Some Type Systems for the Pi Calculus

Simon Gay
Department of Computer Science
Royal Holloway, University of London
Egham, Surrey, TW20 0EX, UK.
<S.Gay@dcs.rhbnc.ac.uk>

March 18, 1999

1 Introduction

The π-calculus [17, 18, 19] is a foundational calculus of concurrent systems, descended from CCS [15]. A typical system consists of a number of parallel components which communicate by sending and receiving messages on channels. The key feature of the π-calculus is that channel names are first class objects, and can be incorporated into messages; indeed, in the purest form of the π-calculus, channel names are the only kind of data. The ability to commmunicate channel names as messages permits a rich class of system behaviour to be described, including dynamic reconfiguration of connection topologies. Processes, represented by the names which give access to them, are in effect allowed to move around a system, and for this reason the π-calculus is referred to as a calculus of *mobility*.

In the years since its introduction, the π-calculus has been used for many purposes, including specification and verification of concurrent systems, and the analysis of high-level concurrent language features [26, 34]. It has also been implemented and developed into the Pict programming language [27]; we will have more to say about this later. When the π-calculus is considered as the core of a programming language, it is natural to introduce types. The primary purpose of a type system for the π-calculus is to guarantee certain properties of the run-time behaviour of processes. The usual tension exists between safety and expressiveness, and therefore a range of type systems have been proposed, guaranteeing various run-time properties and supporting various programming styles. Some of the type systems also assist with reasoning about process equivalence. This may happen for two reasons, the first of which is that in some cases the run-time behavioural guarantees are directly helpful in establishing equivalences. The other is that in practice it may be sufficient to know that two processes behave equivalently if they are only ever executed in environments of certain restricted kinds, and these restrictions may correspond to obeying the constraints of a particular type system.

In this article we will present the key ideas and features of a number of type systems for the π-calculus (or sometimes extensions of it). We will describe the typing rules and how they work, the run-time properties which are guaranteed, and the proof techniques which are used to establish the desired properties of well-typed processes. We will also indicate where results from typing can be used to reason about process behaviour. We will use a consistent notation to present all of the type systems, even when different notations were used in the original papers, and we will begin with a summary of the syntax and semantics of the π-calculus.

1.1 The π-calculus

There are many variations of the π-calculus, some more significant than others, but the differences in syntax between the presentations surveyed in this article are fairly small. The following

$$
\begin{array}{rcll}
P \mid \mathbf{0} & \equiv & P & \text{S-UNIT} \\
P \mid Q & \equiv & Q \mid P & \text{S-COMM} \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & \text{S-ASSOC} \\
(\nu x)P \mid Q & \equiv & (\nu x)(P \mid Q) \text{ if } x \text{ is not free in } Q & \text{S-EXTR} \\
(\nu x)\mathbf{0} & \equiv & \mathbf{0} & \text{S-NIL} \\
(\nu x)(\nu y)P & \equiv & (\nu y)(\nu x)P & \text{S-PERM}
\end{array}
$$

Figure 1: Structural congruence

$$\frac{m = n}{x\,?\,[y_1,\ldots,y_m]\,.\,P \mid x\,!\,[z_1,\ldots,z_n]\,.\,Q \longrightarrow P\{z_1,\ldots,z_n/y_1,\ldots,y_m\} \mid Q}\ \text{R-COMM}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \text{R-PAR} \qquad \frac{P' \equiv P \quad P \longrightarrow Q \quad Q \equiv Q'}{P' \longrightarrow Q'}\ \text{R-CONG}$$

$$\frac{P \longrightarrow P'}{(\nu x)P \longrightarrow (\nu x)P'}\ \text{R-NEW}$$

Figure 2: Reduction

grammar specifies the standard untyped π-calculus, including some alternatives.

$$
\begin{array}{rcll}
P & ::= & x\,?\,[\tilde{y}]\,.\,P & \mid\ x\,?^{*}\,[\tilde{y}]\,.\,P \\
& \mid & x\,!\,[\tilde{y}]\,.\,P & \mid\ x\,!\,[\tilde{y}] \\
& \mid & (\nu\tilde{x})P & \mid\ \mathbf{0} \\
& \mid & P \mid Q & \mid\ !P
\end{array}
$$

In this grammar P and Q stand for processes, x for a name, and $\tilde{y}$ for a list of names. Asynchronous output, $x\,!\,[\tilde{y}]$ is an abbreviation for $x\,!\,[\tilde{y}]\,.\,\mathbf{0}$ but in some presentations it is the only form of output. Replicated input, $x\,?^{*}\,[\tilde{y}]\,.\,P$, is an abbreviation for $!(x\,?\,[\tilde{y}]\,.\,P)$, but in some presentations it is the only form of replication. These differences affect programming style and ease of implementation, but do not generally have a serious impact on typing rules. When types are introduced, it is usual for input prefixes $x\,?\,[\tilde{y}]\,.\,P$ and restrictions $(\nu\tilde{y})P$ to declare types for $\tilde{y}$, changing the syntax to $x\,?[\tilde{y}:\tilde{T}]\,.\,P$ and $(\nu\tilde{y}:\tilde{T})P$ where $\tilde{T}$ is s list of types. Further syntactic extensions will be introduced when polymorphism (Section 6) and session types (Section 7) are discussed.

Following Milner [17], most of the systems described in this article present the operational semantics of the π-calculus by means of a structural congruence relation, defined by the inference rules in Figure 1, and a reduction relation, defined by the inference rules in Figure 2.

1.2 Barbed Equivalences

A number of the papers summarised in this article study the effect of typing on process equivalence. The equivalences used are all variations on the notion of *barbed bisimulation*, introduced by Milner and Sangiorgi [20]. We will now present the definitions for untyped processes, and indicate later in the article how they are varied for each type system.

If a is a channel name, then process P *exhibits the barb* a, written $P\downarrow_a$, if P contains an input $a\,?\,[\ldots]$ or an output $a\,!\,[\ldots]$ which is not under another prefix or within the scope of a (νa).

Definition A relation $\mathcal{R}$ on processes is a *barbed bisimulation* if $(P,Q) \in \mathcal{R}$ implies the following:

1. if $P \longrightarrow P'$ then $\exists Q'.Q \longrightarrow Q'$ and $(P',Q') \in \mathcal{R}$

$$\frac{}{?\vdash \mathbf{0}}\ \textsc{T-Nil} \qquad \frac{?\vdash P \quad ?\vdash Q}{?\vdash P \mid Q}\ \textsc{T-Par}$$

$$\frac{?, x : T \vdash P}{?\vdash (\nu x : T)P}\ \textsc{T-New} \qquad \frac{?\vdash P}{?\vdash !P}\ \textsc{T-Rep}$$

$$\frac{?\vdash P \quad ?\vdash x : ![\tilde{T}] \quad ?\vdash \tilde{y} : \tilde{T}}{?\vdash x\,!\,[\tilde{y}]\,.\,P}\ \textsc{T-Out} \quad \frac{?, \tilde{y} : \tilde{T} \vdash P \quad ?\vdash x : ?[\tilde{T}]}{?\vdash x\,?\,[\tilde{y} : \tilde{T}]\,.\,P}\ \textsc{T-In}$$

Figure 3: Typing rules for the simple type system

2. if $Q \longrightarrow Q'$ then $\exists P'.P \longrightarrow P'$ and $(P', Q') \in \mathcal{R}$
3. for each channel a, $P \downarrow_a$ if and only if $Q \downarrow_a$.

P and Q are *barbed bisimilar*, written $P \dot{\sim} Q$, if there is a barbed bisimulation $\mathcal{R}$ with $(P, Q) \in \mathcal{R}$. P and Q are *barbed congruent*, written $P \sim Q$, if for all contexts (process expressions containing a hole) C, $C[P] \sim C[Q]$. In general, rather than considering arbitrary contexts, it is sufficient to close under parallel composition and substitution (of names for names).

Weak versions of these equivalences can also be defined. Let $\Longrightarrow$ be the reflexive and transitive closure of $\longrightarrow$, and let $\Downarrow_a$ be the composition $\Longrightarrow \downarrow_a$.

Definition A relation $\mathcal{R}$ on processes is a *weak barbed bisimulation* if $(P, Q) \in \mathcal{R}$ implies the following:

1. if $P \longrightarrow P'$ then $\exists Q'.Q \Longrightarrow Q'$ and $(P', Q') \in \mathcal{R}$
2. if $Q \longrightarrow Q'$ then $\exists P'.P \Longrightarrow P'$ and $(P', Q') \in \mathcal{R}$
3. for each channel a, $P \Downarrow_a$ if and only if $Q \Downarrow_a$.

Weak barbed bisimulation, $\dot{\approx}$, is defined analogously to barbed bisimulation. Weak barbed congruence, $\approx$, means weak barbed bisimulation in all contexts. A coarser relation, *weak barbed equivalence*, is obtained by closing weak barbed bisimulation under parallel composition but not substitution.

One of the attractive features of barbed equivalences is that they can be defined in a uniform way but tailored to an individual system by choosing an appropriate notion of barb. The various versions of typed π-calculus described in this article make use of this feature, and also require contexts to be typed in an environment compatible with the processes being compared.

2 Simple Types

A channel in the π-calculus is used to carry a message consisting of a tuple of channel names. In the simple type system, the type of a channel indicates the types of the channel names whch can be transmitted along it. Types are therefore defined by the grammar

$$T ::= {}^\wedge[\tilde{T}]$$

where $\tilde{T}$ represents a list $T_1, \ldots, T_n$ of types $(n \geqslant 0)$. The inference rules in Figure 3 define the well-typed processes: a process P is well-typed in an environment ? if $? \vdash P$ can appear as the root of a derivation tree constructed from these rules. Note that ? is a set of typed names of the form $x : T$, which can be viewed as a partial function from names to types. When $x \in dom(?)$ and $?(x) = T$, we sometimes write $? \vdash x : T$. Also note that the syntax has been extended to include type declarations for the names bound by input and ν. Rules T-In and T-Out enforce the restriction that channels should only be used in accordance with their declared types; the other rules ensure that all parts of a process are typed in the same environment.

Examples

1. The judgement
$$x : \hat{}[\hat{}[]], y : \hat{}[] \vdash x\,!\,[y]\,.\,\mathbf{0} \mid x\,?\,[z : \hat{}[]]\,.\,\mathbf{0}$$
is derivable.

2. The judgment
$$x : \hat{}[\hat{}[]], y : \hat{}[] \vdash x\,!\,[y, y]\,.\,\mathbf{0} \mid x\,?\,[z : \hat{}[]]\,.\,\mathbf{0}$$
is not derivable, as the output on x is incompatible with the declared type of x.

There is no separate definition of the reduction relation for typed processes; a process can be executed according to the rules in Figure 2 whether or not it is correctly typed. Execution of the process in Example 2 above fails because rule R-Comm does not apply; this represents a runtime error. The aim, of course, is to prove that executing a well-typed process never leads to communication errors of this kind. The key result is the following:

Theorem (Subject Reduction) If $? \vdash P$ and $P \longrightarrow Q$ then $? \vdash Q$.

This theorem is proved by induction on the derivation of the reduction $P \longrightarrow Q$, with an analysis of the last typing rule used in the derivation of $? \vdash P$. There are then two ways of deducing type safety from Subject Reduction.

The first technique is to introduce a process called *Wrong*, and add a reduction rule for the case of communication error:

$$\frac{m \neq n}{x\,?\,[y_1, \ldots, y_m]\,.\,P \mid x\,!\,[z_1, \ldots, z_n]\,.\,Q \longrightarrow \mathit{Wrong}}\ \text{R-CommWrong}$$

The typing rules are unchanged, so that *Wrong* can never be well-typed. Subject Reduction then immediately implies that "well-typed processes never go wrong" (to paraphrase Milner's [14] description of type-safety for a functional language). Part of the work of proving Subject Reduction can be seen as checking that rule R-CommWrong is never used when executing a well-typed process.

The alternative technique does not introduce *Wrong*, but instead proves

Theorem If $? \vdash P$ and $P \equiv (\nu\tilde{x} : \tilde{T})(a\,?[u_1 : U_1, \ldots, u_m : U_m]\,.\,Q_1 \mid a\,![v_1, \ldots, v_n]\,.\,Q_2 \mid Q_3)$ then $m = n$.

In other words, a well-typed process contains no immediate possibility of a communication error. This theorem follows from the fact that the name a has only been given a type once, either in $?$ or in $\nu\tilde{x} : \tilde{T}$, and both the input and the output on a must respect this type.

The first approach is straightforward and intuitive, but requires the calculus to be modified (by the addition of *Wrong*). The second approach appears to be more flexible when the property to be established of well-typed processes is more complex, and is used in most of the type systems described in this article.

Notice that the grammar does not define any ground (or base) types. By taking $n = 0$, the type $\hat{}[]$ describes a channel of pure synchronisations (a CCS action, in effect) and plays the role of a ground type. Alternatively, it is straightforward to add ground types such as int or bool, along with constructors and destructors for expressions of ground type. Several of the type systems described in this article do add at least bool.

Another useful addition is recursive types. This requires a notion of equality between recursive type expressions, in order to support replacing a recursive type by its unwinding.

The simple type system was presented by Milner [17] under the name of *sorts*. His presentation was somewhat different, representing an environment by a function from a set of sorts to tuples of sorts, and implicitly supporting recursive sorts. Pierce and Sangiorgi [23, 24] were perhaps the first to use a presentation based on environments, for their system of input/output subtyping (see Section 3), and all subsequent type systems have been presented in this style.

Turner [31] gives an environment presentation of the simple type system, and also considers recursive types explicitly; his treatment of recursive types is similar to Pierce and Sangiorgi's.

The problem of type reconstruction has been studied by Gay [3] and Vasconcelos and Honda [33].

3 Input/Output Types

Pierce and Sangiorgi [23, 24] proposed a refinement of the simple type system, in which the type of a channel not only specifies the types of channels which may be transmitted along it, but also the direction in which messages may flow. Each channel is declared to be usable for input only, output only or both input and output, and therefore the grammar defining types is

$$T ::= \hat{}[\tilde{T}] \mid ?[\tilde{T}] \mid ![\tilde{T}].$$

This system also incorporates a notion of subtyping, based on the idea that a channel whose declared type permits both input and output can be used wherever a channel of input-only or output-only type is required. The subtyping relation is defined by the following inference rules. (Subtyping for lists of types, $\tilde{T} \leqslant \tilde{U}$, is defined pointwise.)

$$\frac{\Sigma \vdash \tilde{T} \leqslant \tilde{U}}{\Sigma \vdash ?[\tilde{T}] \leqslant ?[\tilde{U}] \quad \Sigma \vdash \hat{}[\tilde{T}] \leqslant ?[\tilde{U}]}$$

$$\frac{\Sigma \vdash \tilde{U} \leqslant \tilde{T}}{\Sigma \vdash ![\tilde{T}] \leqslant ![\tilde{U}] \quad \Sigma \vdash \hat{}[\tilde{T}] \leqslant ![\tilde{U}]}$$

$$\frac{\Sigma \vdash \tilde{T} \leqslant \tilde{U} \text{ and } \Sigma \vdash \tilde{U} \leqslant \tilde{T}}{\Sigma \vdash \hat{}[\tilde{T}] \leqslant \hat{}[\tilde{U}]}$$

Note that the three channel type constructors have different variance: ? is covariant, ! is contravariant, and ^ is invariant. This behaviour can be justified by supposing that the data types int and real exist and that int $\leqslant$ real. We have ?[int] $\leqslant$?[real] because if a process requires a channel of type ?[real] on which it can receive a real number, then it is safe to use a channel of type ?[int] instead; the real number which is received will always be an integer. Conversely, ![real] $\leqslant$![int] because if a channel is required along which an integer can be sent, it is safe to use a channel of type ![real] instead. Invariance of ^ follows from a combination of these arguments.

The typing rules are similar to those of the simple type system, with two main differences: the rules T-In and T-Out require channels to be of input or output type, respectively, and the type of a channel as declared in the environment only has to be a subtype of the type required by the process being constructed. Hence T-Out and T-In become:

$$\frac{? \vdash P \quad ?(x) \leqslant ![\tilde{T}] \quad ? \vdash \tilde{y} : \tilde{T}}{? \vdash x\,!\,[\tilde{y}]\,.\,P}\text{T-Out} \qquad \frac{?, \tilde{y} : \tilde{T} \vdash P \quad ?(x) \leqslant ?[\tilde{T}]}{? \vdash x\,?\,[\tilde{y} : \tilde{T}]\,.\,P}\text{T-In}$$

and the other rules remain the same.

Recursive types are also defined. The subtype relation is defined by an algorithm based on the inference rules above, and also (following Amadio and Cardelli [1, 2]) as a coinductively-defined simulation relation on trees representing type expressions. Recursive types are unwound into infinite trees (Turner [31] gives an alternative definition in which recursive types are unwound on demand). The algorithm is proved sound and complete with respect to the coinductive definition, and can be used in an implementation of a type-checker; the coinductive definition more easily supports proofs of basic properties such as transitivity.

The motivation for the introduction of input/output types is twofold. From a programming point of view, the ability to classify channels more accurately means that the environment in which a process has been typed is more informative. The Pict programming language (see

Section 9) uses this type system as its starting point. Secondly, the refined type system is useful when reasoning about processes. Specifically, Pierce and Sangiorgi consider Milner's [16] encoding of the call-by-value λ-calculus into the π-calculus. While one would like to prove that the encoding of a λ-term and the encoding of its β-reduction are equivalent in a suitable sense (perhaps weak barbed congruence) this turns out not to be true. However, the failure is essentially because a process context can "misuse" certain channels by sending messages along them in a direction not intended by the encoding. When input/output types are introduced, the definition of wek barbed congruence is also modified: the context in which two processes are compared is required to be typed in the same environment as the processes themselves. Because fewer contexts are available, it becomes easier for processes to be equivalent, and the desired property of β-reduction can be proved.

Proving type safety is less straightforward than in the simple type system. During reduction, the type of a channel may be promoted (for example from input/output to input-only) and although it is still true that the type declared in the environment is a subtype of the promoted type associated with a particular occurrence of a channel, the environment no longer contains enough information to ensure that the channel remains input-only during the subsequent execution. Therefore an alternative syntax of *tagged processes* is defined, in which every occurrence of a name is annotated with its actual type. A subject reduction theorem can then be proved for tagged processes, and because every typed but untagged process can be tagged in a canonical way, type safety for the original language follows. The reduction rules for tagged processes make use of a *Wrong* process, which arises when a channel is used incorrectly for communication (i.e. in conflict with its tag) or, as in the simple type system, when a communication takes place with an arity mismatch.

4 Linear Types

The ideas of linear logic [6] have been applied to a number of programming paradigms and it has been amply demonstrated that linear type information—classifying values according to whether they are used once or more than once—supports various compiler optimisations. Kobayashi *et al.* [11] have developed a linear type system for the π-calculus, partly in order to allow compiler optimisations (a channel which can only be used once can be deallocated after use) and partly so that, as with the input/output subtyping system, more detailed type information can assist with reasoning about processes.

The basic idea is that a linear channel can only be used for one communication, but a little thought shows that the name of such a channel must appear *twice*—once in an input position and once in an output position. It is therefore necessary to control input usage and output usage separately, and this gives the typing rules some of the flavour of the input/output system described in Section 3. Because channels can be sent from one process to another, this control is slightly more complicated than it might appear; the input use of a linear channel might only occur after that channel has been received as a message, and so its name might never appear as the subject of an input prefix.

A channel type is associated with a *multiplicity* m, which is either 1 (for linear use) or ω (for non-linear use), and a *polarity* p, which indicates the input or output use permitted. A polarity is a subset of $\{i, o\}$, and the abbreviations $\updownarrow = \{i, o\}$, $? = \{i\}$, $! = \{o\}$, $| = \{\}$ are used. Booleans are also introduced, so that the grammar of types is

$$T ::= p^m[\tilde{T}] \mid \mathsf{bool}.$$

In a typing judgement $? \vdash P$, the occurrence of $x : p^1[\tilde{T}]$ in $?$ indicates that all of the input or output capabilities of x, as declared in p, *must* be used in P. Therefore the typing rule for parallel is

$$\frac{?_1 \vdash P \quad ?_2 \vdash Q}{?_1 + ?_2 \vdash P \mid Q} \text{ T-PAR}$$

where the partial operation + combines usages: it is defined on polarities by

$$\begin{array}{rcl} p^\omega + q^\omega & = & (p \cup q)^\omega \\ p^1 + q^1 & = & (p \cup q)^1 \quad \text{if } p \cap q = \emptyset \end{array}$$

and extended to environments with equal domains by

$$(?_1, x : p^m[\tilde{T}]) + (?_2, x : q^n[\tilde{T}]) = (?_1 + ?_2), x : (p^m + q^m)[\tilde{T}].$$

The + operation is also used to add a single capability to a channel in an environment:

$$(?, x : p^m[\tilde{T}]) + x : q^n[\tilde{T}] = ?, x : (p^m + q^m)[\tilde{T}].$$

For example,

$$\frac{x : !^1[\mathsf{bool}] \vdash x \,!\, \mathit{true} \quad x : ?^1[\mathsf{bool}] \vdash x \,?\, [y : \mathsf{bool}] \,.\, P}{x : \updownarrow^1 [\mathsf{bool}] \vdash x \,!\, \mathit{true} \mid x \,?\, [y : \mathsf{bool}] \,.\, P}$$

is a valid instance of T-Par. If + appears in a typing rule, then that rule can only be applied if + is actually defined in the case required.

A process can use a particular capability of a channel either by performing an input or an output on that channel or by passing the channel to another process, which then becomes responsible for using the capability. The typing rule for output takes this into account.

$$\frac{? \vdash P}{? + x : !^m[\tilde{T}] + \tilde{y} : \tilde{T} \vdash x \,!\, [\tilde{y}]P} \text{ T-Out}$$

For example, a process which sends a linear channel along a linear channel can be typed as follows. An *unlimited* environment is one in which all the linear channels have polarity $|$.

$$\frac{? = x : |^1 \, [!^1[\mathsf{bool}]], y : |^1 \, [\mathsf{bool}] \text{ unlimited}}{? + x : !^1[!^1[\mathsf{bool}]] + y : !^1[\mathsf{bool}] \vdash x \,!\, [y]} \text{ T-Out}$$

The final typing judgement is

$$x : !^1[!^1[\mathsf{bool}]], y : !^1[\mathsf{bool}] \vdash x \,!\, [y].$$

The process $x\,!\,[y]$ uses the output capability of x itself, and delegates the responsibility for using the output capability of y to whichever process receives it. (Note that the choice of ! for the polarity of y is arbitrary at this point.)

A subject reduction theorem is proved, and it is then noted that a well-typed process contains no immediate possibility of communication failure or misuse of capabilities. Two results are then established which enable linearity to assist with reasoning about process behaviour. Firstly, communication on a linear channel is deterministic: if $P \longrightarrow Q$ and $P \longrightarrow R$ and both reductions use the same linear channel then $Q \equiv R$. Secondly, reduction on a linear channel is confluent with respect to all other reductions: if a process contains two possible communications, one of which is on a linear channel, then they can be carried out in either order and the same result will be achieved. A series of process equivalences are defined, leading to weak barbed congruence (for a suitable notion of barb—it is necessary to consider whether or not the channel a in $\downarrow_a$ has any unused capabilities in an appropriate environment). The main result which expolits linearity is that reduction on a linear channel preserves the weak congruence class of a process.

Igarashi and Kobayashi [8] have added subtyping to the linear type system, and have considered the problem of type reconstruction. The subtype relation includes both input/output subtyping, as defined by Pierce and Sangiorgi, and an ordering on multiplicities. Multiplicities are recorded separately for input and output usage (so that more refined classifcations are possible) and, for example, a channel of type $[\mathsf{bool}]^{(\omega,\omega)}$, which can be used any number of times to input or output a boolean, can be promoted to a channel of type $[\mathsf{bool}]^{(1,\omega)}$, which can be used any number of times for output but only once for input.

The goal of type reconstruction is to find an environment in which a given process can be typed, and ideally one would like to find the principal typing, or most general environment from which all others can be obtained by instantiating type variables. It is shown that principal typings do not exist for this system. Instead, an algorithm is presented which constructs an environment (containing type variables) and a system of constraints; solving the constraints then yields all possible environments which permit the original process to be typed.

5 Types for Partial Deadlock-Freedom

In the linear type system of Kobayashi *et al.* (Section 4), reduction on linear channels is deterministic and confluent, but less is guaranteed about reduction on non-linear channels. Kobayashi [9, 10] has extended the idea of guaranteeing desirable properties of reduction involving only a certain class of channels. In this system, channels are classified as *reliable* or *unreliable*, and it is guaranteed that a process which uses only reliable channels will never deadlock. Furthermore, if a process which uses both reliable and unreliable channels does deadlock, then the deadlock can only be caused by an attempt to communicate on an unreliable channel.

Two kinds of deadlock are identified: cyclic dependencies in the order of channel use, and the possibility that a receive or send operation can be blocked because no corresponding sender or receiver becomes available. The first case is illustrated by the process

$$x\,?\,[\,]\,.\,y\,!\,[\,] \mid y\,?\,[\,]\,.\,x\,!\,[\,]$$

and the second by

$$(\nu x : \hat{}[\,])x\,?\,[\,]\,.\,y\,!\,[\,] \mid y\,?\,[\,]\,.\,P$$

The reliable channels are divided into three classes: *linear* channels, *mutex* channels, and *replicated* channels. Linear channels behave in the same way as that described in Section 4. Mutex channels behave rather like storage cells which can only be accessed under mutual exclusion: if a process reads from a mutex channel it must write to it at some later time. Replicated channels are used for replicated input. A replicated input is treated as a persistent resource which may or may not be used, and if a process is blocked waiting for input on a replicated channel then it is not considered to be deadlocked.

In order to avoid the construction of processes which contain cyclic dependencies, every environment contains a partial order on the linear and mutex channels which it declares. In the first example above, the two subprocesses would be typed as

$$x : \hat{}[\,], y : \hat{}[\,], x < y \vdash x\,?\,[\,]\,.\,y\,!\,[\,]$$

$$x : \hat{}[\,], y : \hat{}[\,], y < x \vdash y\,?\,[\,]\,.\,x\,!\,[\,]$$

The typing rule for parallel composition checks that the partial orders can be combined without introducing cycles, and therefore these two processes cannot be put in parallel. The way in which the type system rejects the second example is more complex, but essentially the typing rules prohibit x from being linear, mutex or replicated, and so the process cannot be typed using only reliable channels. As usual, the Subject Reduction theorem means that such situations cannot arise during execution either. The fact that channels can be transmitted between processes makes things more complex: when a channel is sent as a message, the typing rules must ensure that the partial orders of the sending and receiving processes are compatible.

As a demonstration that a useful range of processes can be guaranteed deadlock-free in this way, it is shown that both the λ-calculus (with various reduction strategies) and concurrent objects [26] can be encoded using only reliable channels.

6 Polymorphism

When the π-calculus is viewed as a programming language, polymorphism is an obvious feature to consider adding. Turner [31] has studied a polymorphic type system for the π-calculus, and polymorphism has been included in Pict [27]. An example of a process which ought to be polymorphic in the type T is

$$\mathsf{Id} = i\,?\,[x : T, r : \hat{}[T]]\,.\,r\,!\,[x]$$

If a client of Id sends a fresh channel to Id and waits for the result to come back:

$$\mathsf{IdClient} = (\nu r : \hat{}[T])i\,!\,[x, r]\,.\,r\,?\,[y : T]\,.\,\ldots$$

then we can view Id as the identity function.

The definitions in this example contain the particular type T explicitly, and x must be of type T in order for $\mathsf{Id} \mid \mathsf{IdClient}$ to be typable. In Turner's polymorphic type system, a message is a package consisting of a tuple (possibly empty) of types and a tuple of values, and input prefixes can contain type variables. The example above can be rewritten as

$$\mathsf{PolyId} = i\,?\,\{X\}[x : X, r : \hat{}[X]]\,.\,r\,!\,[x]$$

$$\mathsf{PolyIdClient} = (\nu r : \hat{}[T])i\,!\,\{T\}[x, r]\,.\,r\,?\,[y : T]\,.\,\ldots$$

and the type of i is $\hat{}\{X\}[X, \hat{}[X]]$. It is straightforward to formalise typing rules and the reduction relation for this system, and to prove Subject Reduction.

In the work of Gay [3] and Vasconcelos and Honda [33] on type reconstruction for the simple type system, the result of type reconstruction is a typing of a process in an environment containing type variables which can be instantiated once and for all to yield a valid monomorphic typing judgement. Turner's system is different in that instantiation of type variables is internalised and can take place at runtime; the difference is similar to that between the polymorphism of Standard ML [21] and the polymorphic λ-calculus [5, 29]. Liu and Walker [12] have considered a rather different system of polymorphism, which we will not discuss further as it does not fit into the common presentational style of this article.

Pierce and Sangiorgi [25] (from whose paper the above example is taken) have considered the effect of Turner's system of polymorphism on process equivalence, in particular weak barbed equivalence. They show that polymorphism can be used to hide the implementation details of an abstract data type (ADT), in a similar way to that described by Mitchell and Plotkin [22] for the polymorphic λ-calculus. This means that different implementations of an ADT, which differ radically in the way in which a value interacts with its users, can nevertheless be proved equivalent.

7 Session Types

Except for the polymorphic type system, all of the systems described so far have the property that the type of a channel specifies the type of a single message, and although it might be possible to use a channel more than once, every message transmitted must have the same type. If one process is to send an integer to another process and then receive a boolean, then it is necessary either to use two channels:

$$x : \hat{}[\mathsf{int}], y : \hat{}[\mathsf{bool}] \vdash x\,!\,[2]\,.\,y\,?\,[a : \mathsf{bool}]\,.\,P \mid x\,?\,[a : \mathsf{int}]\,.\,y\,!\,[\mathit{true}]\,.\,Q$$

or to create a fresh channel for the second message and transmit that channel as part of the first message:

$$x : \hat{}[\mathsf{int}, \hat{}[\mathsf{bool}]] \vdash (\nu y : \hat{}[\mathsf{bool}])x\,!\,[2, y]\,.\,y\,?\,[a : \mathsf{bool}]\,.\,P \mid x\,?\,[a : \mathsf{int}, y : \hat{}[\mathsf{bool}]]\,.\,y\,!\,[\mathit{true}]\,.\,Q.$$

In the first case, the types declared in the environment are not as informative as we might wish: there is no indication that both x and y are used to carry parts of what is, logically, a single dialogue. The second case is an improvement, but the nesting of type constructors in the type of x becomes deeper as longer sequences of messages are encoded, and the need to create a fresh continuation channel for each message obscures the definition of the process. Adding polymorphism does not help because it introduces too much flexibility: a polymorphic channel can carry messages of *any* type, and there is no way to specify that the first message must be an integer and the second a boolean.

Honda *et al.* [7, 30] have proposed a type system which introduces the idea of *session channels*. In this system, the example above can be typed as

$$x : ![\mathsf{int}] \, . \, ?[\mathsf{bool}] \, . \, \mathsf{end}^2 \vdash x \, ! \, [2] \, . \, y \, ? \, [a : \mathsf{bool}] \, . \, P \mid x \, ? \, [a : \mathsf{int}] \, . \, y \, ! \, [\mathit{true}] \, . \, Q$$

if we assume that P and Q do not use x. (Honda *et al.* use a rather different syntax, but here we are aiming for consistency with our presentation of the other type systems.)

Session channels must be treated linearly: the two ends must be distinguished, and each end must only be used by one process. Because messages can be sent in both directions there is no longer an input end and an output end, but rather two complementary ends. The two subprocesses of the example above are typed as

$$x : ![\mathsf{int}] \, . \, ?[\mathsf{bool}] \, . \, \mathsf{end} \vdash x \, ! \, [2] \, . \, y \, ? \, [a : \mathsf{bool}] \, . \, P$$

$$x : ?[\mathsf{int}] \, . \, ![\mathsf{bool}] \, . \, \mathsf{end} \vdash x \, ? \, [a : \mathsf{int}] \, . \, y \, ! \, [\mathit{true}] \, . \, Q$$

and the typing rule for parallel composition must check that the two types of x are complementary in the evident sense, and combine them into the fully used (indicated by the superscript of 2) session type $![\mathsf{int}] \, . \, ?[\mathsf{bool}] \, . \, \mathsf{end}^2$.

Originally [30], linear use of session channels was guaranteed syntactically; session channels formed a separate syntactic category and could not be transmitted between processes. Later [7] the system was generalised to allow session channels to be transmitted, subject to controls similar to those in the linear type system of Kobayashi *et al.*. It is possible for a session channel to be given to another process after part of the communication sequence has been carried out; for example, a process which is engaged in a session can execute part of the session itself and then pass the channel to another process, delegating the responsibility for completing the session.

A session type can also allow branching. The type $\&\langle l : T, m : U\rangle$ describes a channel which can receive a label, either l or m, and then behave as a channel of type T or U respectively. The complementary type is $\oplus\langle l : \overline{T}, m : \overline{U}\rangle$ (where $\overline{T}$ denotes the complement of T), indicating that either l or m can be sent in order to make a choice. In order to use branching types, the syntax of the π-calculus must be extended with operations for sending and receiving labels. The process $x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$ receives a label (which should be one of the l_i) on channel x and then behaves as the appropriate P_i. The process $x \triangleleft l \, . \, P$ sends the label l along channel x and then behaves as P. The definition of reduction for processes involving choice is

$$\frac{i \in \{1, \ldots, n\}}{x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \mid x \triangleleft l_i \, . \, Q \longrightarrow P_i \mid Q} \ \textsc{R-Select}$$

The case in which the label being chosen is not one of the labels being offered is an error. The type system guarantees avoidance of such errors, as well as the usual communication mismatches, and for this purpose the following typing rules are introduced.

$$\frac{?, x : S_i \vdash P}{?, x : \oplus\langle l_1 : S_1, \ldots, l_n : S_n\rangle \vdash x \triangleleft l_i \, . \, P} \ \textsc{T-Choose}$$

$$\frac{?, x : S_1 \vdash P_1 \ldots ?, x : S_n \vdash P_n}{?, x : \&\langle l_1 : S_1, \ldots, l_m : S_m\rangle \vdash x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}} \ \textsc{T-Offer}$$

One of the motivating examples of the use of session types is client-server systems. A typical client-server protocol specifies a sequence of message, which can be described by a session type. The server offers a choice of services, from which the client can select, using the operations described above. Dually, there may be times when the server can send one of a range of responses to the client; this can also be represented by means of the same branching type constructors. Recursive types are also useful in order to describe protocols which can run through an arbitrary number of cycles, and can easily be added.

8 Session Types with Subtyping

A natural extension to session types is that addition of subtyping, and this has been considered by Gay and Hole [4]. For non-session types, subtyping is defined in the same way as in Pierce and Sangiorgi's system (Section 3). Subtyping for session types arises in two ways. When branching is not involved, the subtype relation is the pointwise extension along sequences of the Pierce and Sangiorgi subtype relation. Note, however, that because the input/output constructor does not appear in session types (because the direction of a message must be unambiguously specified), only the covariant action of input and the contravariant action of output are involved. For branch and choice types, subtyping is more interesting. If a process needs a channel of type $\&\langle l_1 : T_1, \ldots, l_n : T_n \rangle$ which allows it to offer a choice from $\{l_1, \ldots, l_n\}$, then it can safely use a channel of type $\&\langle l_1 : T_1, \ldots, l_m : T_m \rangle$, where $m \leqslant n$, instead. The channel type prevents the process from ever receiving labels $l_{m+1}, \ldots, l_n$ but every label that can be received will be understood. Furthermore, a channel of type $\&\langle l_1 : S_1, \ldots, l_m : S_m \rangle$ can be used if each $S_i \leqslant T_i$, as this means that after the choice has been made the continuation process uses a channel of type S_i instead of T_i and this is safe.

For example, a server offering equality testing for integers might interact with a client over a channel of type S, defined (recursively) by

$$S = \&\langle \mathsf{equal} : ?[\mathsf{int}] \,.\, ?[\mathsf{int}] \,.\, ![\mathsf{bool}] \,.\, S \rangle.$$

In order to start a dialogue with the server, a client process would create a fresh channel of type S and send it to the server along a channel of type $\hat{}[S]$. (The type S does not allow the dialogue to be terminated, but this problem can be solved by adding the option $\mathsf{stop} : \mathsf{end}$ to the choice.)

If the server is upgraded to offer equality testing for real numbers, and also a negation service on integers, the type of channel which it expects to use becomes T, defined by

$$T = \&\langle \mathsf{equal} : ?[\mathsf{real}] \,.\, ?[\mathsf{real}] \,.\, ![\mathsf{bool}] \,.\, T, \mathsf{negate} : ?[\mathsf{int}] \,.\, ![\mathsf{int}] \,.\, T \rangle.$$

Because $S \leqslant T$ (for the reasons outlined earlier), the old client can interact with the upgraded server in the same way as before, by sending it a channel of type S. No runtime errors will result, but the full capabilities of the server will not be used.

Type safety for this system is proved in the same way as for Pierce and Sangiorgi's system of input/output subtyping, by introducing a tagged syntax.

9 Pict's Type System

Pict [27] is a concurrent programming language based on the π-calculus, which has been developed by Pierce and Turner. The core of Pict is an implementation of the asynchronous π-calculus, and various higher-level features, such as function definitions and calls, are built on top of the core. One of the original goals of the Pict project was to explore programming in the π-calculus, and to discover which high-level programming idioms are most useful in practice. Because types and type-checking are so important for realistic programming, Pict is a typed language, and has become a vehicle for experimenting not only with type systems but also with type checking and type inference algorithms [28]. Pict starts with Pierce and Sangiorgi's input/output types with subtyping, and adds record types and some basic data types such as int and bool. Polymorphism has been included, and indeed extended to a system of higher-order polymorphism. The linear types of Kobayashi *et al.* have been added as an experimental extension (although not included in a release) and perhaps the more recent work of Igarashi and Kobayashi on type reconstruction for linear types with subtyping can be used to reduce the amount of linear type information which the programmer needs to provide. The present author hopes to implement session types with subtyping as an extension to Pict in the near future.

10 Conclusions

In this article we have surveyed a certain progression of research on types for the π-calculus, concentrating on type systems which classify channels in ways which are useful for programming. We have not covered any of the recently-developed type systems which describe distribution or security. Nor have we discussed other frameworks for types for concurrent systems, in particular Nierstrasz' *regular types* and their subsequent development by Puntigam. Nevertheless, we hope that the present survey will provide a useful introduction to the range of type-theoretic technology which is available for programming in the π-calculus.

Acknowledgements

Thanks are due to the EPSRC for funding the project "Novel Type Systems for Concurrent Programming Languages" (GR/L75177), to the Nuffield Foundation for a research grant, and to the organisers of the EPSRC/LMS Mathfit meeting on Semantics and Types for Concurrency for inviting me to give the lectures on which this article is based.

References

[1] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proceedings, 18th ACM Symposium on Principles of Programming Languages*. ACM Press, 1991.

[2] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[3] S. J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proceedings, 20th ACM Symposium on Principles of Programming Languages*. ACM Press, 1993.

[4] S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. In *ESOP'99: Proceedings of the European Symposium on Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[5] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, University of Paris VII, 1972.

[6] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[7] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the European Symposium on Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[8] A. Igarashi and N. Kobayashi. Type-based analysis of communication for concurrent programming languages. In *SAS'97: Proceedings of the International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, 1997.

[9] N. Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1997.

[10] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20:436–482, 1998.

[11] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *Proceedings, 23rd ACM Symposium on Principles of Programming Languages*, 1996.

[12] X. Liu and D. Walker. A polymorphic type system for the polyadic π-calculus. In *CONCUR'95: Proceedings of the International Conference on Concurrency Theory*, volume 962 of *LNCS*. Springer-Verlag, 1995.

[13] I. Mackie. Lilac : A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):1–39, October 1994.

[14] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

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

[16] R. Milner. Functions as processes. In *Proceedings of ICALP 90*, volume 443 of *Lecture Notes in Computer Science*, pages 167–180. Springer-Verlag, 1990.

[17] R. Milner. The polyadic π-calculus: A tutorial. Technical Report 91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

[18] R. Milner. *Communicating and Mobile Systems.* Cambridge University Press, 1999.

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

[20] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings of ICALP*, 1992.

[21] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[22] J. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), 1988.

[23] B. C. Pierce and D. Sangiorgi. Types and subtypes for mobile processes. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1993.

[24] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.

[25] B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *Proceedings, 24th ACM Symposium on Principles of Programming Languages*, 1997.

[26] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In T. Ito and A. Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer-Verlag, April 1995.

[27] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1998.

[28] B. C. Pierce and D. N. Turner. Local type inference. In *Proceedings, 25th ACM Symposium on Principles of Programming Languages*, 1998.

[29] J. C. Reynolds. Towards a theory of type structure. In *Paris colloquium on programming*, volume 19 of *LNCS*. Springer-Verlag, 1974.

[30] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of the 6th European Conference on Parallel Languages and Architectures*, number 817 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[31] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1996.

[32] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *7th International Conference on Functional Programming and Computer Architecture*, 1995. Also technical report TR-1995-8, Computing Science Department, University of Glasgow.

[33] V. T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic π-calculus. In *CONCUR'93: Proceedings of the International Conference on Concurrency Theory*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[34] D. Walker. Objects in the pi-calculus. *Information and Computation*, 116:253–271, 1995.