differences in the methods proposed by each of them. We now give a brief introduction to the method advocated by Quine [1960].

Consider a first-order predicate logic, with the following:

Alphabet: $x,\ y,\ z \ldots$ variables
P predicate symbol of arity n
A quantifier.
Terms and Formulas: The usual standard definitions.

In order to eliminate variables and quantifiers from every formula of the above theory, Quine introduced the *combinators inv* (*Minor Inversion*), *Inv* (*Major Inversion*), *Ref* (*Reflection*), and *Der* (*Derelativization*). These *combinators* operate iteratively on the predicate P, to yield new predicates which are in turn defined over the original universe only. The *combinators* are defined as

$(inv\ P)\ x_1\ \ldots\ x_{n-2}\ x_{n-1}\ x_n$ iff $P\ x_1\ \ldots\ x_{n-2}\ x_n\ x_{n-1}$;
$(Inv\ P)\ x_1\ \ldots\ x_{n-1}\ x_n$ iff $P\ x_n\ x_1\ \ldots\ x_{n-1}$;
$(Ref\ P)\ x_1\ \ldots\ x_{n-1}$ iff $P\ x_1\ \ldots\ x_{n-1}\ x_{n-1}$; and
$(Der\ P)\ x_1\ \ldots\ x_{n-1}$ iff $Ax_n\ P\ x_1\ \ldots\ x_{n-1}\ x_n$, where x_n is a new variable.

For illustration, consider the formula $Ax\ Pxyxz$ with arity$(P) = 4$. In order to rid quantifier A and the bound variable x from this formula, we have to use the combinators. The formula $Ax\ Pxyxz$ can be transformed to $Ax\ (inv\ P)\ xyzx$. The formula obtained can be further transformed into $Ax\ (Ref\ Inv\ inv\ P)\ yzx$. Another transformation leads to $(Der\ Ref\ Inv\ inv\ P)\ yz$, which has neither a quantifier, nor a bound variable.

The work reported in Raja and Shyamasundar [1995b] extends the above technique to the setting of higher-order languages such as λ-calculus.

2.2 A Brief Review of the Asynchronous π-Calculus

The presentation in this section closely follows that of Boudol [1992] and Milner [1991]. Asynchronous π-Calculus (API) [Boudol 1992; Honda and Tokoro 1991] is a model of concurrent computation that supports process mobility by naming and passing channels. It consciously forbids the transmission of processes as messages. API is a two-sorted theory consisting of the sorts *names* (*channels, ports*) and *processes* (*agents*).

Definition 2.2.1 *(Names and Processes).* *Names* $(x, y, z, \ldots \in \mathcal{N})$ are atomic entities while *Processes* $(P, Q, \ldots \in \mathcal{P})$ have the following structure:

$$P \ ::= \ 0 \ \mid \ \overline{x}y \ \mid \ x(y).P \ \mid \ (P|Q) \ \mid \ !P \ \mid \ (\nu x)P$$

The term 0 represents an *inactive* process, which cannot perform any action. (We shall omit the trailing ".0" from process terms.) The construct $x(y)$ (called an *input prefix*) represents an atomic action, where name x *binds* name y. The process term $x(y).P$ waits for a name to be transmitted along channel x, *substitutes* the *received* name for all free occurrences of y in P, and then triggers P. The construct $\overline{x}y$ (representing an atomic action) *outputs* the name y along x, but *does not bind* name y. The form $P|Q$ denotes that P and Q are *concurrently* active, independent, and can *communicate*. Operator "!" is called *replication*, and $!P$ denotes $P|!P$. Finally, $(\nu x)P$ *restricts* the use of name x to P. Apart from input prefix, "ν" is another mechanism for *binding* names within a process term in API. Operator "ν" may also be thought of as *creating* new channels.

We define the operational semantics of API using the chemical abstract machine formalism (CHAM) [Berry and Boudol 1992]. The CHAM formulation greatly simplifies the reduction rules for systems dealing with concurrently active entities [Milner 1992b].

A CHAM is specified by defining its *molecules* (terms of algebras, with specific operations), *solutions* (finite multisets of *molecules*, which represent the state of the system), and *transformation rules* (which dictate the evolution of *solutions*).

Notation 2.2.2. $m_1, m_2, \ldots$ range over *molecules*; $\{|.|\}$ is a *membrane* operator where $\{|m_1, \ldots, m_k|\}$ represents a finite multiset of *molecules*; $S_1, S_2, \ldots$ range over *solutions (*finite multisets of *molecules)*; $S_1 \uplus S_2$ denotes the union of two multisets; $S_1 \Rightarrow S_2$ denotes the transformation of solutions; the context notation $m[S]$ denotes a molecule containing a submolecule S which is a solution.

Definition 2.2.3 *(Molecules and Solutions for API).*

(1) Any process term P of the calculus is a molecule

(2) Any solution $\mathcal{S} = \{|m_1, \ldots, m_k|\}$ is a molecule

(3) If m is a molecule and x is a name, then $(\nu x)m$ is a molecule.

The behavior of a CHAM consists in state changes, described by transformations of solutions. The transformation rules are divided into two categories — *general rules* applicable to all CHAMs and *specific rules* that define the given CHAM.

Definition 2.2.4 *(General Transformation Laws).*

$$\frac{S_1 \Rightarrow S_2}{S_1 \uplus S \Rightarrow S_2 \uplus S} \qquad \text{(Chemical Law)}$$

$$\frac{S_1 \Rightarrow S_2}{\{|m[S_1]|\} \Rightarrow \{|m[S_2]|\}} \qquad \text{(Membrane Law)}$$

The specific transformation rules of the CHAM for API may be classified into three kinds: *heating rules, cooling rules,* and *reaction rules.* The heating and cooling rules perform structural manipulations, while reaction rules really change the information in the solution in an irreversible way.

Notation 2.2.5. The *heating rules* are denoted by $\rightharpoonup$; *cooling rules* by $\leftharpoondown$; and *reaction rules* by $\mapsto$. The transformation $\Rightarrow$ is the union of these three relations; and $\overset{*}{\rightleftharpoons}$ denotes the reflexive and transitive closure of $\rightleftharpoons$.

Definition 2.2.6 *(Specific Transformation Rules for API).*

$$\begin{array}{rcll}
x(y).P\ ,\ \overline{x}z & \mapsto & P[y \leftarrow z] & \text{(Reaction)} \\
(P|Q) & \rightleftharpoons & P, Q & \text{(Parallel)} \\
!P & \rightleftharpoons & P, !P & \text{(Replication)} \\
(\nu x)P & \rightleftharpoons & (\nu x)\{|P|\} & \text{(Scoping Membrane)} \\
(((\nu x)P)|Q) & \rightleftharpoons & (\nu x)(P|Q) \quad (x \text{ not free in } Q) & \text{(Scope Migration)} \\
(\nu x)P & \rightleftharpoons & (\nu y)P[x \leftarrow y] \quad (y \text{ not free in } P) & \text{(Name Conversion)} \\
(\nu x)P & \rightleftharpoons & P \quad (x \text{ not free in } P) & \text{(Scope Extinction)} \\
(\nu x)(\nu y)P & \rightleftharpoons & (\nu y)(\nu x)P & \text{(Scope Exchange)}
\end{array}$$

Definition 2.2.7 *(Reduction Relation for API).* We say that

(1) Q and R are *structurally congruent* whenever $Q \overset{*}{\rightleftharpoons} R$

(2) the term Q reduces to R, in notation $Q \longrightarrow R$, whenever $Q \overset{*}{\rightleftharpoons} Q'$, $Q' \mapsto R'$, and $R' \overset{*}{\rightleftharpoons} R$.

Following Milner and Sangiorgi [Milner 1991; Milner and Sangiorgi 1992], we define the notions of bisimulation and congruence for API.

Definition 2.2.8 *(Unguarded Process).* A process Q occurs *unguarded* in P if it has some occurrence in P which is not under a prefix.

Definition 2.2.9 *(Observable Action in API).* A process P can perform an *observable action*, written $P\downarrow$, if for some pair of names x, y either the input construct $x(y).Q$ or the output construct $\overline{x}y$ occurs unguarded in P with x unrestricted.

Definition 2.2.10 *(Barbed Bisimulation for API).* A relation R over processes is a *barbed simulation* if $P\ R\ Q$ implies

(1) if $P \longrightarrow P'$ then $Q \longrightarrow Q'$ and $P'\ R\ Q'$ and

(2) $P\downarrow$ implies $Q\downarrow$.

Relation R is a *barbed bisimulation* if R and R^{-1} are *barbed simulations.* Processes P and Q are *barbed-bisimilar*, if $P\ R\ Q$ is true for some *barbed bisimulation* R.

Definition 2.2.11. A *process context* $\mathcal{C}[\ ]$ is a process term with a single hole, such that placing a process in the hole yields a well-formed process.

Definition 2.2.12 *(Barbed Congruence for API).* Processes P and Q are said to be *barbed-congruent*, written $P \sim Q$, if for each process context $\mathcal{C}[\ ]$ it holds that $\mathcal{C}[P]$ is barbed-bisimilar to $\mathcal{C}[Q]$.

3. COMBINATORY VERSION OF THE ASYNCHRONOUS π-CALCULUS

In this section, we introduce a combinatory formulation for the asynchronous π-calculus, through a series of illustrative examples. Section 3.1 shows how the *Basic Combinators* and *Transformation Rules* arise when we try to eliminate input prefix. In Section 3.2 we demonstrate that the same combinators suffice to handle more complex situations. We shall use acronyms API and CAPI to refer to the asynchronous π-calculus and its combinatory version respectively.

3.1 A Gentle Initiation

API has two distinct sorts of entities: *names* (*ports, channels*) and *processes* (*agents*). In CAPI there is one more distinct sort called *combinators*. *Names* in API are atomic entities devoid of any structure, and there are two forms of atomic *actions* that a process can perform: sending or receiving a *name*. However *processes* cannot be sent or received. All these properties continue to hold in CAPI.

Combinator "S." We introduce the *Basic Combinator* "S" to represent the action of *sending* (this is not the **S** of classical combinatory logic). Before a *send* action can take place, the knowledge of two names is required — on *which* name to send and *what* name to send. The API process $\overline{x}z$ is represented in CAPI as the process — Sxz — which denotes "on the channel x send the name z." The combinator S needs to be supplied with two names as arguments in order to construct a process term from it. Subsequently we need the ability to determine the *number of names* that a given combinator needs before it can be turned into a process. Hence we define a function called "Valency" to represent this information, e.g., Valency$(S) = 2$.

Combinator "R." The other basic action in API is that of *receiving* a name. The API process $x(y)$ denotes "receive some *name* — call it y — on channel x." Here the name y is said to be bound by x. In other words, y is being used as a *dummy name*, which will get replaced by the *actual name* that is received along x. We introduce the next *Basic Combinator* "R" to represent the action of *receiving*. Analogous to S, combinator R needs two arguments before it can represent a process, i.e., Valency$(R) = 2$. Let us supply the arguments one at a time. Thus Rx denotes "*receive* on the channel x." Next let us examine the interpretation of Rxy. Since bound names have no place in the combinatory version, an expression like Rxy where y is free would take on the following meaning: on the channel x the name y *was received*. While we want to specify future actions in our calculus, we seem to be able only to report history. However, combinator "*Deg*" described below saves us from this piquant situation.

Combinator "Deg." We now introduce the third *Basic Combinator* "*Deg*" (*Degeneralization*) with the requirement that Valency$(Deg) = -1$. The *negative* Valency makes it apparent that *Deg cannot directly operate on names*. Instead *Deg* operates on the combinator R to give a new combinator "*Deg R*" which in turn is capable of operating on names. We extend the definition of the Valency function to include sequences of *Basic Combinators* in its domain. For example, Valency$((Deg\ R)) =$ Valency$(Deg) +$ Valency$(R) = -1 + 2 = 1$. Thus the expres-

sion $(Deg\ R\ x)$ now represents a process which may receive any name whatsoever on channel x. We shall see in the following subsection that after receiving the name y the process $(Deg\ R\ x)$ gets transformed to Rxy, which is a historical record of a receive action that has already taken place. We hasten to add that though we give a meaning to the Combinator $(Deg\ R)$, we shall never assign a meaning to the combinator $(R\ Deg)$, thus ensuring well-foundedness in the interpretation of the combinators. Further, note that the domain of the newly constructed $(Deg\ R)$ is no different from that of R; both work on names only.

Transformations "Reaction" and "Cleanup." The *Reaction* rule captures the act of communication.

$$Deg\ C\ \vec{y}x\ ,\ S\ xz \quad \mapsto \quad C\ \vec{y}xz \qquad \text{(Reaction)}$$

where C is any string of *Combinators*, and $\vec{y}$ is any string of *names*.

The *Cleanup* rule eliminates *inert* molecules which precipitate from the solution.

$$R\ x_1x_2 \quad \rightharpoonup \qquad \text{(Cleanup)}$$

Let us compose $(DegR\ x)$ in parallel with Sxz and observe the result.

$$\begin{array}{rcll} (DegR\ x)\mid Sxz & \rightleftharpoons & (DegR\ x), Sxz & \text{(Parallel)} \\ (DegR\ x), Sxz & \mapsto & R\ xz & \text{(Reaction)} \\ R\ xz & \rightharpoonup & & \text{(Cleanup)} \end{array}$$

The molecule $(Deg\ R\ x)$ can accept a message on x, while the molecule Sxz can send the message z on x. Hence, *Reaction* occurs. The *Reaction* rule does not involve bound names or binding mechanisms or any *substitution*. The *Cleanup* rule eliminates the *Precipitate* Rxy, as it cannot perform any more actions.

Discussion. The above example may be dismissed as a trivial case, wherein we could get away without using bound variables because we never needed to refer to them anyway. Before we go on to consider more complicated instances, there are certain features of the Combinatory Version that we wish to point out. First, we chose to introduce a new combinator Deg instead of redefining Valency$(R) = 1$, because the new combinator Deg helps us to uniformly tackle similar situations which will crop up later. Second, observe that the structure of the process terms we have considered so far have the form <Sequence of Combinators><String of Names>. The combinators and the names that constitute a process term are clearly separated as distinct strings. Third, Valency(<Sequence of Combinators>) = Length (<String of Names>) where the Valency of a sequence of Combinators is the sum of the Valencies of the individual Combinators that constitute the sequence. For example in the process term $(Deg\ R\ x)$, Valency$(Deg\ R)$ = Valency(Deg) + Valency(R) $= -1 + 2 = 1 = |x|$.

Transformation "p-Bonding."

$$(C_1\ \vec{x}\mid C_2\ \vec{y}) \quad \rightleftharpoons \quad (C_1|C_2)\ \vec{x}\vec{y} \qquad (\mathrm{p}-\mathrm{Bonding})$$

where Valency$(C_1) = |\vec{x}|$ and Valency$(C_2) = |\vec{y}|$.

Consider the API composition $\overline{x}z \mid u(y)$, which in CAPI is $Sxz \mid (DegRu)$. *Reaction* cannot occur because the channels used for sending and receiving are different in the two processes. However, the two process terms may evolve reversibly as

$$Sxz \mid (DegR\ u) \rightleftharpoons (S|(DegR))\ xzu$$

to form a *molecular bond*. Valency information is required when the process molecules have to *split* the molecular bond, so as to take part in future reactions. At the time of splitting the molecular bond, the combined argument string of names has to be separated at the appropriate position:

$$(S|(DegR))\ xzu \rightleftharpoons Sxz \mid (DegR\ u)$$

where Valency $(S) = 2 = |xz|$, and Valency$(DegR) = -1 + 2 = |u|$. The *p-Bonding* transformation is not a superfluous rule that we can do away with. It will turn out to be crucial when we want to encode API terms which have nested parallelism in their structure.

Discussion. In λ-calculus there is only one "abstractor," namely λ. The symbol λ by itself has no meaning. On the other hand in API there are innumerably many "abstractors," since each of the infinitely many names can be used as an abstraction operator. Thus even after eliminating a *bound name*, the identity of the "corresponding abstractor" (*binding name*) should be retrievable. The best way to retain such information is to encode it in the structure of the term itself. Therefore we shall require immediately after the elimination of a bound name from a process term that the corresponding *binding name* should be the last element in the string of names. In other words, just before the final step of eliminating a bound name from a process term, the corresponding *binding name* should figure as the penultimate element in the string of names.

Combinator "Ref." Consider the API process $x(y).\overline{y}y$. Here the dummy name y seems to be unavoidable, since the specification of future actions seems to be affected crucially by the name which will be received. However, for a moment, let us assume that y is a free name that has already been received on the channel x. The situation prevailing in such a case can be represented in CAPI as $Rxy,\ Syy$ where Rxy is a record of the past action. We now introduce the *"History"* Transformation Rule which *precipitates* records of past actions.

$$(R\ C)\ x_1x_2x_3\ldots x_n \quad \rightharpoonup \quad R\ x_1x_2,\ C\ x_3\ldots x_n \qquad \text{(History)}$$

Note that we obtain $(Rxy,\ Syy)$ from $(RS\ xyyy)$ by the *History* transformation rule. Now we introduce the fourth *Basic Combinator "Ref"* with Valency$(Ref) = -1$ and the *"Reflection"* Transformation Rule.

$$\text{Ref}\ C\ x_1\ldots x_n \quad \rightharpoonup \quad C\ x_1\ldots x_nx_n \qquad \text{(Reflection)}$$

The above rule helps in getting the required CAPI term $(DegRefRefRS\ x)$, with the "abstractor" x in the appropriate position.

Combinators "Inv" and "inv." The *Basic Combinators*, "*Inv*" and "*inv*," follow the "*Major Inversion*" and "*Minor Inversion*" transformation rules respectively.

$$\begin{array}{rcll} Inv\ C\ x_1 \ldots x_{n-1} x_n & \rightharpoonup & C\ x_n x_1 \ldots x_{n-1} & \text{(Major Inversion)} \\ inv\ C\ x_1 \ldots x_{n-2} x_{n-1} x_n & \rightharpoonup & C\ x_1 \ldots x_{n-2} x_n x_{n-1} & \text{(Minor Inversion)} \end{array}$$

Between themselves, these two combinators can permute any element of the argument string to an arbitrary position in the string.

3.2 Representing More Complex Terms

Processes with "Par(|)." In order to encode the API term $u(x).(\overline{x}v \mid \overline{x}w)$, we begin by encoding the subterm containing "|". The subterm can be mapped to $Sxv \mid Sxw$. At this stage the "*p-Bonding*" transformation rule is essential to derive

$$(S|S)\ xvxw \rightleftharpoons Sxv \mid Sxw.$$

We then proceed as in earlier examples and eliminate the bound name x to obtain the final encoding as $(Deg\ Ref\ Ref\ Inv\ Inv\ inv\ Inv\ inv\ R\ (S \mid S)\ vwu)$. From this encoding we notice that process constructors of API may occur along with the basic combinators in CAPI. But, the separation between *combinators* and *names* still persists. We extend Valency function to include the process constructors in its domain by defining Valency(|) = 0.

Input Prefix versus Restriction. There are two distinct binding mechanisms in API: *input prefix* (denoted by ".") and *restriction* (denoted by "ν"). The similarity ends in the fact that both mechanisms bind names in process terms. However, there are many substantial differences between the two bindings [Milner 1991; Milner et al. 1992]. We indicate these differences below.

Consider an API term $x(y).P$, where the input prefix $x(y)$ binds name y in process P. Name y is said to be *bound* because y can as well be replaced by any other name z (which is not free in P), without affecting the meaning of the process term. This fact is expressed by saying that $x(y).P$ is structurally congruent to the term $x(z).P\{x \leftarrow z\}$, where z is not free in P. The binding of names due to an input prefix is similar to the binding of variables by λ in the λ-calculus. During an interaction, the bound name y acts as a placeholder for any name which is received on channel x. The received name replaces the bound name y in the process term P. The received name, which is going to substitute y, can be any name whatsoever. In fact it may even already occur free in P. Also note that the replacement for y is obtained explicitly from another process term within the calculus itself. Our combinators are concerned with this receiving and substitution mechanism.

Now consider an API process term, $(\nu y)\ Q$, where name y is bound by the restriction operator. The restriction mechanism combines two distinct roles in one operator. First, it hides all interactions on the name y within Q, thus preventing external processes from interfering on communications along channel y. In effect, it declares a local name y, for use exclusively within Q. In this sense it is similar to the *let* construct used in functional languages and similar to new variable declarations in block-structured languages (e.g., Pascal). Thus name y can be replaced by any other name which is not free in the Q. This is expressed by saying that the term $(\nu y)\ Q$ is structurally congruent to $(\nu z)\ Q\{y \leftarrow z\}$, where z is not

free in Q. Second, the restriction operator has closer connections with *references* (mutable storage cells) of ML [Fiore et al. 1996; Stark 1996]. This can be seen from the fact that the restriction operator ensures that name y is distinct from all external names too [Milner 1991; Milner et al. 1992]. This is required because API allows local names to be communicated to external processes. A term of the form $(\nu y)(\overline{x}y.Q)$ can be viewed as simultaneously creating and transmitting a new name. Name y is at first local to Q and becomes active after the transmission. Thus, the operator ν may be thought of as a mechanism which creates globally unique channel names. In this role it also has an analog the *Actor* model of computation [Hewitt et al. 1973], which uses a purely local mechanism for generating globally unique *Actor addresses*. There is another important aspect to be noted. Though the bound name y in (νy) Q may be replaced by any other globally unique name z without affecting the meaning of (νy) Q, such globally unique names are not received from other process terms within the calculus. Unlike the input prefix, there is no mechanism to receive such names on a particular channel, and hence there is no accompanying mechanism of *dynamic* substitution.

Retaining "Restricted" Names. In this subsection and the next, we illustrate the translation of API process terms containing the restriction operator. In this subsection, we do not eliminate names bound by the ν operator, i.e., we retain the implicit capability of the ν operator, to generate globally unique names in situ. Consider the API term $(\nu x)(\overline{x}y \mid x(u).\overline{u}v)$. We first encode $\overline{x}y \mid x(u).\overline{u}v$ as $(S \mid (Deg\ Ref\ Inv\ Inv\ Inv\ R\ S\ ))\ xyvx$. The next step in the encoding process is obtained by the ν-*Bonding* Transformation Rule.

$$(\nu\ x)(C_1\ \vec{x} \mid C_2\ \vec{y}) \quad \rightleftharpoons \quad (\nu\ (C_1 \mid C_2))\ x\vec{x}\vec{y} \qquad (\nu - \text{Bonding})$$

The above rule finally yields $(\nu\ (S \mid (Deg\ Ref\ Inv\ Inv\ Inv\ R\ S))\ xxyvx)$. We also extend the definition of Valency to define Valency$(\nu) = 1$.

Eliminating "Restricted" Names. In this subsection, we do not require the implicit capability of "ν" to generate globally unique names. Instead we introduce a globally accessible parametric process $U(z)$, which captures the metalevel capability of generating unique channel names and internalizes it as a process term within the calculus itself. We regard the symbol ν as a special, globally accessible channel over which $U(z)$ transmits globally unique names. Any other process may use the channel ν only for receiving globally unique names. $U(z) = \overline{\nu}z.U(z')$, and it evolves as $\overline{\nu}z.\overline{\nu}z'.\overline{\nu}z'' \ldots$. It is composed in parallel with the entire system of processes. To encode the API term $(\nu x)(x(u)|\overline{x}y)$, we first rewrite it as $\nu(x).(x(u)|\overline{x}y)$. Note the change in notation — (νx) *has been rewritten as* $\nu(x)$. The transformed notation clearly indicates that the binding due to the restriction operator can be regarded similar to the binding caused by an input prefix, under the conditions formulated above. Thus, an additional reduction step has been introduced. The transformed term, $\nu(x).(x(u)|\overline{x}y)$, is simply an API process term with multiple input prefixes, and its CAPI translation is $(Deg\ Ref\ Inv\ Inv\ Inv\ R\ ((DegR) \mid S))\ y\nu$. Thus, there is no need for any additional combinators to be defined. Instead of using a global name generator, it is possible to specify a distributed name generator. Such a specification is presented in Bodei et al. [1996] which also gives an algebraic

characterization of the resulting scenario. The results of this subsection are not included in the formal definition of API that we present later in this article. This is in keeping with the aim of this article, which is to model the operational details of the flow of names across API processes, without making major changes to API otherwise.

Processes with Dynamic Replication. The efficacy of our combinators becomes apparent when we demonstrate that they can encode infinite behavior too. No additional combinators are required in order to get the combinatory representation of API processes with the ! operator. Consider the API process $!x(u).\overline{u}v$. Begin by encoding $x(u).\overline{u}v$ to get $(Deg\ Ref\ Inv\ Inv\ Inv\ R\ S\ vx)$. The CAPI translation of $!x(u).\overline{u}v$ is given by $!(Deg\ Ref\ Inv\ Inv\ Inv\ R\ S\ vx)$. The definition of Valency is extended by defining Valency(!) = 0.

4. FORMAL DEFINITION OF THE COMBINATORY CALCULUS

Definition 4.1 *(Basic Combinators).*

$$\text{Basic Combinators} = \{S, R, Deg, Ref, Inv, inv\}$$

Definition 4.2 *(Combinators).* The set of *Combinators*, **C**, consists of all finite strings of the elements of the set $\{S, R, Deg, Ref, Inv, inv, |, \nu, !\}$. We shall use $C, C', C'', C_1, C_2, \ldots$ to range over **C**.

Definition 4.3 *(Valency Function). Valency*: $\mathbf{C} \rightarrow \mathbf{Z}$ is given by

(1) $Valency(\Lambda) = 0$ (where Λ denotes an empty string)

(2) $Valency(S) = Valency(R) = 2$

(3) $Valency(Deg) = Valency(Ref) = -1$

(4) $Valency(Inv) = Valency(inv) = 0$

(5) $Valency(\nu) = 1$

(6) $Valency(|) = Valency(!) = 0$

(7) $Valency(C_1C_2) = Valency(C_1) + Valency(C_2)$.

Definition 4.4 *(Names and Processes). Names* $(x, y, z, u, v, w \ldots \in \mathcal{N})$ are atomic entities; *processes* have the following structure:

$$P ::= C\ \vec{x} \mid (P|P) \mid !P \mid (\nu x)P$$

where $\vec{x}$ is any string of *names*, and $Valency(C) = |\vec{x}|$ $(\equiv Length(\vec{x}))$.

Once again, we define the operational semantics of CAPI using a CHAM [Berry and Boudol 1992]. Section 2.2 presents a CHAM for API. The *Reaction* rule (Section 2.2) gets replaced by a new rule of the same name, as given below. Further, there are a few additional rules which specify the transition rules for the combinators. All the other rules remain the same.

Definition 4.5 *(Transformation Rules for CAPI).*

$$
\begin{array}{rcll}
Deg\ C\ \vec{y}x\ ,\ S\ xz & \mapsto & C\ \vec{y}xz & \text{(Reaction)} \\
Inv\ C\ x_1 \ldots x_{n-1}x_n & \rightharpoonup & C\ x_n x_1 \ldots x_{n-1} & \text{(Major Inversion)} \\
inv\ C\ x_1 \ldots x_{n-2}x_{n-1}x_n & \rightharpoonup & C\ x_1 \ldots x_{n-2}x_n x_{n-1} & \text{(Minor Inversion)} \\
Ref\ C\ x_1 \ldots x_n & \rightharpoonup & C\ x_1 \ldots x_n x_n & \text{(Reflection)} \\
R\ C\ x_1 x_2 x_3 \ldots x_n & \rightharpoonup & R\ x_1 x_2,\ C\ x_3 \ldots x_n & \text{(History)} \\
R\ x_1 x_2 & \rightharpoonup & & \text{(Cleanup)} \\
(\nu\ x)(C_1\ \vec{x}\ |\ C_2\ \vec{y}) & \rightleftharpoons & (\nu\ (C_1\ |\ C_2))\ x\vec{x}\vec{y} & (\nu - \text{Bonding}) \\
(C_1\ \vec{x}\ |\ C_2\ \vec{y}) & \rightleftharpoons & (C_1\ |\ C_2)\ \vec{x}\vec{y} & (\text{p} - \text{Bonding})
\end{array}
$$

where $Valency(C_1) = |\vec{x}|$ and $Valency(C_2) = |\vec{y}|$.

With the above rules on hand, we define $\hookrightarrow$: CAPI $\rightarrow$ CAPI below:

Definition 4.6 (Reduction relation for CAPI). We say that

(1) Q and R are *structurally equivalent* whenever $Q \overset{*}{\rightleftharpoons} R$

(2) the term Q reduces to R, in notation $Q \hookrightarrow R$, whenever $Q \overset{*}{\rightleftharpoons} Q'$, $Q' \mapsto R'$, and $R' \overset{*}{\rightleftharpoons} R$.

Following Milner and Sangiorgi [Milner 1991; Milner and Sangiorgi 1992], we define the notions of observable actions, barbed bisimulation, and barbed congruence for CAPI.

Definition 4.7 *(Observable Action in CAPI).* A process P can perform an *observable action*, $P\downarrow$, if

(1) P (or a *p-Bond* subcomponent of P) is structurally congruent to some process term $C_p\ \vec{z_p}$, where the leading basic combinator of C_p is *Deg*, and the trailing name of $\vec{z_p}$ is some x, such that there is no *ν-Bond* subcomponent of the form $(\nu\ x)$ in P or

(2) P (or a *p-Bond* subcomponent of P) is structurally congruent to some process term $S\ xy$, where for some pair of names x, y the name x does not occur in a *ν-Bond* subcomponent of the form $(\nu\ x)$ in P.

Definition 4.8 *(Barbed Bisimulation for CAPI).* A relation R_c over processes is a *barbed simulation* if $P\ R_c\ Q$ implies

(1) if $P \hookrightarrow P'$ then $Q \hookrightarrow Q'$ and $P'\ R_c\ Q'$ and

(2) $P\downarrow$ implies $Q\downarrow$.

Relation R_c is a *barbed bisimulation* if R_c and R_c^{-1} are *barbed simulations*. Processes P and Q are *barbed-bisimilar*, if $P\ R_c\ Q$ is true for some *barbed bisimulation* R_c.

Definition 4.9. A *process context* $\mathcal{C}[\ ]$ is a process term with a single hole, such that placing a process in the hole yields a well-formed process.

Definition 4.10 *(Barbed Congruence for CAPI).* Processes P and Q are *barbed-congruent*, written $P\ \sim_c\ Q$, if for each process context $\mathcal{C}[\ ]$ it holds that $\mathcal{C}[P]$ is barbed-bisimilar to $\mathcal{C}[Q]$.

5. EMBEDDING API INTO CAPI

In this section, we shall present a technique to translate any given API process term to an equivalent CAPI process term.

Definition 5.1. If $\vec{z}$ is a sequence of names, and y is any name, then let $\vec{z} \odot y$ denote the resultant of extracting all occurrences of y to the trailing position of $\vec{z}$; and let $\vec{z} \ominus y$ denote the resultant of taking away all occurrences of y from $\vec{z}$.

The following lemma essentially says that if we can arbitrarily permute a list of names, then we can permute all occurrences of the bound name to the end of the list and then combine them using the combinator *Ref*.

LEMMA 5.2. *For any CAPI process term $C\ \vec{z}$ and any name y we can construct $C_1\ C_2\ C\ \vec{z}$ such that $C_1\ C_2\ C\ (\vec{z} \ominus y)y \xrightarrow{*} C\ \vec{z}$ where $C_1 \in \{\text{Ref}\}^*$ and $C_2 \in \{Inv, inv\}^*$.*

PROOF. This is proved from the definition of *Inv*, *inv*, and *Ref*. □

We now introduce a pseudo name-abstraction operator $x^*(y)$ on CAPI terms.

THEOREM 5.3. *Given any CAPI term $C\ \vec{z}$ and names x, y we can construct an agent $x^*(y).C\ \vec{z}$ with the property*

$$x^*(y).C\ \vec{z}\ \mid\ \overline{x}w \hookrightarrow C\ \vec{z}\,[y \leftarrow w].$$

PROOF. It follows from the properties of the *Basic Combinators* and the above lemma that $x^*(y).C\ \vec{z}$ is given by $Deg\ C_1C_2R\ C\ \vec{z}_c\ x$ where $C_1 \in \{Ref\}^*$ and $C_2 \in \{Inv, inv\}^*$ and where $\vec{z}_c$ denotes $\vec{z} \ominus y$. □

Now, we present the formal translation from API to CAPI.

Definition 5.4 *(Transformation from API to CAPI).* Let P and Q denote API process terms. Let $[\![P]\!]$ and $[\![Q]\!]$ denote their translations in CAPI respectively. The general rules for translating P to $[\![P]\!]$ are

(1) $[\![0]\!] = \Lambda$ (where Λ denotes an empty string),
(2) $[\![\overline{x}y]\!] = \text{S}\ xy$,
(3) $[\![x(y)]\!] = (\text{Deg R}\ x)$ (this step is a special case of the following one and is included only for clarity),
(4) $[\![x(y).P]\!] = x^*(y).[\![P]\!]$,
(5) if $[\![P]\!] = C_p\ \vec{z_p}$ and $[\![Q]\!] = C_q\ \vec{z_q}$ then $[\![P \mid Q]\!] = (C_p \mid C_q)\ \vec{z_p}\vec{z_q}$,
(6) if $[\![P]\!] = C_p\ \vec{z_p}$ then $[\![!P]\!] = (!\ C_p\ \vec{z_p})$,
(7) if $[\![P]\!] = C_p\ \vec{z_p}$ then $[\![(\nu x)P]\!] = (\nu\ C_p)\ x\vec{z_p}$.

Note that the combinators really deal with sending and receiving names; the parallel, restriction, and replication operators are used as in the π-calculus. Hence it is easy to show that the combinatory version preserves the semantics of the original calculus.

THEOREM 5.5 (SEMANTIC CORRESPONDENCE). *Given any two process terms P, Q in API and the corresponding translated process terms $[\![P]\!]$, $[\![Q]\!]$ in CAPI we have the following:*

(1) *There exists a bisimulation R in API such that $P\ R\ Q$ if and only if there exists a bisimulation relation R_c in CAPI such that $[\![P]\!]\ R_c\ [\![Q]\!]$.*

(2) *P and Q are barbed-congruent in API ($P \sim Q$) if and only if $[\![P]\!]$ and $[\![Q]\!]$ are barbed-congruent in CAPI ($[\![P]\!] \sim_c [\![Q]\!]$).*

PROOF. It is easy to see from Definition 5.1 and Theorem 5.3 that the *reduction relations* of the two formalisms differ only in the *structural transformations*, but have identical *reaction transitions*. The transformation rules defining structural transformations are all deterministic and well founded, except the *bonding-rules*, which nevertheless do not introduce or take away any reaction transitions. In other words, the translation preserves reaction transitions. Thus, the two formalisms have precisely the same reaction transitions, thereby preserving bisimulation and congruence. □

6. RELATED WORK

6.1 Sequential Systems

The discipline of *combinators* in sequential programming has become a classic in our times. Following Schönfinkel [1924] and Curry [1930], there have been various related proposals to tame substitution in different contexts [Abadi et al. 1991; de Bruijn 1972; Curien 1993; Kennaway and Sleep 1988]. The combinators of Bernays [1959] and Quine [1959] were initially proposed to encode systems which do not employ self-application. The work reported in Raja and Shyamasundar [1995b] extends it to include self-application and higher-order functions, thereby encoding the λ-calculus.

6.2 Concurrent Systems

Though, there has been a phenomenal amount of work on combinators for sequential systems, there has not been much in the concurrency domain. The work in Cleaveland and Yankelevich [1994] comes close to providing a system of combinators. It models CCS with value-passing [Milner 1989] using explicit routing information between processes. However, unlike our work, it does not deal with processes which have a dynamic interconnection topology. While the technique of our article can be easily extended to capture CCS with value-passing, the method of Cleaveland and Yankelevich [1994] does not generalize to cover process calculi, which deal with dynamic interconnection topologies between processes.

The work in Honda and Yoshida [1994a; 1994b] constructs a combinatory version of a variant of API [Honda and Tokoro 1991]. They show that the concurrent composition of a small subset of fixed-form API processes can represent all API terms. In Honda and Yoshida [1994a] is a system that encodes a finite subset — process replication excluded — of API. In Honda and Yoshida [1994b] is a system that gives a combinatory encoding of process replication also. The systems propounded in these papers are geared to work only in the setting of a particular calculus and cannot be easily modified to suit other calculi as well. However, the framework of these papers is completely different from the one in this article. The notion of basic combinators and their rewrite rules are different, and the method of term formation is entirely different. One basic feature of Honda-Yoshida's combinators is that they are essentially a subset of the original API terms (indeed, modulo renam-

ing and behavioral equality, they give finite generators of the concerned algebra), which nevertheless can represent the original calculus. This feature is shared by Schonfinkel-Curry's combinators in the setting of λ-calculus and is important for its semantic use. This feature is not shared by the Quine-Bernays technique, though the Quine-Bernays technique enjoys a much wider applicability within a uniform and simpler scheme, which is clearly its advantage. Thus one can find here a clear difference in the orientations of the two approaches. Precisely due to this reason, the two approaches to "combinators for concurrency" offer complementary insights into the structure of concurrent calculi. Their technique clarifies the synchronization behavior of API processes, while our proposal of combinatory representation sheds more light on the distribution mechanism of the received value and gives a technique that can be applied to almost any concurrent calculi.

7. CONCLUSION AND FUTURE DIRECTIONS

Inspired by an unexplored technique of Quine in logic, we devised combinatory formulations in the setting of concurrent systems. We provided an alternative semantics for the asynchronous π-calculus in terms of combinators, by eliminating the need for bound names and the metalevel operation of substitution from the calculus. The combinators explicitly handle all the operational factors that arise in the communication of values between processes, while preserving the semantics of the original calculus. The same set of combinators are amenable to alterations to suit other process algebras operations as well. Process algebras which support process-passing as a primitive operation, e.g., CHOCS [Thomsen 1990], can be represented along the lines of the work reported in Raja and Shyamasundar [1995b] which extends the technique to capture higher-order languages such as the λ-calculus. Our future research goals include proving more refined algebraic equivalences, developing type-theoretic foundations, and exploring the relation of these combinators with *Interaction Nets* [Lafont 1990] and *Action Structures* [Milner 1992a]. The whole area of concurrent combinators is in a stage of infancy. Further research in this area will elucidate the structure of concurrent systems and will give valuable insights about the semantic structure of concurrency by providing various representability results. In analogy with the success of combinators in sequential systems, combinators for concurrency could have an impact on the theory of concurrency and the implementation of concurrent systems.

ACKNOWLEDGEMENTS

We wish to thank the anonymous referees for constructive comments which were of immense help in improving the content and presentation of this article. Our thanks to Margaret D'Souza for typing and typesetting this article.

REFERENCES

Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. 1991. Explicit substitutions. *J. Funct. Program. 1,* 4 (Oct.), 375–416.

Barendregt, H. 1984. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam.

Bernays, P. 1959. Über eine natürliche erweiterung des relationenkalkuls. In *Constructivity in Mathematics*, A. Heyting, Ed. North-Holland, Amsterdam, 1–14.

Berry, G. and Boudol, G. 1992. The chemical abstract machine. *Theor. Comput. Sci. 96*, 217–248.

Bodei, C., Degano, P., and Priami, C. 1996. Handling locally names of mobile agents. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*. Lecture Notes in Computer Science, vol. 1099. Springer-Verlag, Berlin, 490–501.

Boudol, G. 1989. Towards a lambda-calculus for concurrent and communicating systems. In *Proceedings of the International Joint Conference on the Theory and Practice of Software Development*. Lecture Notes in Computer Science, vol. 351. Springer-Verlag, Berlin, 149–161.

Boudol, G. 1992. Asynchrony and the π-calculus. Tech. Rep. 1702, INRIA, Sophia Antipolis, France.

Cleaveland, R. and Yankelevich, D. 1994. An operational framework for value-passing processes. In *Proceedings of the Annual Symposium on Principles of Programming Languages*. ACM, New York, 326–338.

Curien, P.-L. 1993. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Birkhauser, Boston.

Curry, H. 1930. Grundalagen der kombinatorischen logik. *Math. Annalen. 92*, 305–366.

Curry, H. and Feys, R. 1958. *Combinatory Logic*. North Holland, Amsterdam.

de Bruijn, N. 1972. Lambda-calculus notation with nameless dummies. *Indagationes Mathematicae 34*, 381–392.

Fiore, M., Moggi, F., and Sangiorgi, D. 1996. A fully-abstract model for the π-calculus. In *Proceedings of the Symposium on Logic in Computer Science*. IEEE, New York, 43–54.

Hewitt, C., Bishop, P., and Sterger, R. 1973. A universal modulator actor formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, San Mateo, Calif., 235–245.

Hindley, J. and Seldin. 1986. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, Cambridge, Mass.

Hoare, C. 1985. *Communicating Sequential Processes*. Prentice-Hall, London, U.K.

Honda, K. and Tokoro, M. 1991. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 512. Springer-Verlag, Berlin, 133–147.

Honda, K. and Yoshida, N. 1994a. Combinatory representation of mobile processes. In *Proceedings of the Annual Symposium of Programming Languages*. ACM, New York, 348–360.

Honda, K. and Yoshida, N. 1994b. Replication in concurrent combinators. In *Proceedings of TACS*. Lecture Notes in Computer Science, vol. 789. Springer-Verlag, Berlin, 786–805.

Kennaway, R. and Sleep, R. 1988. Director strings as combinators. *ACM Trans. Program. Lang. Syst. 10*, 4 (Oct.), 602–626.

Lafont, Y. 1990. Interaction nets. In *Proceedings of the Annual Symposium on Principles of Programming Languages*. ACM, New York, 95–108.

Milner, R. 1989. *Communication and Concurrency*. Prentice-Hall, London, U.K.

Milner, R. 1991. The polyadic π-calculus: A tutorial. Tech. Rep. ECS-LFCS-91-180, LFCS, Univ. of Edinburgh, Edinburgh, U.K. Oct. Also in *Logic and Algebra of Specification*, F. L. Bauer, W. Brauer, and H. Schwichtenberg, Eds. Springer-Verlag , 1993.

Milner, R. 1992a. Action structures. Tech. Rep. ECS-LFCS-92-249, LFCS, Univ. of Edinburgh, Edinburgh, U.K.

Milner, R. 1992b. Functions as processes. *Math. Struct. Comput. Sci. 2*, 2, 119–141.

Milner, R., Parrow, J., and Walker, D. 1992. A calculus of mobile processes. *Inf. Comput. 100*, 1–77.

Milner, R. and Sangiorgi, D. 1992. Barbed bisimulation. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*. Lecture Notes in Computer Science, vol. 623. Springer-Verlag, Berlin.

Peyton Jones, S. L. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall, London, U.K.

QUINE, W. 1959. Eliminating variables without applying functions to functions. *J. Symbol. Logic 24,* 4, 324–325.

QUINE, W. 1960. Variables explained away. *Proc. Am. Philos. Soc. 104*, 343–347.

RAJA, N. AND SHYAMASUNDAR, R. 1995a. Combinatory formulations of concurrent languages. In *Proceedings of the Asian Computing Science Conference.* Lecture Notes in Computer Science, vol. 1023. Springer-Verlag, Berlin, 156–170.

RAJA, N. AND SHYAMASUNDAR, R. 1995b. The quine-bernays combinatory calculus. *Int. J. Found. Comput. Sci. 6,* 4 (Dec.), 417–430.

RUSSELL, B. AND WHITEHEAD, A. 1912. *Principia Mathematica.* Cambridge University Press, Cambridge, U.K.

SCHÖNFINKEL, M. 1924. Über die bausteine der mathematische logik. *Math. Annalen 92*, 305–316. English translation with an introduction by W. V. Quine in *From Frege to Gödel*, J. van Heijenoort, Ed. Harvard University Press, 1967.

STARK, I. 1996. A fully abstract domain model for the π-calculus. In *Proceedings of the Symposium on Logic in Computer Science.* IEEE, New York, 36–42.

THOMSEN, B. 1990. Calculi for higher-order communicating systems. Ph.D. thesis, Imperial College, London Univ., London, U.K.

TURNER, D. 1979a. Another algorithm for bracket abstraction. *J. Symbol. Logic 44,* 2, 267–270.

TURNER, D. 1979b. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9*, 31–49.

Received April 1996; March 1997; accepted May 1997