Combinatory Formulations of Concurrent Languages

N.Raja and R.K.Shyamasundar

Computer Science Group,
Tata Institute of Fundamental Research,
Bombay 400 005, INDIA
e-mail: {raja,shyam}@tifrvax.tifr.res.in

Abstract. We design a system with six Basic Combinators, and prove that it is powerful enough to embed the full asynchronous \( \pi \)-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 combinatorial version, and show that the combinatorial version preserves the semantics of the original calculus. One of the distinctive features of this approach is that it can be used to rework many more process algebras in order to derive their equivalent Combinatory Versions.

1 Introduction

The discipline of combinatory logic [28, ?] began in the study of foundations of mathematics, to overcome the drawbacks of substitution in mathematical logic [27]. Subsequently, computer science gave impetus to research on combinators. The study of combinators has led to deep insights in the theory of sequential programming, and has also had a great influence in the implementation of functional programming languages.

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 there was little common meeting ground among the various models of concurrency, in the kind of primitives they employed. In the initial period of research on foundational models of concurrency [12, ?], 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 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 which employ similar mechanisms for communication of data have been designed [21, 20, 5, 29]. Influenced by Milner's ideas [18], in these process algebras, communication consists in sending and receiving a value synchronously through a shared port.
Consider the following parallel ("||") composition in the \(\pi\)-calculus [21]:

\[
x(y).P \parallel \bar{v}z.Q \leadsto P\{y \leftarrow z\} \parallel Q
\]

In the above expression \(x(y).P\) and \(\bar{v}z.Q\) are processes which communicate through the common port \(x\). The process \(\bar{v}z.Q\) sends the value \(z\) on port \(x\), and then activates \(Q\). The 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, which indicates that the name \(x\) binds the name \(y\). So, once again we encounter the horrid mechanism of substitution, with its attendant paraphernalia of binding mechanisms and bound entities.

At first sight, the problem of eliminating substitution in the process algebras appears to be simple. In the \(\pi\)-calculus, the values we substitute are always names, rather than processes (vice versa for certain other process algebras like CHOCs [29]). So, eliminating bound names seems to be easy. However, in comparison with the \(\lambda\)-calculus [2], the problem of ridding 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 \(\lambda\)-calculus. Most such calculi do not possess the \(<\text{operator}>\text{<operand}>\) kind of applicative structure found in the \(\lambda\)-calculus. Hence, the flow of information is not just confined to syntactically adjacent terms. The \(\lambda\)-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’ (\(\lambda\)) in the \(\lambda\)-calculus, while there are infinitely many ‘abstractors’ (infinitely many names) in the process algebras. There is only one other ‘operation’ in the \(\lambda\)-calculus (namely application), while in the process algebras there is a rich set of other ‘operations’ (for example, ‘\(|\)’, ‘\(+\)’, ‘\(\nu\)’, and ‘\(!\)’ in the \(\pi\)-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 paper 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 meta-level operation such as substitution. Such a combinatorial 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 paper, we shall work in the setting of the asynchronous \(\pi\)-calculus with replication. The combinators we design, arise from a technique that was formulated independently by Bernays [3] and Quine [24] 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 \(\pi\)-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 many other process algebras [21, 19, 20, 29, 5] in order to derive their combinatory formulations.