Processes as Types:
A Generic Framework of Behavioral Type Systems for Concurrent Processes

Atsushi Igarashi (Kyoto Univ.)

based on joint work [POPL2001, TCS2003]

with Naoki Kobayashi (Tohoku Univ.)
Programming is hard ...
Concurrent programming is much harder, because...
Additional Complexity in Concurrent Programs

- Multiple threads of control
- Non-determinism
- Deadlock / livelock
Static Checking to Rescue?

Two popular approaches:

♦ Type Systems
  - (Said to be) good at finding 'shallow' bugs
    • e.g., arity mismatch in communication
  - Directly deal with program code

♦ Model Checking
  - Good at verifying 'deep' properties
    • e.g., deadlock freedom
  - Daunting(?) model extraction from programs
Previous Type Systems

♦ **I/O mode** ([Pierce & Sangiorgi 93])
  - Channels are used for correct I/O modes.

♦ **Linearity, race conditions, atomicity** ([Kobayashi, Pierce & Turner 96] [Abadi, Flanagan & Freund 99, 2000] etc.)
  - Certain communications do not suffer from non-determinism.

♦ **Deadlock/Livelock-freedom** ([Yoshida 96; Kobayashi et al. 97, 98, 2000; Puntigam 98] etc.)
  - Certain communications succeed eventually.

♦ **Security properties** ([Honda, Vasconcelos & Yoshida 2000; Hennessy & Riely 2000; Kobayashi 2005] etc.)
Problems of Previous Type Systems for Concurrent Programs

- Designed in an ad hoc manner
  - Unclear essence
  - Difficulty of integrating different type systems.
  - A lot of repeated work:
    - type soundness proofs
    - type inference algorithms

⇐ No common framework

c.f. Curry-Howard isomorphism, Effect systems
This Talk:

**Generic Type System**

♦ Provides a common framework of type systems for concurrent programs

♦ Can be instantiated easily to various type systems (e.g., for race-freedom, deadlock-freedom)

♦ Enables sharing of a large amount of work for development of type systems
  - type soundness proofs
  - type inference algorithms
Idea: Types as Abstract Processes

c.f. Abstract Interpretation [Cousot&Cousot77]
Idea: Types as Abstract Processes

c.f. Abstract Interpretation [Cousot&Cousot77]
Idea: Types as Abstract Processes

User Program (Concrete Process)

\textbf{type check/inference}

Process Type (Abstract Process)

\textbf{analyzer/verifier}

\(\pi\)-calculus [Milner et al.]:
Dynamic change of communication topology
⇒ Expressive, but hard to analyze

CCS (w/o channel creation):
No dynamic change of communication topology
⇒ Much easier to analyze
Idea: Types as Abstract Processes

User Program
(Concrete Process)

*type check/inference*

Process Type
(Abstract Process)

Hybrid approach combining
- Type systems
  - Type inference as syntax-directed, automatic model extraction from a program
- Model checking
  - Analyzer/verifier as model checker

*analyzer/verifier*
Outline

♦ Target Language
  - Syntax
  - Operational semantics
♦ Process Types
♦ Generic Type System
  - Typing rules
  - Type soundness
♦ How to obtain specific type systems

User Program (Concrete Process)

\[\text{type check/inference}\]

Process Type (Abstract Process)

\[\text{analyzer/verifier}\]
Outline

♦ Target Language
  - Syntax
  - Operational semantics

♦ Process Types

♦ Generic Type System
  - Typing rules
  - Type soundness

♦ How to obtain specific type systems

User Program (Concrete Process)

Process Type (Abstract Process)

*type check/inference*

*analyzer/verifier*
Outline

♦ Target Language
  - Syntax
  - Operational semantics

♦ Process Types

♦ Generic Type System
  - Typing rules
  - Type soundness

♦ How to obtain specific type systems

User Program (Concrete Process)
  
  type check/inference

Process Type (Abstract Process)
  
  analyzer/verifier
Outline

♦ Target Language
  - Syntax
  - Operational semantics

♦ Process Types

♦ Generic Type System
  - Typing rules
  - Type soundness

♦ How to obtain specific type systems
Target Language
- Syntax
- Operational semantics

Process Types

Generic Type System
- Typing rules
- Type soundness

How to obtain specific type systems

User Program (Concrete Process)

**type check/inference**

Process Type (Abstract Process)

**analyzer/verifier**
Target Language: $\pi$-calculus [Milner et al.]

Calculus of concurrent processes with:
- Message passing via communication channels
- First-class channels
- Dynamically created channels
- Infinite behavior by replication
Target Language: \( \pi \)-calculus [Milner et al.]

\( P, Q \) (Processes) ::=  

- \( 0 \) (inaction)  
- \( x!^t[v_1, \ldots, v_n].P \) (output)  
- \( x?^t[y_1, \ldots, y_n].P \) (input)  
- \( \text{new } x_1, \ldots, x_n \text{ in } P \) (channel creation)  
- \( P|Q \) (parallel execution)  
- \( \ast P \) (replication: \( \cong P | P | \ldots \))  

\( s, t \) : labels to identify program points

\( x![v_1, \ldots, v_n].P | x?[y_1, \ldots, y_n].Q \rightarrow P | [v_1/y_1, \ldots, v_n/y_n]Q \)  
(c.f. \( \beta \)-reduction: \( (\lambda x.M)N \rightarrow [N/x]M \))
Example: Function Server

Server: \( *\text{succ}?[n, r].r ![n+1] \)

Client: new \( r \) in (\( \text{succ}![1,r] \mid r? [x]... \))
Example: Function Server

Server: \( *\text{succ}[n, r].r ![n+1] \)

Client: \( \text{new } r' \text{ in } (\text{succ}![1, r] | r? [x]...) \)

\( *\text{succ}[n, r].r ![n+1] | \text{new } r' \text{ in } (\text{succ}![1, r'] | r' ?[x].\text{print}![x]) \)
Example: Function Server

Server: *success?[n, r].r ![n+1]

Client: new r in (success![1, r] | r? [x]...)

success?[n, r].r ![n+1] new r’ in (success![1, r’] | r’ ?[x].print![x])

server

client

→ *success?[n, r].r ![n+1] new r’ in (r’ ![2] r’ ?[x].print![x])
Example: Function Server

Server: \[ \ast succ?[n, r].r ![n+1] \]

Client: \( \text{new } r \text{ in } (succ![1,r] | r? [x]...) \)

\[
\ast succ?[n, r].r ![n+1] | \text{new } r' \text{ in } (succ![1,r'] | r'?[x].print![x])
\]

→ \[
\ast succ?[n, r].r ![n+1] | \text{new } r' \text{ in } (r' ![2] | r' ?[x].print![x])
\]

→ \[
\ast succ?[n, r].r ![n+1] | \text{print} ![2]
\]
Outline

♦ Target Language
  - Syntax
  - Operational semantics

♦ Process Types

♦ Type System
  - Typing rules
  - Type soundness

♦ Instances of Generic Type System
Process Types

\[ \Gamma, \Delta \text{ (process types) ::= 0 (inaction)} \]

\[ x!^t[\tau]. \Gamma \text{ (output a value on } x \text{, then behaves like } \Gamma) \]

\[ x?^t[\tau]. \Gamma \text{ (input from } x \text{, then behaves like } \Gamma) \]

\[ t.\Gamma \text{ (wait for event } t \text{, then behaves like } \Gamma) \]

\[ \Gamma|\Delta \text{ (parallel composition)} \]

\[ *\Gamma \text{ (replication)} \]

\[ \Gamma & \Delta \text{ (non-deterministic choice)} \]

\[ \tau \text{ (tuple types) ::= } \]

\[ (x_1,\ldots,x_n)\Gamma \text{ (type of a tuple of the form } [x_1,\ldots,x_n], \]

\[ \text{which should be used according to } \Gamma) \]
Examples

♦ $x^s[\text{Int}].y^t[\text{Int}]$
  - Receives an integer through $x$ and then sends an integer through $y$
    (e.g. $x^s[n].y^t[n+1]$)

♦ $*x^s[(y)y^t[\text{Int}]]$
  - Repeatedly receives a channel and sends an integer through the received channel
    (e.g. $*x^s[y].y^t[2]$)
Examples

♦ \(\mu \alpha. \text{put?}[\text{Int}]. \text{get?}[(y) \ y!]\text{[Int]}]. \alpha\)

- The type of a one-place buffer:
  - \(\text{Buffer} = \text{put?}[n]. \text{get?}[r]. r!\text{[n]}\).\text{Buffer}\)
    -(expressed by using replication)
  - \(\mu \alpha. \Gamma \ldots\) recursive process type that satisfies \(\mu \alpha. \Gamma = [\mu \alpha. \Gamma/\alpha] \Gamma\)
    -(\(*\Gamma\) is actually a syntax sugar using \(\mu\)\)
Process Types Form a Mini-Process Calculus

\[ x!\tau.\Gamma | x?\tau.\Delta \rightarrow \Gamma | \Delta \]

(c.f. \[ x!v.P | x?y.Q \rightarrow P | [v/y]Q \])

**e.g.** \[ x!\tau.y!\text{Int} | x?\tau \rightarrow y!\text{Int} \]

\( \tau = (z)z!\text{Int} \)

\[ x!y | x?z.z!2 \rightarrow y!2 \]
### Summary of Processes and Process Types

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x!^t[v_1,...,v_n].P$</td>
<td>output</td>
<td></td>
</tr>
<tr>
<td>$x?^t[y_1,...,y_n].P$</td>
<td>input</td>
<td></td>
</tr>
<tr>
<td>$P</td>
<td>Q$</td>
<td>parallel</td>
</tr>
<tr>
<td>new $x_1,...,x_n$ in $P$</td>
<td>new channel</td>
<td></td>
</tr>
<tr>
<td>$*_P$</td>
<td>replication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wait</td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-determinism</td>
<td></td>
</tr>
</tbody>
</table>

- $x!^t[v_1,...,v_n].P$: Output
- $x?^t[y_1,...,y_n].P$: Input
- $P|Q$: Parallel
- new $x_1,...,x_n$ in $P$: New channel
- $*_P$: Replication
- $x!^t[\tau].\Gamma$: Output
- $x?^t[\tau].\Gamma$: Input
- $\Gamma|\Delta$: Parallel
- $\Gamma \& \Delta$: Non-determinism
- $*_\Gamma$: Replication
- $t.\Gamma$: Wait
<table>
<thead>
<tr>
<th>Process Expression</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x!^t[v_1,\ldots,v_n].P)</td>
<td>output</td>
<td>x!^t[\tau].\Gamma</td>
</tr>
<tr>
<td>(x?^t[y_1,\ldots,y_n].P)</td>
<td>input</td>
<td>x?^t[\tau].\Gamma</td>
</tr>
<tr>
<td>(P</td>
<td>Q)</td>
<td>parallel</td>
</tr>
<tr>
<td>new (x_1,\ldots,x_n) in (P)</td>
<td>new channel</td>
<td></td>
</tr>
<tr>
<td>(*P)</td>
<td>replication</td>
<td>(*\Gamma)</td>
</tr>
<tr>
<td></td>
<td>wait</td>
<td>(t.\Gamma)</td>
</tr>
<tr>
<td></td>
<td>non-determinism</td>
<td>(\Gamma &amp; \Delta)</td>
</tr>
</tbody>
</table>

No value passing, only synchronization behavior
Outline

♦ Target Language
  - Syntax
  - Operational semantics

♦ Process Types

♦ Type System
  - Typing rules
  - Type soundness

♦ Instances of Generic Type System

User Program (Concrete Process)

**type check/inference**

Process Type (Abstract Process)

 analyzer/verifier
Recipe for Type Systems

♦ Type judgment relation
  - with supposed meaning

♦ Typing rules to derive type judgments
  - One rule for one syntactic construct

♦ Type soundness theorem
  - Evidence that the supposition is indeed true

♦ (Type inference algorithm)
Type Judgment

\[ \Gamma \vdash P \]

- \( P \) has process type \( \Gamma \)
- \( \Gamma \) is an abstraction of \( P \)
- \( P \) matches specification \( \Gamma \)

Examples:

- \( x?^s[\text{Int}].y!^t[\text{Int}] \vdash x?^s[n].y!^t[n+1] \)
- \( *x?^s[(y)y!^t[\text{Int}]] \vdash *x?^s[y].y!^t[2] \)
## Summary of Processes and Process Types

<table>
<thead>
<tr>
<th>Process Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x!t[v_1,...,v_n].P</code></td>
<td>output</td>
</tr>
<tr>
<td><code>x?t[y_1,...,y_n].P</code></td>
<td>input</td>
</tr>
<tr>
<td>`P</td>
<td>Q`</td>
</tr>
<tr>
<td><code>new x_1,...,x_n in P</code></td>
<td>new channel</td>
</tr>
<tr>
<td><code>*P</code></td>
<td>replication</td>
</tr>
<tr>
<td><code>wait</code></td>
<td></td>
</tr>
<tr>
<td><code>non-determinism</code></td>
<td></td>
</tr>
<tr>
<td><code>x!t[\tau].\Gamma</code></td>
<td></td>
</tr>
<tr>
<td><code>x?t[\tau].\Gamma</code></td>
<td></td>
</tr>
<tr>
<td>`\Gamma</td>
<td>\Delta`</td>
</tr>
<tr>
<td><code>*\Gamma</code></td>
<td></td>
</tr>
<tr>
<td><code>t.\Gamma</code></td>
<td></td>
</tr>
<tr>
<td><code>\Gamma&amp;\Delta</code></td>
<td></td>
</tr>
</tbody>
</table>
Typing Rule (parallel composition)

\[ \Gamma \vdash P \quad \Delta \vdash Q \]

\[ \Gamma \mid \Delta \vdash P \mid Q \]

(similarly for \(*\) )
<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x!^t[v_1,\ldots,v_n].P$</td>
<td>output</td>
<td></td>
</tr>
<tr>
<td>$x?^t[y_1,\ldots,y_n].P$</td>
<td>input</td>
<td></td>
</tr>
<tr>
<td>$P \mid Q$</td>
<td>parallel</td>
<td></td>
</tr>
<tr>
<td>new $x_1,\ldots,x_n$ in $P$</td>
<td>new channel</td>
<td></td>
</tr>
<tr>
<td>$*P$</td>
<td>replication</td>
<td></td>
</tr>
<tr>
<td>$\text{wait}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{non-determinism}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Typing Rule (Output)

$$\Gamma \vdash P$$

$$x!^t[(y)\Delta].(\Gamma \mid [v/y]\Delta) \vdash x!^t[v].P$$

$\Delta$ expresses how the receiver uses $y$

Example:

$$x!^t[\tau].y!^u[Int] \mid x?^s[\tau] \vdash x!^t[y] \mid x?^s[z].z!^u[2]$$

(for $\tau = (z)z!^u[Int]$)
Typing Rule (Input)

\[ \Gamma, \Delta \vdash P \quad y \not\in \text{FV}(\Gamma) \]

\[ x!^t[(y)\Delta].\Gamma \vdash x?^t[y].P \]

Example:

\[ x!^t[\tau].y!^u[\text{Int}] \mid x?^s[\tau] \vdash x!^t[y] \mid x?^s[z].z!^u[2] \]

(for \( \tau = (z)z!^u[\text{Int}] \))
Typing Rule (Channel Creation)

\[
\begin{align*}
\Gamma & \vdash P \quad \text{ok}(\Gamma \downarrow \{x_1, \ldots, x_n\}) \\
\Gamma \uparrow \{x_1, \ldots, x_n\} & \vdash \text{new } x_1, \ldots, x_n \text{ in } P
\end{align*}
\]

\text{ok}(\Gamma \downarrow \{x_1, \ldots, x_n\}) : 

Check that \(x_1, \ldots, x_n\) are used 'appropriately'

(\text{Depending on type system instances})

\Gamma \uparrow \{x_1, \ldots, x_n\} : 

Possible blocking recorded by \(t.\Gamma\) (for deadlock analysis)

\[
(x?^{t[\tau]}\Gamma)\downarrow \{x\} = t.(\Gamma \downarrow \{x\})
\]
Typing Rule (Subsumption)

\[ \Gamma \vdash P \quad \Gamma' \leq \Gamma \]

\[ \Gamma' \vdash P \]

Process type can be replaced with a coarser abstraction.

- \( \Gamma' \leq \Gamma \): Subtyping relation
  - Depends on type system instances
Weak Type Soundness Theorem
(Subject Reduction)

\[ \Gamma \vdash P \]

\[ \Delta \vdash Q \]

\[ \Gamma \text{ simulates the behavior of } P \]
**General Type Inference**

◆ Principal typing theorem:
For any process $P$, there is a “most general” type $\Gamma$ s.t. $\Gamma \vdash P$

− (after a slight (routine) modification of the type system)
Outline

♦ Target Language
♦ Process Types
♦ Generic Type System
♦ Instances
  - Analysis of race, deadlock
  - What type system can be obtained, and how?

User Program (Concrete Process)

Type check/inference

Process Type (Abstract Process)

Analyzer/verifier
A process $P$ is **race-free** on output if:

$$P \not\rightarrow^* \text{new } y_1, \ldots, y_n \text{ in } (x![...,Q_1 \mid x![...,Q_2 \mid R)$$

A type $\Gamma$ is **race-free** on output if:

$$\Gamma \not\rightarrow^* x!^s[\tau].\Delta_1 \mid x!^t[\tau].\Delta_2 \mid \emptyset$$

**Theorem:** If $\Gamma \models P$, $\Gamma$ is **race-free**, and $\text{ok}(\Delta)$ implies $\Delta$ is race-free, then $P$ is race-free.
To verify a property of $P$, verify the corresponding property of $\Gamma$.
Example: Race Analysis

\[ x!^[1] \mid y!^[x] \mid y?[z].z!^[2] \]

\[ \downarrow \]

\[ x![\text{Int}] \mid y!^[\tau]. x![\text{Int}] \mid y?[\tau] \rightarrow x![\text{Int}] \mid x![\text{Int}] \]

race on X
Example: Deadlock Analysis

new y in (x?^S[n].y!^t[n+1] | y?^u[m].x!^v[m+1])

♦ The type of y: s.y!^t[Int] | y?^u[Int].v
  - Event S must occur before event U occurs

♦ The type of x: x?^S[Int] | x!^v[Int]
  - Event U must occur before event S occurs

Cannot hold at once!
Example: Concurrent Objects with Non-uniform Service Availability

[Puntigam'99, Ravara&Vasconcelos.'00]

Buffer = put?[n].get?[r].r![n].Buffer

Can be viewed as a concurrent object with two methods put and get, invoked alternately

- $\Gamma_{buf} = \mu \alpha.\\put?[\text{Int}].\\get?[(y) y!][\text{Int}].\alpha \vdash \text{Buffer}$

- $\Gamma_{client} = \mu \alpha.0&(put![\text{Int}] | get![(y) y!][\text{Int}].\alpha)$

- $\Gamma_{buf} \mid \Gamma_{client}$ never get deadlocked (on outputs)

- $\Gamma_{client} \mid \Gamma_{client}$ put ![2].new r in get ![r].r ![n].P

- $\Gamma_{client} \mid \Gamma_{client}$ put ![2] | new r in get ![r].r ![n].P

- $\Gamma_{client} \mid$ new r in get ![r].r ![n].put![2]
General Principle Revisited

To verify a property of $P$, verify the corresponding property of $\Gamma$

♦ How can we find the “corresponding” property?
♦ For what kind of property, does the “corresponding” property exist?
How to Obtain Specific Type Systems?

♦ To guarantee that every process satisfies $A$:
  
  – Choose $A^\ast$ so that:
    
    $P$ satisfies $A$ whenever its type $\Gamma$ satisfies $A^\ast$
  
  – Accept $P$ only if its type $\Gamma$ satisfies $A^\ast$

♦ How can we find $A^\ast$?

♦ For what $A$, does $A^\ast$ exist?
Logic to Describe Properties of Processes and Types

\[ A, B \] (formulae) ::= 

\[ x! \] (Ready to output a value on \( x \))

\[ x? \] (Ready to input a value from \( x \))

\[ A \| B \] (Parallel composition of a process satisfying \( A \) and a process satisfying \( B \))

\[ <x> A \] (Can satisfy \( A \) after a communication on \( x \))

\[ ev(A) \] (Can eventually satisfy \( A \))

\[ \neg A \]

\[ A \lor B \]

**Example:**

\[ \neg \exists x.\ ev(x! \mid x!) \] No race on output.
Semantics of the Logic

♦ \( P |= A : \) Process \( P \) satisfies \( A \)

- \( x![\ ].Q | y![\ ].R |= x! \)
- \( x![\ ].Q | y![\ ].R |= x! | y! \)
- \( x![\ ].y![\ ].Q \not|= x! | y! \)
- \( x![\ ] | x?[\ ].y![\ ] |= \text{ev}(y!) \land <x>y! \)

♦ \( \Gamma |= A : \) Process type \( \Gamma \) satisfies \( A \)
Logic for Properties

\[ A \downarrow B : \Gamma \vdash P \text{ and } \Gamma \models A \implies P \models B \]

\[ A \uparrow B : \Gamma \vdash P \text{ and } P \models A \implies \Gamma \models B \]

\[ \frac{A \uparrow C \quad B \uparrow D}{A|B \uparrow C|D} \quad \frac{A \uparrow B}{\text{ev}(A) \uparrow \text{ev}(B)} \]

\[ \frac{A \uparrow B \quad A \uparrow B \lor C}{\neg A \downarrow \neg B} \]
Strong Type Soundness

Theorem: \( \downarrow A \) for any “negative” formula \( A \).

(bad) things do not happen

In other words, ...

Let \( A \) be a negative formula.

If \( \Gamma \vdash P \) and \( \Gamma \models A \), then \( P \models A \).

Examples of negative formulas:

- \( \neg \exists x.\text{ev}(x! \mid x!) \) No race on output
- \( \neg \exists x.\text{ev}(\langle x\rangle\text{ev}(x!\lor x?)) \) No channel is used twice
To verify a property of $P$, verify the corresponding property of $\Gamma$

♦ How can we find the “corresponding” property?

Choose the property described by the same formula

♦ For what kind of property, does the “corresponding” property exist?

For any negative formula, at least
To guarantee that every process satisfies $A$:

- Choose $A^*$ so that:
  
  $P$ satisfies $A$ whenever its type $\Gamma$ satisfies $A^*$

- Accept $P$ only if its type $\Gamma$ satisfies $A^*$

How can we find $A^*$?

$A^* = A$

For what $A$, does $A^*$ exist?

For any negative formula $A$
Deadlock Freedom Revisited

- Deadlock freedom cannot be described by a negative formula
  - Action labelled $t$ does not deadlock
    = “whenever an action labelled $t$ is tried, there is a further reduction”

- Separate proof of soundness required

(good) things do happen
Some Limitations

♦ Due to expressiveness of process types
  – Information on channel creation lost
  – Impossible to check “at most n channels are created”

♦ Due to the requirement for $ok(\Gamma)$
  – Invariant cond.: if $ok(\Gamma)$ and $\Gamma \rightarrow \Gamma'$, then $ok(\Gamma')$
  – Impossible to check “before x is used, y should be used”
Conclusion

Generic type system for concurrent progs

- Key idea: Abstract Processes as Types
- Many type systems are obtained as instances:
  - Race detection
  - Deadlock-freedom
    - Concurrent objects with non-uniform service availability
  - Linear channels, etc.
- Many issues can be discussed uniformly.
  - type soundness
  - type inference
Combination of

- Type Theory
- Model Checking

- Chaki et al. [POPL2002] implemented (a variant of) this framework using SPIN as model checker
Current/Future Work

♦ Extensions of the generic type system
  - More expressive power
    • Restriction operator [Chaki et al. 2002, Kobayashi 2005]
  - More of common theories
    • 'Generic' typed process equivalence

♦ Formal verification of the correctness of the generic type system (using Coq)
  - Automatic extraction of type inference algorithms?