Behavioral Types as Interfaces for Concurrent Processes

Naoki Kobayashi
Tohoku University
What’s This Talk About?

♦ Review of behavioral types for $\pi$-calculus, from the perspective of interfaces
  - Channel usage type system
  - Extensions
    • Capability/obligation levels
    • CCS types
  - Undecidable type systems
  - Hybrid type systems
What are Interfaces?

♦ Abstract specifications of components
  - Abstract enough to hide implementation details
  - Precise enough to determine composability of components
  - Equipped with algorithms for:
    • Interface conformance: \( P \models I \)
    • Subinterface: \( I < I' \)
Types as Interfaces?

- Abstract specifications of components
  - Abstract enough to hide implementation details yes
  - Precise enough to determine composability of components yes
- Equipped with algorithms for:
  - Interface conformance: $P \models I$ type checking
  - Subinterface: $I < I'$ subtyping
Types as Interfaces for *Concurrent* Components?

- Abstract specifications of components
  - Abstract enough to hide implementation details
  - Precise enough to determine composability of components
  - Equipped with algorithms for:
    - Interface conformance: \( P \models I \)
    - Subinterface: \( I \prec I' \)

- type checking
- subtyping

- yes
- ??
Behavioral Types as Interfaces for Concurrent Components?

- Abstract specifications of components
  - Abstract enough to hide implementation details
    yes
  - Precise enough to determine composability of components
    maybe
  - Equipped with algorithms for:
    - Interface conformance: \( P \models I \)
    type checking?
    - Subinterface: \( I < I' \)
    subtyping?
Outline

♦ $\pi$-calculus
♦ Channel usage type system
♦ Extensions
  - capability/obligation levels
  - CCS types
♦ Algorithmic issues
  - (Un)decidability of usage type system
  - Impreciseness of type checking as conformance checking
What is $\pi$-calculus?

- Consists of basic concurrency primitives

```
new r in c![1,r] | c?[x,y].y![x] → r![1]
```

- Channel creation
- Message send
- Message reception
What is $\pi$-calculus?

♦ Consists of basic concurrency primitives

\[ \text{new } r \text{ in } c!\{1,r\} \mid c?\{x,y\}.y!\{x\} \]

♦ Expressive enough to model features of modern programming languages
  - (Higher-order) Functions
  - Concurrent objects
  - Synchronization mechanisms (locks, etc.)
Syntax of $\pi$-calculus

$P, \ Q$ (Processes) ::= 
- $0$ (inaction)
- $\text{new } x \ \text{in } P$ (channel creation)
- $x!v.P$ (output)
- $x?y.P$ (input)
- $P|Q$ (parallel execution)
- $\text{if } b \ \text{then } P \ \text{else } Q$ (conditional)
- $*P$ (replication)

$x!v.P \ | \ x?y.Q \rightarrow P \ | \ [v/y]Q$
(c.f. $(\lambda x.M)N \rightarrow [N/x]M$)
Outline

♦ $\pi$-calculus
♦ Channel usage type system
♦ Extensions
  - capability/obligation levels
  - CCS types
♦ Algorithmic issues
  - (Un)decidability of usage type system
  - Impreciseness of type checking as conformance checking
Channel Usage Types

Channel types extended with *usages*, expressing how each channel is used.

- **chan(Int, !)**: channels used once for sending an integer.
- **chan(Int, ?.!):** channels used for receiving an integer, and then for sending an integer.
- **chan(chan(Int, !), ?):** channels used for receiving a channel that should be used for sending an integer.
Channel Usage

\[ U ::= 0 \quad \text{not used} \]
\[ ?.U \quad \text{used for input, and then as } U \]
\[ !.U \quad \text{used for output, and then as } U \]
\[ U_1 | U_2 \quad \text{used as } U_1 \text{ and } U_2 \text{ in parallel} \]
\[ U_1 & U_2 \quad \text{used as } U_1 \text{ or } U_2 \]
\[ \mu a.U \quad \text{recursion} \]
\[ *U \quad \text{used as } U \text{ arbitrarily many times} \]
Type Judgment

\[ x_1: \tau_1, \ldots, x_n: \tau_n |- P \]

P uses each \( x_i \) according to \( \tau_i \)

Example:

\checkmark \ x:Chan(Int, !) |- x![1]

\times \ x:Chan(Int, !), b:bool |- 
\quad \text{if } b \text{ then } x![1] \text{ else } 0

\checkmark \ \text{ping: Chan(Chan(Int,!), ?)} |- \text{ping?[r].r![1]}
Typing Rules

\[ \Gamma, y: \tau, x: \text{Chan}(\tau, U) \vdash P \]

\[ \begin{array}{c}
\Gamma, x: \text{Chan}(\tau, ?.U) \vdash x?[y].P \\
\end{array} \]

\[ \begin{array}{c}
\Gamma \vdash P \\
\Delta \vdash Q \\
\hline
\Gamma \mid \Delta \vdash P \mid Q \\
\end{array} \]
Type Environments as Interfaces

- Interface for a lockable reference cell (from a client's view)

read: (Int Chan(!)) Chan(*!),
write: (Int \times Unit Chan(!)) Chan(*!)
lock: Unit Chan(*?.!)

- An integer will be returned for each request
- Read request can be sent arbitrarily many times
- Can be locked arbitrarily many times. Must be unlocked after each locking
Outline

♦ $\pi$-calculus
♦ Channel usage type system
♦ Extensions
  - capability/obligation levels
  - CCS types
♦ Algorithmic issues
  - (Un)decidability of usage type system
  - Impreciseness of type checking as conformance checking

Basis of TyPiCal program analysis tool for deadlock/livelock-freedom, information flow analysis, etc.
Why Extensions?

♦ Usage types are not precise enough to determine composability

\[ (a:?, b:!) \mid (b:?, a:!) \]
Why Extensions?

Usage types are not precise enough to determine composability.

\[ a : \text{?}, b : \text{!} \vdash a ? x . b ! x \]

\[ b : \text{?}, a : \text{!} \vdash b ? x . a ! x \]

\[ a : \text{?} | !, b : ! | ? \]

\[ a ? x . b ! x \mid b ? x . a ! x \]

\[ \checkmark \quad \text{deadlock!} \]
Capability/Obligation Levels
[Kobayashi 05, Acta Informatica]

♦ Annotations for usages ! and ?
  - e.g. ?(0, ∞)

♦ Prevent cyclic dependencies between:
  - capabilities (to successfully communicate)
    • assumptions on the environment
  - obligations (to send/receive messages)
    • guarantees for the environment
    (c.f. assume-guarantee reasoning)

♦ Obligations must be fulfilled by using only capabilities of smaller levels
**Capability/Obligation levels: Example**

**Capability Level**

\[ a : ?(0,-), \quad b : !(-,1) \]

**Obligation Level**

\[ b : ?(0,-), \quad a : !(-,1) \]

\[ a?x.b!x \]

\[ b?x.a!x \]

**Assuming the success of input on \( a \), \( b \) will be used for output**
**Capability/Obligation levels:**

**Example**

Assuming the success of input on b, a will be used for output.
Capability/Obligation levels: Example

<table>
<thead>
<tr>
<th>Capability Level</th>
<th>Obligation Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a: ?(0, -)$, $b: !(−, 1)$</td>
<td>$b: ?(0, -)$, $a: !(−, 1)$</td>
</tr>
</tbody>
</table>

$\neg a?x.b!x$

$\neg b?x.a!x$

$a: ?(0, -)$ $|$ $!(−, 1)$

$b: !(−, 1)$ $|$ $?(0, −)$

Capability (or, assumption) is not met by obligation (or, guarantee)
**Capability/Obligation levels:**

**Example**

- **Capability Level**
  - $a: ?(0, -)$
  - $b: !(-, 1)$

- **Obligation Level**
  - $b: ?(1, -)$
  - $a: !(-, 0)$

- **Logic:**
  - $a?x.b!x$
  - $a!1.b?x$
  - $a?x.b!x | a!1.b?x$
  - $a: ?(0, -) | !(-, 0)$
  - $b: !(-, 1) | ?(1, -)$

**Result:**

- **Capability (or, assumption) is met by obligation (or, guarantee)**
CCS Processes as Types
[Igarashi&Kobayashi POPL01][Chaki et al. POPL 02]

\[ a?.b! \mid a?x.b!x \quad b?.a! \mid \neg b?x.a!x \]

\[ a?.b! \mid b?.a? \]

\[ \times \text{ deadlock!} \]
Outline

♦ $\pi$-calculus
♦ Channel usage type system
♦ Extensions
  - capability/obligation levels
  - CCS types
♦ Algorithmic issues
  - (Un)decidability of usage type system
  - Impreciseness of type checking as conformance checking
Behavioral Types as Interfaces for Concurrent Components?

♦ Abstract properties of components
  - Abstract enough to hide implementation details
  - Precise enough to determine composability of components
  - Equipped with algorithms for:
    • Interface conformance: \( P \models I \)
    • Subinterface: \( I < I' \)

  yes
  yes
  probably yes (depending on “composability”)

  type checking?
  subtyping?
Undecidable Behavioral Type Systems
[Kobayashi&Suto, ICALP 2007]

Subusage (or, sub-interface) relation $U \leq U'$ is a simulation for 2-label BPP, which is undecidable

$U ::= 0 \quad \text{not used}$

$? . U \quad \text{used for input, and then as } U$

$! . U \quad \text{used for output, and then as } U$

$U_1 \mid U_2 \quad \text{used as } U_1 \text{ and } U_2 \text{ in parallel}$

$U_1 \& U_2 \quad \text{used as } U_1 \text{ or } U_2$

$\mu a . U \quad \text{recursion}$

$^*U \quad \text{used as } U \text{ arbitrarily many times}$
Subusage (or, sub-interface) relation $U \leq U'$ is a simulation for 2-label BPP, which is undecidable.

⇒ A channel usage type system with explicit usage declaration is undecidable.

Solution: Restrict the language of usage declarations (e.g., to regular languages or deterministic Petri net languages) (c.f. [Kobayashi et al., VMCAI 2006])
Limitation of Type Checking as Interface Checking Algorithm

♦ Too imprecise local reasoning

\[
\begin{align*}
a &: (?.!)&0, \ n : \text{Int} & |\times \ \\
& (\text{if } n > 0 \ \text{then } a?x \ \text{else } 0); \\
& (\text{if } n > 0 \ \text{then } a!x \ \text{else } 0)
\end{align*}
\]

Solution:

- Integration with other verification methods via hybrid or semantic type systems
  
  [Kobayashi\&Sangiorgi CAV 2008] [Caires CALCO 2007]

- Dependent behavioral types?
Conclusion

♦ Behavioral types may be used as interfaces for concurrent components

♦ Appropriate behavioral types should be chosen, depending on the definition of “composability”

♦ Type checking algorithms may be used as interface checking algorithms, with some adjustments:
  - restriction of the language of interfaces
  - integration with other verification methods