Behavioral Types as Interfaces for Concurrent Processes

Naoki Kobayashi
Tohoku University

What's This Talk About?

- Review of behavioral types for π-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 |= I — type checking
    - Subinterface: I < I' — subtyping

Types as Interfaces for *Concurrent* Components?

- Abstract specifications of components
  - Abstract enough to hide implementation details — yes
  - Precise enough to determine composability of components — ??
  - Equipped with algorithms for:
    - Interface conformance: P |= I — type checking
    - Subinterface: I < I' — subtyping

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 |= I — type checking?
    - Subinterface: I < I' — subtyping?

Outline

- π-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 π-calculus?

- Consists of basic concurrency primitives

parallel composition

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

Channel creation

Message send

Message reception

$\rightarrow$ r![1]

What is π-calculus?

- Consists of basic concurrency primitives

```
new r in   c![1,r] | 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 π-calculus

P, Q (Processes) ::=

0	(inaction)
new x in P	(channel creation)
x!v.P	(output)
x?y.P	(input)
P|Q	(parallel execution)
if b then P else Q	(conditional)
*P	(replication)

$x!v.P \mid x?y.Q \rightarrow P \mid [v/y]Q$

(c.f. $(\lambda x.M)N \rightarrow [N/x]M$)

Outline

- π-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

$U ::= 0$	not used
$?.U$	used for input, and then as U
$!.U$	used for output, and then as U
$U_1 \mid U_2$	used as U_1 and U_2 in parallel
$U_1 \& U_2$	used as U_1 or U_2
$\mu a.U$	recursion
$*U$	used as U arbitrarily many times

Type Judgment

$x_1: \tau_1, \ldots, x_n: \tau_n \vdash P$

P uses each x_i according to τ_i

Example:

✓ x:Chan(Int, !) |– x![1]

✗ x:Chan(Int, !), b:bool |–
if b then x![1] else 0

✓ ping: Chan(Chan(Int,!), ?) |– ping?[r].r![1]

Typing Rules

$$\frac{\Gamma,\ y{:}\tau,\ x{:}Chan(\tau,\ U)\ \vdash\ P}{\Gamma,\ x{:}\ Chan(\tau,\ ?.U)\ \vdash\ x?[y].P}$$

$$\frac{\Gamma\ \vdash\ P \qquad \Delta\ \vdash\ Q}{\Gamma\ |\ \Delta\ \vdash\ P\ |\ Q}$$

Type Environments as Interfaces

- Ir[...]kable (f[...]w)

An integer will be returned for each request

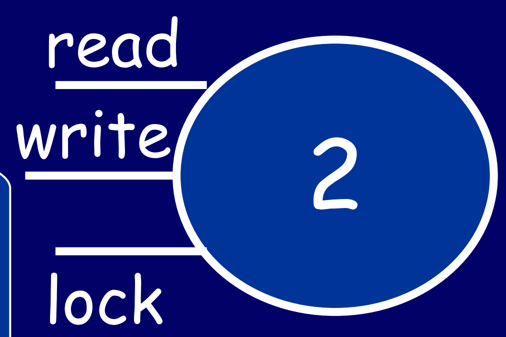
read request can be sent arbitrarily many times

read: (Int Chan(!)) Chan(*!),

write: (Int × Unit Chan(!)) Chan(*!)

lock: Unit Chan(*?.!)

Can be locked arbitrarily many times. Must be unlocked after each locking



Outline

- π-calculus
- Channel us
- 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:! ⊢ a?x.b!x

b:?, a:! ⊢ b?x.a!x

(a:?, b:!) | (b:?, a:!)

Why Extensions?

- Usage types are not precise enough to determine composability

a:?, b:! ⊢ a?x.b!x

b:?, a:! ⊢ b?x.a!x

a:?|!, b: !|? ✓

a?x.b!x | b?x.a!x ✘ deadlock!

Capability/Obligation Levels [Kobayashi 05, Acta Informatica]

- Annotations for usages ! and ?
  - e.g. $?_{(0,\ \infty)}$
- 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

Obligation Level

a:?(0,-), b:!(-,1) |– a?x.b!x

b:?(0,-), a:!(-,1) |– b?x.a!x

Assuming the success of input on a,
b will be used for output

Capability/Obligation levels: Example

Capability Level

Obligation Level

a:?(0,-), b:!(-,1) |– a?x.b!x

b:?(0,-), a:!(-,1) |– b?x.a!x

Assuming the success of input on b, a will be used for output

Capability/Obligation levels: Example

Capability Level

Obligation Level

a:?(0,-), b:!(-,1) ⊢ a?x.b!x

b:?(0,-), a:!(-,1) ⊢ 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

Obligation Level

a:?(0,-), b:!(-,1) |– a?x.b!x

b:?(1,-), a:!(-,0) |– a!1.b?x

a: ?(0,-) | !(-, 0)
b: !(-, 1) | ?(1, -) ✓

a?x.b!x | a!1.b?x ✓

Capability (or, assumption) is met by obligation (or, guarantee)

CCS Processes as Types

[Igarashi&Kobayashi POPL01][Chaki et al. POPL 02]

a?.b! ⊢ a?x.b!x

b?.a! ⊢ b?x.a!x

a?.b! | b?.a?

✘ deadlock!

Outline

- π-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 — yes
  - Precise enough to determine composability of components — probably yes (depending on "composability")
  - Equipped with algorithms for:
    - Interface conformance: P |= I — type checking?
    - Subinterface: I < I' — 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	not used
	$?.U$	used for input, and then as U
	$!.U$	used for output, and then as U
	$U_1 \mid U_2$	used as U_1 and U_2 in parallel
	$U_1 \& U_2$	used as U_1 or U_2
	$\mu a.U$	recursion
	$*U$	used as U arbitrarily many times

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

$\Rightarrow$ 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

  a: (?.!)&0, n:Int ⊢✗
  (if n>0 then a?x else 0);
  (if n>0 then a!x else 0)

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