Type Systems for Concurrent Programming Calculi

N. Raja and R.K. Shyamasundar
School of Technology & Computer Science
Tata Institute of Fundamental Research
Mumbai 400 005, INDIA
Email: {raja, shyam}@tifr.res.in

Keywords: Type theory, concurrency, process calculi

Received: March 12, 2003

We explore the role of types *in models of concurrent computation, particularly in the concrete setting of the asynchronous π-calculus. The major theme of this work may be summarized by the slogan –* "Wherever you see structure, think of *types*". *We propose* type *annotations not merely to channels, but also to the highly structured set of processes. The type system guarantees that well typed expressions cannot go wrong. Polymorphic process types formalize extant informal ideas regarding the channel passing and process passing approaches to process mobility. Further, subtyping relation between process types distinguishes between true concurrency and nondeterministic choice.*

1 Introduction

Type systems for sequential programming languages lead to many advantages [3, 20, 27]. In programming practice: types help in structuring programs, they assist in compile-time error detection, and they are useful in optimizing the target code during the compiling process. In the theoretical study of programming language concepts: types help in the creation of succinct metalanguages that act as models for the study of real-life programming languages, and they serve as intermediate code in the task of providing mathematical semantics for programming languages. All this has naturally led to investigations regarding the role of types in theories of concurrency.

The goal of this paper is to examine whether there are any benefits to be gained by introducing *types* in models for concurrent computation. In this paper, we illustrate the role of *types* in the concrete setting of the asynchronous π-calculus (API) [17, 6]. We choose API as it is one of the most prominent calculi for concurrency and communication. API has two kinds of entities – *names* (also called *channels*) and *processes* (also called *agents*). *Names* do not possess any structure, whereas a good amount of structure is needed to build *processes*. The type system we propose, assigns types to both processes and channels. The type assigned to channels, characterizes the length and the nature of the elements that the channel may carry in a communication. The type assigned to processes, characterizes the set of actions that the process is committed to. This results in a rich notion of types which is very useful in the monadic as well as the polyadic versions of API. The type system proposed shows that there are substantial benefits to be reaped by exploring the idea of typing processes. The usage of our type system entails the following advantages:

- It provides a scaffolding for the structured use of the π-calculus, by which we can abolish certain undesirable features – like infinite concurrent activity – right at the early stage of building process terms, rather than at the stage of the reduction system.
- Guarantees *safety*, that well typed expressions will not go wrong.
- Does not constrain the expressive power of the π-calculus.
- Our type system, with minor changes, can be applied to all process algebra formalisms of concurrency. Thus, it provides a uniform basis for the relative assessment of various formalisms. For example, polymorphism in

process types brings out potential impredicativity in the semantics of some of these formalisms.

– Subtyping relation among process types helps in distinguishing true concurrency from nondeterministic choice.

The rest of this paper is organized as follows: Section 2 gives a brief review of the asynchronous π-calculus (API); Section 3 presents the type system; Section 4 shows that the type system preserves the semantics of API; Section 5 examines the type system with regard to those properties which are normally of interest in sequential languages; Section 6 explores further extensions to the type system – polymorphism and subtyping – by analogy with traditional type theory; Section 7 describes related work on concurrency and types. The conclusions and future research directions are presented in Section 8.

2 The Asynchronous π-Calculus

In this section, we include a brief review of the asynchronous π-calculus (API) [17, 6] notions that are required for this paper.

Following Milner's idea, a number of calculi for concurrent computation have been proposed, where the communication mechanisms are similar. Communication consists in synchronously sending and receiving through a shared labeled channel.

API [17, 6, 22, 25, 23, 35] is a model of concurrent computation that supports process mobility by naming and passing channels. It consciously forbids the transmission of processes as messages. One of its goals is to demonstrate that in some sense it is sufficiently powerful to allow only names to be the content of communications. API has two kinds of entities – names (channels), and processes (agents).

Names ($x, y, \ldots \in \mathcal{X}$), have no structure.

Processes ($P, Q, \ldots \in \mathcal{P}$) possess a well defined structure given by

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

The construct $\overline{x}y$ outputs the name y along x, and does not bind y. The construct $x(y)$ inputs a name, say y, along x, and binds y in the prefixed process. The word 'asynchrony' in this calculus means that message output is non-blocking. This is ensured by restricting the formation of a term $\overline{x}y.P$ in the π-calculus to the case where P is an inactive process. API is powerful enough to encode the synchronous message passing discipline of the π-calculus [36, 30]. The term 0 represents an inactive process. We have extended the π-calculus by including a constant process called ERROR, to represent the kind of type mismatches that we wish to avoid at run-time. The form $P|Q$ means that P and Q are concurrently active, are independent, and can also communicate. The operator "!" is called replication, and $!P$ means $P|P|\ldots$; as many copies as you wish. 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. The operator "ν" may also be thought of as creating new channels.

The operational semantics of API is given in two stages, as shown in Figure 1. A structural congruence is first defined over the process terms, and then a reduction relation is defined. Notice that the rules do not allow reduction under prefix or replication. Also, as expected there are no reduction rules for ERROR. For more details about API, the reader is referred to [17, 6].

3 The Type System

We present our type system in three stages – first, the syntax; second, the typing rules corresponding to API process constructors; and finally, the typing rules corresponding to the reduction system of API. The following subsections are devoted to each of these three stages respectively. Though we use the monadic asynchronous π-calculus to illustrate our typing system, our results can be extended to the polyadic case in a straightforward manner.

3.1 Syntax for Types

We shall call the type information assigned to names as *sort* (ranged over by the metavariable s), and shall use the term *type* (ranged over by the metavariable t) to designate the type information assigned to processes. Our typing scheme is an implicit one (Curry-style typing), because we want to illustrate our work in the setting of a familiar calculus, without any syntactic modifica-

Definition 2.1 (Structural Congruence over Process Terms)
$\equiv$ is the smallest congruence relation over process terms such that the following laws hold:

1. Processes are identified if they only differ by a change of bound names
2. $(\mathcal{P}/\equiv, |, 0)$ is a symmetric monoid
3. $!P \equiv P|!P$
4. $(\nu x)0 \equiv 0, (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
5. If $x \notin freeNames(P)$ then $(\nu x)(P|Q) \equiv P|(\nu x)Q$
6. $P|\textsc{Error} \equiv !\textsc{Error} \equiv (\nu x)\textsc{Error} \equiv \textsc{Error}$

Definition 2.2 (Reduction Relation)
The reduction relation $\rightarrow$ over processes is the smallest relation satisfying the following rules:

$$\text{Comm} \quad (\ldots + x(y).P) \mid (\ldots + \overline{x}[z].0) \;\rightarrow\; P\{y \leftarrow z\} \mid 0$$

$$\text{Par} \quad \frac{P \rightarrow P'}{(P|Q) \rightarrow (P'|Q)}$$

$$\text{Struct} \quad \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

$$\text{Res} \quad \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

Figure 1: Operational Semantics of API

Sorts	$s ::= BasicSort \mid (s)^R \mid (s)^S$
Type – Variables	$T \mid U \mid V$
Pre – Types	$\sigma ::= \epsilon \vert \phi \vert T \vert Name(Name : s)^R \vert Name(Name : s)^S \vert \sigma \rightarrow \sigma \vert \sigma \cap \sigma \vert \mu T.\sigma$
Pre – Types	$\sigma \mid \sigma_{ext} \mid \sigma_{int}$
Types	$t ::= < \sigma_{ext}, \sigma_{int} >$
TypeEnvironments	$\Gamma ::= \{\} \mid \Gamma, x : s \mid \Gamma, P : t$

Figure 2: Syntax of the Type System

tions to the term structure of the calculus.

The sort 's' denotes the length and nature of names which a given channel may carry in a communication. The superscripts R, S, indicate that the channel usage as "receive mode" and "send mode" respectively.

In API, processes may be viewed as programs which manipulate names (which in turn can be considered as data). As mentioned earlier the data manipulated by API programs are unstructured entities. The data develops some structure only in the polyadic extension of API. In the monadic case, the data are atomic entities while in the polyadic case they are n-tuples. Thus the notion of sorts starts making sense only in the polyadic case.

On the other hand, processes have a well-formed structure even in the monadic case; hence types are of significance in both versions of API. The type 't' denotes a process type; it comes in various forms as depicted in Figure 2. The *arrow type* arises due to the *prefix* constructor; the *intersection type* arises due to *par*; and the *recursive type* arises due to *Bang*; the *internal* and *external* types arise due to the *hiding* operator. The API expressions leading to the above types will become clear as we look at each of the typing rules given in the following subsections.

3.2 Types for Processes

An API process $\alpha.A$ can be regarded as an action α and a continuation A. $\alpha.A$ is called a commitment – it is a process committed to act at α [22]. This is precisely the information that the type associated with a process embodies.

Proposition 3.1 (Process-types and Commitment) *A process type describes the sequence of actions that a process is committed to.*

This will become clear from the following subsections.

API is based on the object model of computing [26]. Objects have an independent identity and they have a persistent state which may not be entirely visible to the other agents. Thus the type associated with a process has two facets – one which specifies its interface on the outside and the other which determines its internal transitions. Our type system brings out this aspect of API explicitly by making the type associated with a process to be a tuple comprising its external and internal types respectively.

The typing rules corresponding to each of the process constructors that API allows, are listed in Figure 3. Among all the typing rules listed in Figure 3, the internal and external types turn out to be distinct only when the 'hiding operator' occurs in the process term. Hence, only the *New-Channel* typing rule shows both components of the *type* associated with a process term. The types are to be viewed as being implicitly universally quantified on name sorts. The typing rules are given in a syntax directed way, and can be checked for well-formedness by structural induction over the API syntax.

Arrow types

Arrow types are familiar from type systems for sequential programming. The typing rule *Prefix-R* states in its premises that if x, y are names, y has *sort* s, and x has sort $(s)^R$ – which means that the channel x may be used for receiving a *name* of sort s – and the process P has type t; then the API term $x(y).P$ is assigned the type $x(y : s)^R \to t$. The type indicates that process $x(y).P$ can use channel x for receiving only, indicated by the superscript R. Further, after such a communication occurs (and only after), it may proceed to behave like a process having type t. This strict sequentiality imposed by the prefix constructor of API is made explicit by the $\to$. The rule *Prefix-S* is very similar except that it shows that the name x may be used only for sending (the superscript S) by the newly constructed process.

We shall discuss the prefix rule again when we consider higher-order models for concurrency. The *Prefix* type rules will reveal any impredicativity which could be lurking in the semantics of the calculus being typed. More about impredicativity will be discussed in Section 7.

Intersection types

The rule *Par-I* says that the *intersection type* '$\cap$' arises when a process is built by the parallel composition of two other process terms. The parallel composition operator '$|$' allows the components to make transitions independently (i.e., disjoint parallelism). Thus, the set of actions that a process belonging to an intersection type can indulge in,

Zero	$\Gamma \vdash 0 : \phi$
Prefix – R	$\dfrac{\Gamma\vdash x{:}(s)^R,\ y{:}s \quad \Gamma\vdash P{:}t}{\Gamma\vdash\ x(y).P\ :\ x(y{:}s)^R{\to}t}$
Prefix – S	$\dfrac{\Gamma\vdash x{:}(s)^S,\ y{:}s}{\Gamma\vdash\ \overline{x}(y)\ :\ x(y{:}s)^S}$
Par – I	$\dfrac{\Gamma\vdash P{:}t_1 \quad \Gamma\ \vdash\ Q{:}t_2}{\Gamma\vdash\ P\mid Q\ :\ t_1{\cap}t_2}$
Bang	$\dfrac{\Gamma\vdash P{:}(t_1{\to}t_2)}{\Gamma\ \vdash\ !P\ :\ \mu T.t_1{\to}(t_2{\cap}T)}$
New – Channel	$\dfrac{\Gamma\vdash P{:}t \quad \Gamma\vdash x{:}s}{\Gamma\ \vdash\ (\nu x)P{:}{<}t[x{\leftarrow}\epsilon],\ t{>}}$

Figure 3: Typing Rules for Process Constructors

is given by the conjunction of the set of possible actions of its component processes. Intersection types are also called 'conjunctive types' in the parlance of type theory.

There is a notable difference between the conventional usage of intersection types [3], and the way they are used in this work. In this work, the intersection type corresponds to a process constructor (par, '|'). Traditionally, intersection types are used for typing a term which belongs to various structurally unrelated types. For example, the symbol '+' is used to represent integer addition, and real addition. The type assigned to such a function is $((int \to int \to int) \cap (real \to real \to real))$. In other words, conventional intersection types are used to represent 'overloading'. Notably also absent from our type system, is the universal type ω (such that $P : \omega$ for all terms P), which accompanies intersection types normally.

The parallel composition operator '|' also allows the components to communicate. Hence we shall encounter the *intersection type* '$\cap$' once again in the typing rule describing communication between the two component processes.

Recursive Types

The *Bang* typing rule is another instance where the relevance of types in concurrency is very clearly brought out. The operator "!" is called replication and $!P$ – "bang P"– means $P|P\ldots$; as many copies as you wish. In API the "!" operator can be applied to any process term P to form the process $!P$ (where P has been constructed using any rule for building processes). The important point to be noted is that API does not enforce any restrictions on P before the "!" operator may be applied to it.

However the typing rule *Bang* states in its premise that the type of P should be an "arrow type" such as $(t_1 \to t_2)$ before we can apply "!" to P to get $!P$. This makes it mandatory that the outermost constructor of process P be a prefix, before the "!" may be applied to it. Thus the replication operator can be used on guarded processes only. $!\pi.P$ is a common instance of replication – it indicates a resource P which can be replicated only when a requester communicates via π. This shows that the premise in the typing rule *Bang* is meaningful. The next question which arises is whether the typing rule *Bang* is being too restrictive by imposing such a condition. Before we answer this query, let us examine the meaning of a term such as $!P$ when it is not required of P that its outermost constructor be a prefix. Such a term, "$!P$", means a resource which replicates asynchronously – replicates without demand, without requirement. $!P$ appears to be acting on its own *free will*, so to say. In other words it represents *infinite concurrent activity.* Now this is certainly not a meaningful construct, and we would rather not have such a term in our calculus. Hence the typing rule *Bang* does not strip API off any expressive power; in fact it rules out an entire class of meaningless terms from being constructed. API abolishes such behaviour by taking recourse to its reduction rules. However we have done better in our type system, in that, we even forbid the occurrence of such terms right at the level of syntax, by enforcing a discipline in the structured construction of API programs.

After having looked at the premise, let us now examine the conclusion of the *Bang* rule. It infers that the process term $!P$ has the type $(\mu T.t_1 \rightarrow (t_2 \cap T))$. μ represents recursion and the type variable T is the parameter of the recursion. The recursive type makes the recursive behavior of "!" operator explicit. The intuition provided by the recursive type is well supported when we turn to API and find that all parametric recursive process definitions can be encoded by replication. Let us come back to the *Bang* typing rule: When P has the type $(t_1 \rightarrow t_2)$, it means that P behaves as dictated by the type t_1 and then (sequentially) behaves as dictated by t_2. The recursive type assigned to $!P$ says that $!P$ behaves as required by t_1 and then as required by $(t_2 \cap T)$. The intersection type mirrors the fact that an independent process of type t_2 has been spawned, which executes in parallel with the resource of type T. But T is the parameter of recursion, and we eliminate it by recursive unfolding, that is we replace T by $(\mu T.t_1 \rightarrow (t_2 \cap T))$ and proceed further as before.

The Recursive type in this setting is very similar to that used in sequential programming. The type $\mu T.t_p$ stands for the least fixed point solution of the type equation $T = t_p$. The solutions of such equations will be infinite types, which can be represented by infinite labeled binary trees. The definition of such trees is provided in Figure 4. The same Figure also gives a congruence relation on types with the help of such trees [9, 10].

Internal and External Types

In all the typing rules that we have considered so far, the external and internal types are identical. However, the operator ν used as $(\nu x)P$ localizes (restricts) the use of the channel x within P. The channel name x is guaranteed to be different from any other channel name which finds an occurrence outside P. Hence communications can be sent and received on x only internally within process P. This brings us to the next typing rule, *New-Channel*, which gives the external and internal type of a process term which has been built using the operator ν. The notion of distinguishing between the external and internal type of a process is derived from the notion of existential types and explicit witnesses [28], and the notion of partially abstract types [8]. The *external* type-component states that if the process P has type t and the channel x has sort s, then the external type of the process term $(\nu x)P$ is $t[x \leftarrow \epsilon]$ which means that in the type t all occurrences of x are replaced by ϵ, thereby making the channel x unavailable for communication with the outside world. The *internal* type-component states that as far as the internal type of $(\nu x)P$ is concerned, there is no change, the type continues to be t.

That explains all the typing rules that have arisen because of the process constructors that are allowed in API.

3.3 Reduction rules and Types

The typing rules shown in Figure 4 correspond to the congruence relation over types. They spell out when two types may be considered to be congruent.

The remaining typing rules, shown in Figure 5, correspond to the reduction system of API. The typing rules *Inter-E, Comm, Par-R, Res*, and *Struct* tell us how to consistently infer the type of the term which results from a reduction.

The rule *Comm* mentions the types required of each term so that the communication between the two processes will result in a proper reduction (one which does not result in ERROR), and gives the type of the resultant process. The rule *Par-I* mentioned earlier as giving rise to the *intersection type* '$\cap$', can be considered to be a special case of this rule. If there is no communication possibility allowed by the types of the interacting processes

Definition 3.2
The tree corresponding to the process type t, written as $T(t)$, is defined as follows:

$T(\phi) \ = \ \phi;$

$T(t_1 \ \rightarrow \ t_2) \ = \ (\rightarrow, \ T(t_1), \ T(t_2));$

$T(t_1 \ \cap \ t_2 \ = \ (\cap, \ T(t_1), \ T(t_2));$

$T(\mu T.\ t) \ = \ T(t[T \ \leftarrow \ \mu T.\ t]).$

Definition 3.3
$\approx_t$ is the smallest congruence relation over types, such that, the following laws hold:

CR-1 Process types are identified if they only differ by a change of bound names;

CR-2 $t \cap \phi \approx_t \phi \cap t \approx_t t$;

CR-3 $t_1 \ \approx_t \ t_2$, if $T(t_1) \ = \ T(t_2)$.

Figure 4: Congruence Relation for Types

Inter − E	$\dfrac{\Gamma \vdash P{:}t_1 \cap t_2}{\Gamma \vdash P{:}t_1 \qquad \Gamma \vdash P{:}t_2}$
Comm	$\dfrac{x(y).P \ : \ (x(y{:}s)^R \rightarrow t_P), \quad \overline{x}(z) \ : \ (x(z{:}s)^S)}{x(y).P \ \mid \ \overline{x}(z) \ \rightarrow \ P\{y \leftarrow z\} \ : \ t_P\{y \leftarrow z\}}$
Par − R	$\dfrac{P{:}t_P \ \rightarrow \ P'{:}t_{P'}}{(P \mid Q) \ : \ t_P \cap t_Q \ \rightarrow \ (P' \mid Q) \ : \ t_{P'} \cap t_Q}$
Res	$\dfrac{P{:}{<}t_P,t_P{>} \ \rightarrow \ P'{:}{<}t_{P'},t_{P'}{>}}{(\nu x)P{:}{<}t_P[x \leftarrow \epsilon],t_P{>} \ \rightarrow \ (\nu x)P'{:}{<}t_{P'}[x \leftarrow \epsilon],t_{P'}{>}}$
Struct	$\dfrac{t_Q \equiv_t t_P \quad P{:}t_P \rightarrow P'{:}t_{P'} \quad t_{P'} \equiv_t t_{Q'}}{Q{:}t_Q \rightarrow Q'{:}t_{Q'}}$

Figure 5: Reduction Rules and Types

(disjoint parallelism), then the resulting type of the compound term is given by the *Par-I* typing rule. It is worth noting that the typing rules corresponding to process constructors and the typing rules corresponding to the reduction system, cannot be kept separated in the type system for API. This is because the operator *par* '|' is overloaded – it represents both concurrency (a process building operation), as well as communication (an operation which is a part of the reduction rules). However, such a clear separation can be achieved in the case of a type system constructed along similar lines for Boudol's *concurrent* λ*-calculus* [5].

Once again we mention that in all these rules, except *Res* the inference is valid for both components of the process type – external as well as internal. The *Res* typing rule explicitly indicates the process type as a tuple and gives the corresponding new components of the type after reduction.

4 Soundness and Type Safety

In this section, we examine the effect of the type-system on the semantics of API. First, we show that our type system preserves the semantics of API, and prove that well typed expressions never reduce to ERROR – which means process types guarantee the safety property.

The operational semantics of API was defined in two stages [22, 26] as shown in Section 2. A structural equivalence on process terms was given first, and then a reduction relation was given which describes the act of communication. We prove below that our notion of *type* is consistent with each of these two stages.

Theorem 4.1 *Types preserve the structural congruence rules on process terms.*

Proof: We prove this theorem by examining the structure of the definition of structural congruence on process terms.

1. Types respect α-conversion (typing rule *CR-1*), hence agents are identified if they only differ by a change of bound names.

2. Using the typing scheme presented in this paper, we show that types preserve the fact that $(\mathcal{P}/\equiv, |, 0)$ is a symmetric monoid.

   $0 : \phi \quad (Zero)$

   $P|0 : t \cap \phi \quad (Par-I)$

   $t \cap \phi \approx_t t \quad (CR-2)$

   Similarly,

   $0|P : \phi \cap t \quad (Par-I)$

   $\phi \cap t \approx_t t \quad (CR-2)$

   By steps 3 and 5, it follows that types preserve the monoidal structure of $P/\equiv$, where '|' is the associative operator of the monoid, and 0 forms the identity w.r.t '|'.

3. The typing rule *Bang* has been explained in sufficient detail in Section 3. It clearly follows from the illustration given there that types guarantee $!P \equiv P|!P$.

4. The inactive process 0 has the type ϕ as both its external and internal type. The restricted process $(\nu x)0$ continues to have the same type. Hence $(\nu x)0 \equiv 0$.

   If the process P has the process type $< E_P, I_P >$ then the process term $(\nu x)(\nu y)P$ has the type $< E_P[x \leftarrow \epsilon, y \leftarrow \epsilon], I_P >$ which is equivalent to the type $< E_P[y \leftarrow \epsilon, x \leftarrow \epsilon], I_P >$ associated with the process term $(\nu y)(\nu x)P$.

5. From the typing rules *Par-I*, and *New-Channel* it immediately follows that if x is not free in P then $(\nu x)(P|Q) \equiv P|(\nu x)Q$.

Thus *types* preserve the structural congruence on process terms. $\square$

Theorem 4.2 *Well typed expressions can never reduce to* ERROR.

Proof: In the absence of types, the reduction rule which allows communication between process terms states that $x(y).P \mid \overline{x}z \rightarrow P\{y \leftarrow z\}$. The typing scheme assigns to each of the two concurrent process terms the following types –
$x(y).P : x(y:s)^R \rightarrow t_P$, and $\overline{x}(z) : x(z:s)^S$
Further the type scheme allows a reduction to take place by the typing rule *Comm* only when the two types are complementary and the sorts of the channels being used for communication are consistent with each other. These are exactly the conditions required to ensure a meaningful reduction in the π-calculus. The term resulting from the communication is $P\{y \leftarrow z\}$ and its corresponding type is $t_P[y \leftarrow z]$. Then well typed process terms never reduce to ERROR. $\square$

Theorem 4.3 *The type system preserves the semantics of* API.

Proof: Follows as a direct consequence of Theorem 4.1 and Theorem 4.2. □

5 Basic Syntactic Properties

The type system proposed in this work is meant for concurrent calculi, and as is well known, the requirements of concurrent systems are quite different from those of sequential systems. However, there are a number of syntactic properties which have been of interest in traditional type systems for sequential programming [3]. For the sake of completeness we briefly examine such properties in our type system.

1. **Implicit Typing:** The typing scheme we have proposed is an implicit one (Curry-style typing). We chose Curry-style typing because we wanted to illustrate our work in the setting of a familiar calculus without requiring major syntactic modifications to the term structure of the calculus.

2. **Church-Rosser Property (CR):** This is more a property of the underlying calculus being typed, rather than the type system itself. In our case, API does not satisfy the Church-Rosser property, since functions such as 'parallel-or' can be represented in it.

3. **Subject Reduction (SR):** If process term P has the type t_P, and if P reduces to the term P'; then the subject reduction property states that the type of P' is also t_P. Such a property does not hold in our type system because process reduction in API is nondeterministic, and also due to name passing, the interface of a process may change with reduction.

4. **Strong Normalization (SN):** This property states that all reduction sequences terminate eventually. This means that not every computable function is definable in the system. However this property does not hold in our type system because of the presence of recursive process types. With the help of recursive process types we are able to type the "!" operator of API without restricting its expressive power.

5. **Type Checking:** This property states whether, given a typing environment Γ, a process term P, and a type t, is the judgment $\Gamma \vdash P : t$ decidable or not. Type checking is decidable for our type system.

6. **Type Inference:** This requires that given Γ and P, it should be possible to compute a t such that $\Gamma \vdash P : t$ is valid. Type inference is possible for process types.

The above properties gained prominence because of their importance in the traditional application areas of types, such as in proof theory and in sequential programming. In the domain of concurrency, many of the above properties such as CR, SR, and SN are no longer relevant. Instead, properties such as *safety* and *liveness* become important.

6 Further Extensions to the Type System

There are a number of concepts which have played a significant role in the success of type disciplines for sequential systems. Two such concepts are *Polymorphism* and *Subtyping*. In this section we examine whether these concepts shed any light on concurrent calculi. We informally extend our type discipline in two directions – to incorporate polymorphism and subtyping. The results are indeed very promising as we demonstrate in the following subsections. Further research along these lives is sure to lead to insights into concurrent calculi.

6.1 Channel passing versus Process passing

Many distinct formalisms [25, 29, 37, 1, 18, 2, 5] have been invented to describe systems which do not have fixed interconnection topology between processes. All such formulations may be classified into two groups by examining the way in which they achieve mobility. One group achieves mobility by allowing channel names to be communicated [25, 1, 18] – the π-calculus belongs to this group. The other group achieves mobility by supporting the transmission of processes as messages [37, 2, 29, 5] – let us take a particular example from this group, say CHOCS [37].

The name passing approaches to concurrency allow names, but not processes, to be transmitted in communications. On the other hand, the process passing approaches allow processes, but not names, to be transmitted as messages. There are relevant reasons why each of these two approaches allows only either names or processes but not both to be the content of communications. Thus neither of the two approaches can be said to have achieved "uniformity" in dealing with their primitive entities. Further it has been demonstrated [37, 22, 36] that both the paradigms are equally powerful as far as their expressive power is concerned.

The question that we ask now is whether our type system can provide any relevant criteria that favours the choice of one paradigm over the other? The answer is in the affirmative – the type system does provide a measure which helps in discriminating the two paradigms.

In order to see how, let us examine the type that our system assigns to the process constructor which allows abstraction of names and processes in the paradigms of name-passing and process-passing calculi respectively. In this section, let x, y range over *Names*; P, Q range over *Processes*; N_s range over *Name Sorts*; P_t and t_q range over *Process Types*.

Consider the following π-calculus term, and its corresponding type – $x(y).Q : \forall N_s.x(y : N_s)^R \to t_q$. The type expression states that the process term $x(y).Q$ behaves like a program which expects any name y as input (y is a dummy parameter), and then behaves like the process Q. However there is no restriction on what sort of name it can accept as input, as shown by the universal quantifier which ranges over N_s. The important point to be observed is that the entity "$\forall N_s.x(y : N_s)^R \to t_q$" is itself a process type and does not lie in the range of the universal quantifier (which ranges only over name sorts in this case).

Now consider the following CHOCS term, and its corresponding type – $x?(P).Q : \forall P_t.x(P : P_t)^R \to t_Q$. In this case the type expression states that the process term $x?(P).Q$ behaves like a program which expects any P process as input (P is a dummy parameter), and then behaves like the process Q. However the program does not impose any restrictions on the type of the input process (represented by the universal quantifier ranging over P_t. In this case the entity "$\forall P_t.x(P : P_t)^R \to t_Q$" is itself a process type and hence the universal quantifier ranges over this type as well. In other words process types turn out to be impredicative in CHOCS, while they remain predicative in the π-calculus.

It is a well known phenomenon in type theory that the semantics of a predicative formalism is extremely simple and elegant in comparison with the semantics required by an impredicative formalism [11]. Thus conceptual simplicity and elegance in the semantics of the type system associated with a formalism favours π-calculus over CHOCS – or in more general terms, name passing approaches over process passing approaches to concurrency.

6.2 True Concurrency versus Nondeterministic Interleaving

As mentioned in Section 7, the work by Pierce and Sangiorgi has shown that the subtyping relation among name sorts leads to an interesting refinement. In this subsection we examine the relevance of subtyping relation among process types.

In the semantic theories for process algebras such as CCS [21] and CSP [14], concurrency is semantically reduced to nondeterminism. For example the process $a|b$ is considered semantically equivalent to the process $(a.b + b.a)$. It has been demonstrated by Boudol et al. [7], that in certain situations it is meaningful to retain concurrency as a primitive concept without reducing it to nondeterministic interleaving. We now show that process types can be used to maintain such a distinction.

For this purpose we introduce *union types*, '$\cup$', and a *subtyping* relation among union types. Consider a process term of the form $P+Q$. This term can (nondeterministically) indulge, either in the actions specified by P or in the actions specified by Q (exclusive-or of the actions). If the types of P and Q are given by t_p and t_q respectively, then we assign to the process $P + Q$, the type $t_p \cup t_q$. Now we define the subtyping relation '$\subseteq$', by the relations, $t_p \subseteq (t_p \cup t_q)$ and $t_q \subseteq (t_p \cup t_q)$. The *subtyping* relation is reflexive, antisymmetric, and transitive. Intuitively in a context which requires an object of type t, one could as well use an object whose type is a subtype of t, but not vice versa. This intuition is well supported when we examine the process terms themselves. It is important to note that such a subtyping relationship does not hold in the case of intersection types i.e. $t_p \not\subseteq (t_p \cap tq)$ and $t_q \not\subseteq (t_p \cap t_q)$.

Thus we get the type of $a|b$ as $(t_a \cap t_b)$ and the type of $((a.b) + (b.a))$ as $((t_a \rightarrow t_b) \cup (t_b \rightarrow t_a))$. Consider the above processes after they make a transition on 'a'. $(a.b) + (b.a)$ reduces to b. The new process type is a subtype of the original process type, i.e. $t_b \subseteq ((t_a \rightarrow t_b) \cup (t_b \rightarrow t_a))$. On the other hand the process $a|b$ also reduces to b. But the distinction lies in the fact that the new process type is not a subtype of the original process type, i.e. $t_b \not\subseteq (t_a \cap t_b)$. Thus the type equivalence provided by the subtype relation provides a key to distinguish true concurrency from nondeterministic interleaving.

7 Related Work

In this section we briefly discuss work related to type systems for mobile processes. As mentioned earlier, the concurrent calculi that were proposed following Milner's CCS, have two basic syntactic entities – *channels* and *processes*. This situation is unlike that in sequential programming, where the λ-calculus (the de-facto standard sequential language), has only one basic entity – *terms*. Till now a major part of the research on type systems for concurrency has concentrated on assigning type information to the channels only. Such type information has been called *sorts*.

The relevant starting point is the notion of *sorts* introduced in the polyadic π-calculus by Milner [22]. We illustrate Milner's notion of *sorting* with an example. Consider the process term $\overline{x}y.0|x(u).\overline{u}().0|\overline{x}z.0$. In this expression, channels y and z carry only the empty vector if they are ever used for communication. On the other hand, channel x always carries another channel name, which in turn is used in communicating an empty vector. We can represent these observations as: $\{y \mapsto (), z \mapsto (), x \mapsto (())\}$. Notice that the usage of x is characterized by a nesting of parentheses. The above representation is precisely the *sorting* as proposed by Milner. Thus the *sort* associated with a channel captures the length and nature of the vector that the name carries in communications. In the polyadic π-calculus, names may carry n-tuples of other names. Hence the notion of sort information assumes prominence only in the polyadic setting. There are some more points to be noted. Firstly, sort information is assigned to channels only and sort equivalence is by *name matching*. Secondly, names occurring in a perfectly meaningful π-calculus process term may not have any *sorting* at all. This can occur if a term uses names to communicate different entities at different times. Thus the lack of a proper *sorting* does not render a π-calculus expression meaningless. Finally, *sorts* are implicit i.e., they do not occur in the term structure of the calculus. Honda [15] presented similar results, in an independent work. Gay [12] presents an algorithm (quadratic in the length of the input process) for automatically inferring such sort information for channels, from the given π-calculus term. Naturally *sorts* are inferred only if they exist. Honda and Vasconcelos [16] gave an algorithm to the same effect, though linear in the size of the input process. Following Lafont's work on interaction nets [19], Honda proposed conditions on channel sorts, so as to achieve freedom from deadlock in certain finite and simple situations.

Pierce and Sangiorgi [31] extended the notion of sorts by distinguishing between the ability to read from a channel, the ability to write to a channel, and the ability to do both. This refinement gives rise to an interesting subtype relation on channel sorts. Their sort equivalence is by *structural matching*. In Pierce's work, sorts appear explicitly in the term structure and further such sort information is even communicated from one process to another. This requires changes in the π-calculus model, thus resulting in a different concurrent calculus. In Pierce's work, the problem of algorithmic inference of sort information is not considered at all.

The idea of assigning type information to processes has also been used by researchers in other contexts [29, 13, 34]. In Facile, CML, and the Typed λ-calculus with first class processes, the notion of process type is present. However the process types which find usage in these programming languages are predominantly just functional types. The notion of polymorphism has been included in Facile and CML, but once again in the realm of channel sorts, in order to derive more flexible sorting mechanisms.

From the above observations it is clear that the notions of type inference, polymorphism, subtyping, and conditions for deadlock freedom have been explored in the domain of channel sorts. Such investigations in the domain of process types, would yield rich dividends [33, 4, 32, 38, 39].

8 Conclusions and Future Directions

The aim of this work was to establish a bridge between the disciplines of concurrency and type theory. We presented a novel operational semantics for the asynchronous π-calculus, by making reductions sensitive to type. Our type system was unique, in not confining type information to channels only; very informative types were assigned to processes also. The universe of process terms with its rich structure, proved to be a fertile ground for the application of various type constructors. The type system did not restrict the expressive power of the asynchronous π-calculus in any way. Types guaranteed *safety*, that well typed expressions would not go wrong. Further the type system helped in preventing the construction of meaningless expressions, such as those representing infinite concurrent activity, right at the stage of syntactic formation of process terms. The notion of polymorphism brought out the latent impredicativity in the semantics of the process-passing approaches to concurrency. The notion of subtyping helped in distinguishing true concurrency from nondeterministic interleaving.

As further work, it would be highly interesting and relevant to explore how the notion of process types could be put to use in reasoning about *liveness* properties of concurrent systems, such as freedom from deadlock. It would also be fruitful to pursue work towards establishing algebraic equivalences over process types. Also as discussed in the last section, exploring the notions of polymorphism and subtyping looks promising.

This work is part of an ongoing investigation into the role of type theoretic concepts in the setting of concurrency. It would also be productive to carry out such an investigation in a more abstract formalism for concurrency, e.g., like the one provided by *action structures* [24].

Acknowledgment

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

References

[1] E. Astesiano, and G. Reggio (1984) Parametric Channels via Label Expressions in CCS, *Theor. Comp. Science*, Vol. 33, pp. 45–64.

[2] E. Astesiano and G. Reggio (1987) SMoLCS-driven concurrent calculi, *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 249, pp. 169–201.

[3] H. Barendregt, and K. Hemerik (1990) Types in lambda calculi and programming languages, *Proc. ESOP'90*, LNCS 432, pp. 1–36.

[4] M. Berger, K. Honda, and N. Yoshida (2003) Genericity and the π-Calculus, *Proc. FOSSACS'03*, LNCS, To appear.

[5] G. Boudol (1989) Towards a lambda-calculus for concurrent and communicating systems, *Proc. TAPSOFT'89*, LNCS 351, Springer-Verlag, pp. 149–161.

[6] G. Boudol (1992) Asynchrony and the π-calculus, *Rapport de Recherche*, Number 1702, INRIA Sophia-Antipolis.

[7] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn (1991) Observing Localities, *INRIA Report No. 1485.*

[8] L. Cardelli, and P. Wegner (1985) Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, Vol. 17 (4).

[9] F. Cardone, and M. Coppo (1990) Two Extensions of Curry's Type Inference System, *Logic and Computer Science*, Academic Press, pp. 19–75.

[10] B. Courcelle (1983) Fundamental Properties of Infinite Trees, *Theoretical Computer Science*, Vol. 25, pp. 95–169.

[11] R.L. Constable (1991) Type Theory as a Foundation for Computer Science, *Proc. TACS'91*, Lecture Notes in Computer Science, Vol. 526, Springer-Verlag.

[12] S. Gay (1993) A sort inference algorithm for the polyadic π-calculus, *Proc. ACM Symposium on Principles of Programming Languages*, ACM Press.

[13] A. Giacolone, P. Mishra, and S. Prasad (1989) Facile: A symmetric integration of concurrent and functional programming, *Int. Jl. of Parallel Prog.*, Vol. 18, pp. 121–160.

[14] C.A.R. Hoare (1985) Communicating Sequential Processes, *Prentice-Hall*, London.

[15] K. Honda (1993) Types for Dyadic Interaction, *Proc. CONCUR'93*, Lecture Notes in Computer Science, Volume 715, Springer-Verlag.

[16] K. Honda, and V.T. Vasconcelos (1993) Principal typing schemes in a polyadic π-calculus, *Proc. CONCUR'93*, LNCS 715, Springer-Verlag.

[17] K. Honda, and M. Tokoro (1991) An Object Calculus for Asynchronous Communication, *ECOOP'91*, Lecture Notes in Computer Science, Volume 512, Springer-Verlag.

[18] S.R. Kennaway and M.R. Sleep (1985) Syntax and informal semantics of DyNe, a parallel language, *LNCS 207*, Springer-Verlag, pp. 222–230.

[19] Y. Lafont (1990) Interaction Nets, *Proc. POPL'90*, ACM Press, pp. 95–108.

[20] B. Mahr (1993) Applications of Type theory, *Proc. TAPSOFT'93*, Lecture Notes in Computer Science, Volume 668, Springer-Verlag.

[21] R. Milner (1989) Communication and Concurrency, *International Series in Computer Science*, Prentice Hall.

[22] R. Milner (1991) The polyadic π-calculus: a tutorial, *Logic and Algebra of Specification*, Proceedings of International NATO Summer School (Marktoberdorf, Germany), Series F, Vol. 94, Springer.

[23] R. Milner (1999) Communicating and Mobile Systems: The Pi Calculus, Cambridge University Press.

[24] R. Milner (1993) Action Structures and the π-Calculus, *Proof and Computation*, Proceedings of International NATO Summer School (Marktoberdorf, Germany), Series F, Vol. 139, Springer.

[25] R. Milner, J. Parrow, and D. Walker (1992) A calculus of mobile processes (Parts I and II), *Information and Computation*, Vol. 100, pp. 1–77.

[26] R. Milner (1992) Functions as processes, *Journal of Mathematical Structures in Computer Science*, Vol. 2 (2), pp. 119–141.

[27] J.C. Mitchell (1990) Type Systems for Programming Languages, *Handbook of Theoretical Computer Science*, Elsevier Science Publishers.

[28] J.C. Mitchell, and G. Plotkin (1988) Abstract Types have Existential Types, *ACM Transactions on Programming Languages and Systems*, Vol. 10 (3).

[29] F. Nielson (1989) The typed λ-calculus with first class processes, *Proc. PARLE'89*, Lecture Notes in Computer Science, Volume 366, Springer-Verlag.

[30] C. Palamidessi (1997) Comparing the expressive power of the Synchronous and the Asynchronous pi-calculus, *Proc. ACM Symposium on Principles of Programming Languages*, ACM Press, pp. 256–265

[31] B. Pierce, and D. Sangiorgi (1993) Typing and Subtyping for Mobile Processes, *Proc. IEEE Symposium on Logic in Computer Science*, IEEE Press.

[32] B. Pierce, and D. Sangiorgi (2000) Behavioral Equivalence in the Polymorphic Pi-Calculus, *Proc. Journal of ACM*, Vol. 47 (3) pp 531–584.

[33] N. Raja, and R.K. Shyamasundar (1994) Type Systems for Concurrent Calculi, *Proc. of the Tenth Workshop on Abstract Data Types* (ADT'94), Santa Margherita Ligure, Genoa, Italy.

[34] J.H. Reppy (1993) Concurrent ML: Design, Application and Semantics, *Funct. Prog., Concurrency, Simulation and Automated Reasoning*, LNCS 693, Springer-Verlag.

[35] D. Sangiorgi, and D. Walker (2001) The Pi-Calculus – A Theory of Mobile Processes, Cambridge University Press.

[36] D. Sangiorgi (1993) From π-calculus to Higher-Order π-calculus — and back, *Proc. TAPSOFT '93*, Lecture Notes in Computer Science, Volume 668, Springer-Verlag.

[37] B. Thomsen (1993) Plain CHOCS. A Second Generation Calculus for Higher Order Processes, *Acta Informatica*, Vol. 30 (1), pp. 1–59.

[38] D.N. Turner (1996) The Polymorphic Pi-Calculus: Theory and Implementation, *Ph.D. Thesis*, University of Edinburgh.

[39] V. Vasconcelos (1994) Typed Concurrent Objects, *Proc. ECOOP'94*, LNCS, Springer-Verlag, pp. 100–117.