Type Systems for Concurrent Calculi

Benjamin C. Pierce

University of Pennsylvania

Applications of Static Type Systems

- Run-time safety guarantees
  - Functions are always applied to arguments of the shapes they expect

- Protection of programmer-defined abstractions
  - Data abstraction / information hiding
- Powerful proof methods for program equivalences
  - Logical relations / parametricity

in Concurrent Languages

* Senders only transmit messages of shapes expected by receivers

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

* Generalized replicator theorems, partial confluence

Topics

We'll follow a thread of type systems for message-based concurrent languages.

- Untyped core language
- Simple channel types
- Input/output modalities
- Linear typing and variations
- Deadlock freedom
- Regular types
- Polymorphism

Non-topics

There is a *lot* of work in type systems for concurrent language that we won't have time to cover...

- Locality analyses for distributed systems [Sewell, Hennessy & Riely, etc.]
- Security type systems [Abadi, etc.]
- Semantic approaches (e.g. Interaction Categories [Abramsky], etc.)
- Type inference issues [Gay, Fournet/Laneve/Maranget/Rémy, etc.]
- Etc.

Core Language

Untyped Pi-Calculus

`P`	$::=$	`x![y`$_1$`...y`$_n$`]`	output (message)
		`x?[y`$_1$`...y`$_n$`]. P`	input
		`x?*[y`$_1$`...y`$_n$`]. P`	replicated input
		`P	Q`	parallel composition
		`(new x) P`	channel creation

Formally: an asynchronous, choice-free fragment of Milner, Parrow, and Walker's polyadic pi-calculus. [Cf. Honda and Tokoro's nu-calculus, Pict core language].

(Any similar language will do.)

Examples

Communication:

$$\texttt{(x![y] | x?[z]. z![x])} \;\longrightarrow\; \texttt{y![x]}$$

Replicated communication:

$$\begin{array}{ll} & \texttt{(x?*[z]. y![z]) | x![a] | x![b] | x![c]} \\ \longrightarrow & \texttt{(x?*[z]. y![z]) | y![a] | x![b] | x![c]} \\ \longrightarrow & \texttt{(x?*[z]. y![z]) | y![a] | x![b] | y![c]} \\ \longrightarrow & \texttt{(x?*[z]. y![z]) | y![a] | y![b] | y![c]} \end{array}$$

More Examples

"Booleans":

```
(new t) (new f)
      t?*[x y]. x![]
    | f?*[x y]. y![]
    | ...
```

Boolean tester:

```
...
 (new p) (new q)
     somebool![p q]
   | p?[]. P
   | q?[]. Q
```

where `somebool` behaves like `t` or `f`.

Last Example: Functions as Processes

A “factorial server” process:

```
(new fact)
   fact?*[n r].
        if n=0 then r![1]
        else (new r') ( fact![(n-1) r']
                      | r'?[m]. r![n*m] )
 | ...
```

For brevity, the example assumes syntax for conditionals and primitive arithmetic. This can all be desugared into processes in a similar way.

Observational Congruence

$\mathtt{P}$ and $\mathtt{Q}$ are congruent if $C[\mathtt{P}]$ behaves the same as $C[\mathtt{Q}]$ for each observing context $C[\,]$.

We write $\mathtt{P} \approx \mathtt{Q}$ in this case.

Congruence can be formalized in various ways (bisimulation, etc.).

Observational Congruence: Examples

$$\texttt{((new x) x![}y_1\texttt{...}y_n\texttt{]) | P}$$

$$\approx$$

$$\texttt{P}$$

$$\texttt{x?*[}y_1\texttt{...}y_n\texttt{].P}$$

$$\approx$$

$$\texttt{x?*[}y_1\texttt{...}y_n\texttt{].P | x?*[}y_1\texttt{...}y_n\texttt{].P}$$

Simple Types

Milner '91

(cf. simply typed lambda-calculus)

Channel Types

All the processes we have seen obey a simple type discipline in their use of channels. This observation is formalized by associating each channel with a channel type:

$$\mathtt{x} \in \mathtt{\hat{}[T_1 \ldots T_n]}$$

means "channel $\mathtt{x}$ carries tuples of channels of types $\mathtt{T}_1$ to $\mathtt{T}_n$."

$\mathtt{P}$ is well typed with respect to some set Γ of typing assumptions for its free variables

$$\Gamma \vdash \mathtt{P}$$

if its inputs and outputs are consistent with Γ.

N.b.: Entirely static analysis — each channel *always* carries tuples of the same shape. No information about timing or sequencing.

Function Types

The lambda-calculus type $S \rightarrow T$ of “functions from S to T” is encoded as `^[`S` ^[T]]`.

Input/Output Modalities

Pierce and Sangiorgi '93

Two Facts

Straightforward factorial function:

```
f1 = fact?*[n:Int r:^[Int]].
        if n=0 then r![1]
        else (new r')  (fact![(n-1) r']
                       | r'?[m]. r![n*m])
```

A more efficient version:

```
f2 = fact?*[n:Int r:^[Int]].
        if      n=0 then r![1]
        else if n=1 then r![1]
        else if n=2 then r![2]
        else if n=3 then r![6]
        else (new r') ( fact![(n-1) r']
                      | r'?[m]. r![n*m])
```

Too many observers $\longrightarrow$ too few congruences

Unfortunately, `f1` and `f2` are *not* congruent as processes. For example, consider:

$$\mathrm{C}[\,] = [\,] \texttt{ | fact![1 r] | (fact?[a b]. ding![])}$$

Then...

$$
\begin{array}{ll}
 & \mathrm{C[f1]} \\
\longrightarrow & \texttt{f1} \\
 & \texttt{| (if n=0 then r![1] else (new r}' \texttt{) (fact![(n-1) r}' \texttt{] | ...)} \\
 & \texttt{| (fact?[a b]. ding![])} \\
\longrightarrow & \texttt{f1} \\
 & \texttt{| ((new r}' \texttt{) (fact![(n-1) r}' \texttt{] | ...)} \\
 & \texttt{| (fact?[a b]. ding![])} \\
\longrightarrow & \texttt{(new r}' \texttt{)} \\
 & \texttt{f1} \\
 & \texttt{| (fact![(n-1) r}' \texttt{] | ...)} \\
 & \texttt{| (fact?[a b]. ding![])} \\
\longrightarrow & \texttt{(new r}' \texttt{) f1 | ding![]}
\end{array}
$$

But $\mathrm{C[f2]} \not\longrightarrow^{*}$ `... | ding![]`

Input/Output modalities

Solution: distinguish input from output capabilities:

$\mathtt{T}$	$::=$	$\mathtt{\hat{}[T_1 \ ... \ T_n]}$	both capabilities
		$\mathtt{![T_1 \ ... \ T_n]}$	output only
		$\mathtt{?[T_1 \ ... \ T_n]}$	input only

- $\mathtt{x![y]}$ is allowed only if $\Gamma(\mathtt{x}) = \mathtt{\hat{}[T]}$ or $\Gamma(\mathtt{x}) = \mathtt{![T]}$.
- $\mathtt{x?[y].P}$ is allowed only if $\Gamma(\mathtt{x}) = \mathtt{\hat{}[T]}$ or $\Gamma(\mathtt{x}) = \mathtt{?[T]}$.

The type $S \rightarrow T$ is now coded more precisely as `![`S `![T]]`.

Protecting a channel's "input end"

Want to give the observer the channel `fact` with type `![Int ![Int]]`.

Begin with a global channel

$$\texttt{getFact} \in \texttt{\^{}[![Int ![Int]]]}$$

Now write

```
f1-protected = (new fact:^[Int ![Int]])
                    getFact![fact]
                  | f1
```

and

```
f2-protected = (new fact:^[Int ![Int]])
                    getFact![fact]
                  | f2
```

Typed Observational Congruence

Observer must “play by the rules”:

Suppose $\Gamma \vdash \mathtt{P}$ and $\Gamma \vdash \mathtt{Q}$.

Call $C[\,]$ a Γ-context if the hole in C supplies all the capabilities demanded by Γ.

$\mathtt{P}$ and $\mathtt{Q}$ are congruent if $C[\mathtt{P}]$ behaves the same as $C[\mathtt{Q}]$ for each for each Γ-context $C[\,]$.

- Now `f1-protected` and `f2-protected` are congruent
- More generally, beta-reduction is valid for Milner's simple encoding of the (CBV) lambda-calculus into pi-calculus (not the case for untyped or simply typed congruence!)

Subtyping

The inclusions

$$\texttt{\^{}[T}_1\texttt{...T}_n\texttt{]} <: \texttt{![T}_1\texttt{...T}_n\texttt{]}$$

$$\texttt{\^{}[T}_1\texttt{...T}_n\texttt{]} <: \texttt{?[T}_1\texttt{...T}_n\texttt{]}$$

give rise to a standard subtyping relation.

Indeed, the usual subtyping rule is validated by the coding of function types:

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{\texttt{![}S_1 \;\; \texttt{![}S_2\texttt{]]} <: \texttt{![}T_1 \;\; \texttt{![}T_2\texttt{]]}}$$

Linear Types

Kobayashi, Pierce, and Turner, ’96

Still too many observers!

A simple “apply twice” function:

```
at1 = applyTwice?*[f:![Int ![Int]]
                   n:Int
                   r:![Int]].
         (new r1) (new r2)
             f![n r1]
           | r1?[x]. ( f![x r2] | r2?[y]. r![y] )
```

After tail-call optimization:

```
at2 = applyTwice?*[f:![Int ![Int]]
                   n:Int
                   r:![Int]].
         (new r1)
             f![n r1] | r1?[x]. f![x r]
```

These are *not* congruent: `f` is not constrained to “behave like a function” (e.g., return only once).

Linear types

Solution: refine channel types

$$
\begin{array}{lll}
\mathtt{T} & ::= & \mathtt{\hat{}[T_1\ \ldots\ T_n]} \quad \text{channel} \\
 & & \mathtt{![T_1\ \ldots\ T_n]} \quad \text{output channel} \\
 & & \mathtt{?[T_1\ \ldots\ T_n]} \quad \text{input channel}
\end{array}
$$

with multiplicities:

$$
\begin{array}{lll}
\mathtt{T} & ::= & \mathtt{\hat{}[T_1\ \ldots\ T_n]} \quad \text{channel} \\
 & & \mathtt{\hat{}1[T_1\ \ldots\ T_n]} \quad \text{linear channel} \\
 & & \mathtt{![T_1\ \ldots\ T_n]} \quad \text{output channel} \\
 & & \mathtt{!1[T_1\ \ldots\ T_n]} \quad \text{linear output channel} \\
 & & \mathtt{?[T_1\ \ldots\ T_n]} \quad \text{input channel} \\
 & & \mathtt{?1[T_1\ \ldots\ T_n]} \quad \text{linear input channel}
\end{array}
$$

cf. Girard's linear logic, many linear lambda-calculi, systems by Honda and Yoshida

Linear Typed Congruence

Observer must play by the rules: If a process “uses up” all the capabilities for a linear channel `x`, then it cannot interact with the observer along `x`.

So

```
(x![] | x?[].  y![])
```

and

```
y![]
```

are congruent if `x` has a linear type, even though `x` is a free variable.

Validity of tail-call optimization

As desired,

```
at1 = applyTwice?*[f:![Int !1[Int]]
                   n:Int
                   r:!1[Int]].
        (new r1) (new r2)
            f![n r1]
          | r1?[x]. ( f![x r2]
                    | r2?[y]. r![y] )
```

and

```
at2 = applyTwice?*[f:![Int !1[Int]]
                   n:Int
                   r:!1[Int]].
        (new r1)
            f![n r1]
          | r1?[x]. f![x r]
```

are indeed congruent.

Linearity $\longrightarrow$ noninterference

Nondeterminism in the pi-calculus arises from “races” between potential senders or receivers:

```
(x![1] | x![2] | x?[n]. y![n])
(x![] | x?[]. y![1] | x?[]. y![2])
```

If x has linear type, these cannot arise!

Theorem [partial confluence]: If $P \xrightarrow{l} Q$ and $P \xrightarrow{k} R$, where l is a reduction on a linear channel, then $Q \xrightarrow{k} S$ and $R \xrightarrow{l} S$ for some S.

Simplifies reasoning about congruence by reducing the number of cases that need to be considered in proofs.

[cf. Honda and Yoshida, ’94,
Kobayashi, Pierce, and Turner ’96,
Honda ’96]

Variations on Linear Typing

Uniform receptiveness:

- Allow any number of senders, but just one (replicated) receiver
- Again we obtain partial confluence
- Used by Sangiorgi to prove correctness of program transformations on concurrent objects

Linearized types:

- Linearized channel may be used many times, but at any moment there is only one process that has the right to use it next [cf. Milner's notion of unique handling]
- "capability passing" programming style
- Observer can only interact along a linearized channel when it is "its turn"

Deadlock Freedom

Preview of work by Sumii and Kobayashi, to appear at HLCL '98
(generalizing work of Kobayashi, Yoshida)

What is Deadlock?

Naive definition:

> Deadlock occurs when a message is sent but never received, or when a receiver waits forever for a send that never occurs.

But...

- a “function server” is always trying to read requests; it is not deadlocked if nobody happens to be sending any
- a lock channel normally has one output pending (indicating that the lock is available); if no one grabs the lock again, this output should not count as deadlocked

What is “deadlock” depends on the programmer’s intention

Usage Calculus

The type system associates each channel with a collection of:

- capabilities, describing communications that may be expected,
- obligations, describing communications that must be provided.

Examples:

- A linear channel must be used exactly once for input and once for output:

  $I_{co} \mid O_{co}$

- A lock channel must initially be set to available (by performing an output); afterwards, it can be repeatedly acquired (by performing an input), but each time it must be released again:

  $O_o \mid !(I_c.\, O_o)$

Usages form a “mini-process calculus” with parallel composition, sequencing, and replication

Deadlock

Deadlock is said to occur if a process whose usage includes an input (or output) capability for a channel attempts to input (or output) on that channel but never succeeds.

Call a channel reliable if its usage includes some obligation to match each capability.

Then, intuitively, deadlock cannot occur as long as all channels are reliable.

But there is one more issue to consider...

Time Tags

The process

$$\mathtt{x?[].y![] \ | \ y?[].x![]}$$

provides an output for each input, so it satisfies the usage

$$\begin{array}{ll} \mathtt{x} & \mathrm{I}_{\mathrm{co}} \mid \mathrm{O}_{\mathrm{co}} \\ \mathtt{y} & \mathrm{I}_{\mathrm{co}} \mid \mathrm{O}_{\mathrm{co}}. \end{array}$$

But it deadlocks.

$\Longrightarrow$ Need to pay attention to ordering of capabilities and obligations.

We attach a time tag to each input/output in a usage expression:

$$\begin{array}{ll} \mathtt{x} & \mathrm{I}^{s}_{\mathrm{co}} \mid \mathrm{O}^{s}_{\mathrm{co}} \\ \mathtt{y} & \mathrm{I}^{t}_{\mathrm{co}} \mid \mathrm{O}^{t}_{\mathrm{co}} \end{array}$$

Now, in $\mathtt{x?[].y![]}$ we have $s < t$, while in $\mathtt{y?[].x![]}$ we have $t < s$. So $s < t < s$, and we have a cyclic dependency.

Deadlock Freedom

Theorem: A process is guaranteed to be deadlock free as long as

1. all its channel usages are reliable, and
2. the ordering on its time tags contains no cycles.

N.b.: Many interesting technicalities are being elided here! See [Sumii and Kobayashi, 98] for details.

Regular Types

Habermann & Campbell, Nierstrasz, Nielson & Nielson, Puntigam, ...

Regular Types

So far, types have always been associated with individual channels. (Processes were simply either well typed or not.)

What about tracking the ordering of communications on different channels?

E.g., a process like

```
x![a] | y?[b]. z![c]
```

might have a type like $O_x\, I_y\, O_z \mid I_y\, O_x\, O_z \mid I_y\, O_z\, O_x$.

types	$\sim$	regular expressions over I_x and O_x for free channels x
input prefixing	$\longrightarrow$	sequencing $(R_1\, R_2)$
replication	$\longrightarrow$	repetition $(R^\star)$
parallel composition	$\longrightarrow$	interleaving $(R_1 \times R_2)$

Details become difficult (e.g., when channels are passed along other channels).

Polymorphism

Turner ’95, Pierce and Sangiorgi ’96

Polymorphic Pi-Calculus

Polymorphism in functional / imperative programming languages supports

- reuse of generic code (lists, sets, etc.)
- information hiding (ADTs)

Polymorphic channel types:

$$\hat{}\,[X_1 \ldots X_m;\ T_1 \ldots T_n]$$

Each communication on a polymorphic channel carries a tuple of channels *and* a tuple of types:

$$\begin{array}{c} x![S_1 \ldots S_m;\ y_1 \ldots y_n] \\ \mid\ x?[X_1 \ldots X_m;\ z_1 \ldots z_n].\ \ P \\ \longrightarrow \\ \{X_1, \ldots X_m, z_1, \ldots z_n \mapsto S_1, \ldots, S_m, y_1, \ldots, y_n\}P \end{array}$$

Example

The polymorphic identity function:

```
(new id: ^[X; X ![X])
     id?*[X; x r]. r!x
```

Typing rules

$$\frac{\Gamma(\mathtt{x}) = \hat{}\,[\mathtt{X}_1 \ldots \mathtt{X}_m;\ \mathtt{T}_1 \ldots \mathtt{T}_n] \qquad \mathtt{y}_i \in \{\mathtt{X}_1, \ldots, \mathtt{X}_m \mapsto \mathtt{S}_1, \ldots, \mathtt{S}_m\}\mathtt{T}_i \quad \text{for each } i}{\Gamma \vdash \mathtt{x!}[\mathtt{S}_1 \ldots \mathtt{S}_m;\ \mathtt{y}_1 \ldots \mathtt{y}_n]}$$

$$\frac{\Gamma(\mathtt{x}) = \hat{}\,[\mathtt{X}_1 \ldots \mathtt{X}_m;\ \mathtt{T}_1 \ldots \mathtt{T}_n] \qquad \mathtt{X}_1 \ldots \mathtt{X}_m \notin \Gamma \qquad \Gamma, \mathtt{z}_1{\in}\mathtt{T}_1, \ldots, \mathtt{z}_n{\in}\mathtt{T}_n \vdash \mathtt{P}}{\Gamma \vdash \mathtt{x?}[\mathtt{X}_1 \ldots \mathtt{X}_m;\ \mathtt{z}_1 \ldots \mathtt{z}_n].\ \ \mathtt{P}}$$

(Cf. introduction/elimination rules for existential types)

Note that the primitive form of polymorphism in the pi-calculus is existential types, not universal. (This corresponds to the fact that functions in the pi-calculus are written in CPS style.)

Example: An ADT of Booleans

Begin with a global channel

$$\texttt{getBoolOps} \in \texttt{\^{}[B; B B ![B ![] ![]]]}$$

The observer (or another client process) may read from `getBoolOps` to obtain a type `B` and channels

`t`	$\in$	`B`	constant true
`f`	$\in$	`B`	constant false
`cond`	$\in$	`![B ![] ![]]`	conditional

E.g.

```
getBoolOps?[B; t f cond].
  (new a) (new b)
     cond![t a b]
   | a?[]. print!["A"]
   | b?[]. print!["B"]
```

Two Different Boolean ADTs

```
makebools1 =
  (new t) (new f) (new cond)
       t?*[x y]. x![]
     | f?*[x y]. y![]
     | cond?*[b th el].
         b![ th el ]
     | getBoolOps![![![] ![]]; t f cond]

makebools2 =
  (new t) (new f) (new cond)
       t?*[x y]. y![]
     | f?*[x y]. x![]
     | cond?*[b th el].
         b![ th el ]
     | getBoolOps![![![] ![]]; t f cond]
```

[cf. Mitchell and Plotkin's encoding of ATDs
in polymorphic lambda-calculus]

Another ADT of Booleans

Alternative implementation:

```
makebools2 =
  (new t) (new f) (new cond)
        t?*[x:![] y:![]]. y![]
      | f?*[x:![] y:![]]. x![]
      | cond?*[b:BoolImpl th:![] el:![]].
           b![ el th ]
      | getBoolOps![BoolImpl; t f cond]
```

First, note that the untyped versions of `makebools1` and `makebools2` are obviously not congruent (they are not even trace-equivalent).

However, the typed versions of `makebools1` and `makebools2` *are* congruent, since a well-typed observer cannot use its `t` or `f` channels directly for interacting with `makebools` — it can only pass them as arguments along `cond`.

Information Leakage

Surprising fact: an observing process *can* sometimes tell the difference between two channels of completely abstract type!

Example: suppose

```
x ∈ ^[]
a ∈ ^[^[]]
g ∈ ^[X; X ^[X]]
```

Then a well-typed observer can tell the difference between

```
P = g![^[]; x a]
Q = (new y  g![^[]; y a]
```

using

```
tester = g?[X; p q]. (q![p] | a?[z]. z![])
```

Running `(tester | p)` has the effect of sending `[]` along `x`, while `(tester | q)` does not.

Similar examples can be given in any polymorphic language with aliasing, e.g. ML with references.

Conclusions

Summary

A variety of static type systems from functional and imperative programming languages can be transferred to concurrent languages.

More refined typings lead to more accurate (coarser) congruences by restricting the power of observers.

More recent proposals are tuned to the concurrent setting

- uniform receptiveness
- deadlock-freedom

Challenge

A type-theoretic characterization of “termination” for pi-calculus processes

(cf. strong normalization theorems for typed lambda-calculi)