Behavioral Type Systems for Program Analysis: A Tutorial

Naoki Kobayashi
University of Tokyo

What's This Talk About?

- Behavioral type systems for reasoning about precise properties of programs
  - deadlock, resource access protocols, ...
- From behavioral types to higher-order model checking

Disclaimer:
No new results, not a general survey (rather, mainly about our own previous work)

Outline

- Type-based program analysis: general principles
- Types for pi-calculus
  - Syntax and desired properties
  - Demonstration of TyPiCal
  - Channel usage types
  - Types for deadlock-freedom
  - Further extensions
- Types for resource usage verification
- From behavioral types to higher-order model checking
- Conclusion

Type-Based Program Analysis?

- Program analysis formalized in the form of type inference
  - Types as abstract properties of a program
  - Type judgment as a relation between a program and its abstract properties
  - Type inference algorithm as an algorithm for inferring abstract properties of a program

Behavioral type systems as special instances, where:

- temporal properties of programs (e.g. deadlock, resource access order, ...) are of interest
- types are abstract programs

Outline

- Type-based program analysis: general principles
- Types for pi-calculus
  - Syntax and desired properties
  - Demonstration of TyPiCal
  - Channel usage types
  - Types for deadlock-freedom
  - Further extensions
- Types for resource usage verification
- From behavioral types to higher-order model checking
- Conclusion

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

Example: Function Server

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

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

*succ?[n, r].r![n+1]|succ![1,rep] |rep?[m].print![m]

server

client

→*succ?[n, r].r![n+1] | rep![2] | rep?[m].print![m]

→*succ?[n, r].r![n+1] | print![2]

Example: Lock

- Unlocked state = presence of a value

  Locked state = lack of a value

  - lock creation: new lock in (lock![ ] | …)
  - lock acquisition: lock?[]....
  - lock release: lock![]

lock![ ] | lock?[ ].⟨CS1⟩.lock![ ] | lock?[ ].⟨CS2⟩.lock![ ]

→ lock?[ ].⟨CS1⟩.lock![ ] | ⟨CS2⟩.lock![ ]

→ lock?[ ].⟨CS1⟩.lock![ ] | lock![ ]

→⟨CS1⟩.lock![ ]

→lock![ ]

Desired Properties

- A server is always listening to clients' requests.
- A server will eventually send a reply for each request.
  - *ping?[r].if b then r![1] else r![2]
  - *ping?[r].if b then 0 else r![1]
- A process can eventually acquire a lock.
- An acquired lock will be eventually released.

Desired Properties

- A server is always listening to clients' requests.
- A server will eventually send a reply for each request.

  *ping?[r].if b then r![1] else r![2]

  *ping?[r].if b then 0 else r![1]

- A process can eventually acquire a

Certain communications should eventually succeed.

Outline

- Type-based program analysis: general principles
- Types for pi-calculus
  - Syntax and desired properties
  - Demonstration of TyPiCal
  - Channel usage types
  - Types for deadlock-freedom
  - Further extensions
- Types for resource usage verification
- From behavioral types to higher-order model checking
- Conclusion

TyPiCal [K. 2004]

- Deadlock-freedom analysis
  - find communications that will eventually succeed unless the process diverges

Input:

```
if b then x!1 else O  | x?n.print!n
```

Output of analysis:

```
if b then x!1 else O  | x?n.print!n
```

Will succeed
(if it is executed)

May not succeed

TyPiCal

- Deadlock-freedom analysis
  - find communications that will eventually succeed unless the process diverges
- Lock-freedom analysis
  - find communications that will eventually succeed (even in the presence of divergence)
- Termination analysis
- Information flow analysis
- Useless-code elimination

Outline

- Type-based program analysis: general principles
- Types for pi-calculus
  - Syntax and desired properties
  - Demonstration of TyPiCal
  - Simple Types
  - Channel usage types
  - Types for deadlock-freedom
  - CCS types
- Types for resource usage verification
- From behavioral types to higher-order model checking
- Conclusion

Simple Channel Types

(cf. CML, Milner's sorting)

τ chan the type of a channel used for sending/receiving a value of type τ

*ping?[r:int chan].r![“abc”]

*ping?[r:int chan].r![1]

*ping?[r:int chan].if b then 0 else r![1]

Outline

- Type-based program analysis: general principles
- Types for pi-calculus
  - Syntax and desired properties
  - Demonstration of TyPiCal
  - Simple Types
  - Channel usage types
  - Types for deadlock-freedom
  - CCS types
- Types for resource usage verification
- From behavioral types to higher-order model checking
- Conclusion

Channel Types with Usages

τ chan(U) the type of a channel used for sending/receiving a value of type τ according to usage U

*ping?[r:int chan(!)].r!["abc"] ✗

*ping?[r:int chan(!)].r![1] ✓

*ping?[r:int chan(!)].if b then 0 else r![1] ✗

Should be used once for output

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
a	variable

Channel Usage: Example

Server-client connection:

$$\mu a.(?.a) \mid *!$$

Server must be always listening to requests

Client can send requests arbitrarily many times

Reply channel:

$$! \mid ?$$

Lock channel:

$$! \mid *?.!$$

Lock is released first

Lock should be released each time it is acquired

Example: Lock

Should be used as a lock channel

```
newLock?[lock:unit chan(*?.!)].
      lock?[].⟨CS⟩.lock![ ]

newLock?[lock:unit chan(*?.!)].
      lock?[ ]. ⟨CS⟩.if b then 0 else lock![ ]

newLock?[lock: unit chan(*?.!) ].
      lock?[ ]. ⟨CS⟩.(lock![ ] | lock![ ] )
```

Type Judgment

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

P uses each x_i according to τ_i

Example:

- ✓ x:Int Chan(!) |– x![1]
- ✗ x:Int Chan(!), b:bool |–
  if b then x![1] else 0
- ✓ ping: (Int Chan(!)) Chan(?) |– ping?[r].r![1]

Typing Rules

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

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

Outline

- Type-based program analysis: general principles
- Types for pi-calculus
  - Syntax and desired properties
  - Demonstration of TyPiCal
  - Channel usage types
  - Types for deadlock-freedom
  - Further extensions
- Types for resource usage verification
- From behavioral types to higher-order model checking
- Conclusion

Deadlock and Channel Usage

- Channel usage is useful for finding a possibility of deadlock

  if b then x!1 else 0 | x?n.print!n

  x's usage: (!&0) | ?

  ⇒ The input on x may not succeed

- Channel usage is not sufficient for concluding the absence of deadlock

  x?n.y!n | y?m.x!m

  x and y's usages are !|?
  but the inputs on x and y never succeed!
  ⇒ Capability/obligation levels

Types for deadlock-freedom: Key Ideas

- Channel Usage
  - captures channel-wise communication behavior (mode/linearity/causality)
    - For each input action, is there a corresponding output action?
- Capability/Obligation Levels
  - capture inter-channel causality information
    - Is there any cyclic dependency that causes a deadlock?

Capability/Obligation
[Sumii& Kobayashi HLCL98]

- Capability (to successfully complete an action)
- Obligation (to offer an action)

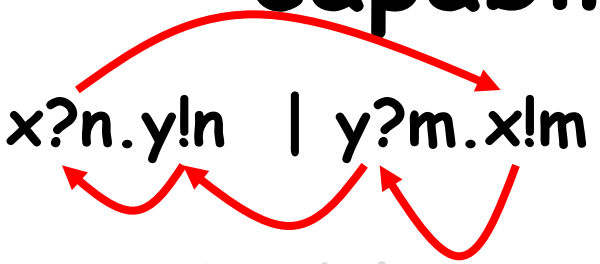
Example: server-client

- Server has an obligation to receive a request, and send a reply back to the client.
- Client has a capability to send a request, and receive a reply.

Example: lock

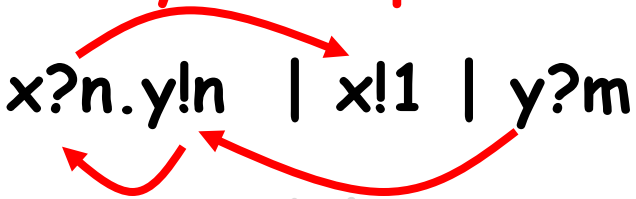
- A process has a capability to acquire a lock.
- The process then has an obligation to release the lock.

Dependencies between capabilities/obligations



Capability on y? depends on obligation on y!
Obligation on y! depends on capability on x?
Capability on x? depends on obligation on x!
Obligation on x! depends on capability on y?

Cyclic dependency on capabilities and obligations!



Capability on y? depends on obligation on y!
Obligation on y! depends on capability on x?
Capability on x? depends on obligation on x!

No cyclic dependency

Capability/Obligation Levels

- Associated with usages

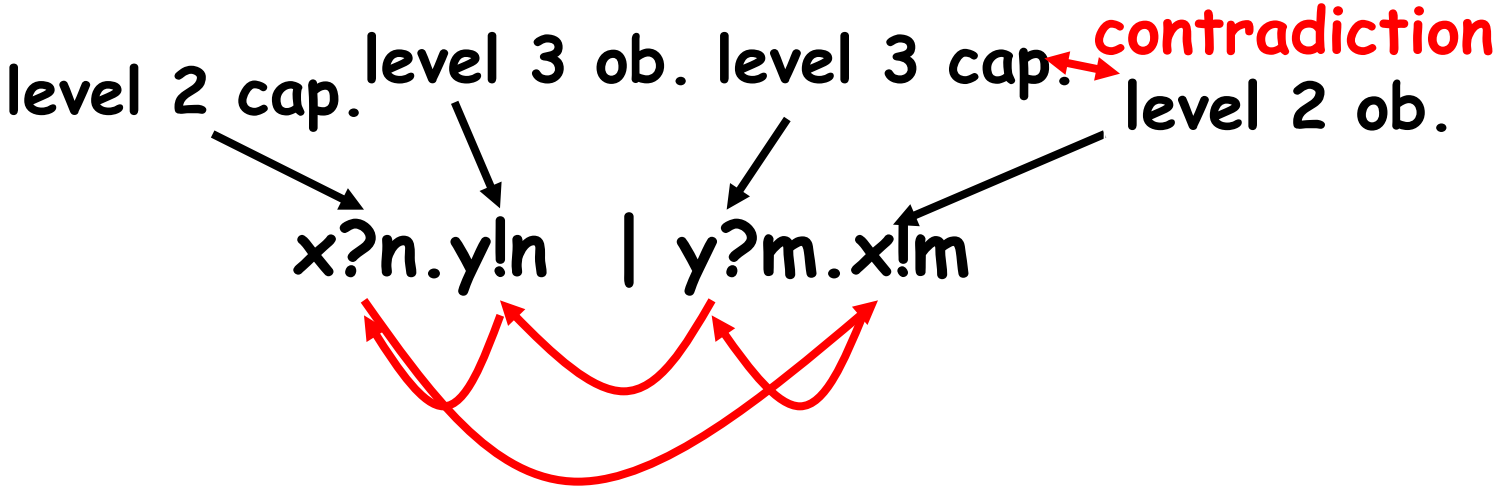
U ::= ?(o,c).U | !(o,c).U | ...

o,c ::= 0 | 1 | 2 |...|∞

absence of capability/obligation

- Prevent cyclic dependency between capabilities/obligations
  - An obligation must be fulfilled by using only capabilities of lower level
  - A capability must be matched by a corresponding obligation of the same level (for the co-action)

Examples revisited



No way to assign finite level capabilities to x? and y?

- An obligation must be fulfilled by using only capabilities of lower levels
- If a process has a capability for an action, the environment has an obligation of the same level for the co-action

Typing Rules for Deadlock-Freedom

$$\frac{\Gamma, y{:}\ \tau, x{:}\ \tau\,\mathsf{chan}(\mathsf{U}) \vdash P}{x{:}\ \tau\,\mathsf{chan}(?(0,c).\mathsf{U}),\ \uparrow^{c+1}\Gamma \vdash x?^{c}y.P}$$

$\uparrow^{c+1}(z{:}\ \sigma\ \mathsf{chan}(!(o_2,c_2).\mathsf{U}))$

=

$z{:}\ \sigma\ \mathsf{chan}(!(\mathsf{max}(c+1, o_2),\ c_2).\mathsf{U})$

Typing Rules for Deadlock-Freedom

$$\frac{\Gamma, y{:}\ \tau, x{:}\ \tau\,\mathsf{chan}(U) \vdash P}{x{:}\ \tau\,\mathsf{chan}(?(0,c).U), \uparrow^{c+1}\Gamma \vdash x?^{c}y.P}$$

$$\frac{\Gamma, x{:}\ \tau\,\mathsf{chan}(U) \vdash P}{x{:}\ \tau\,\mathsf{chan}(!(0,c).U), \uparrow^{c+1}(\Gamma \mid y{:}\tau) \vdash x!^{c}y.P}$$

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

$$\frac{\Gamma, x{:}\ \mathsf{chan}(\tau, U) \vdash P \qquad U \text{ is consistent}}{\Gamma \vdash \mathsf{new}\ x\ \mathsf{in}\ P}$$

Outline

- Type-based program analysis: general principles
- Types for pi-calculus
  - Syntax and desired properties
  - Demonstration of TyPiCal
  - Channel usage types
  - Types for deadlock-freedom
  - Further extensions
- Types for resource usage verification
- From behavioral types to higher-order model checking
- Conclusion

Usage with Varying Argument Types

(cf. session types [Takeuchi, Honda&Kubo, 94])

τ ::= b | chan(U)

U ::= 0	not used
$?[\tau].U$	used for receiving a value of type τ, and then as U
$![\tau].U$	used for sending a value of type τ, 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

Usage with Varying Argument Types

(cf. session types [Takeuchi, Honda&Kubo, 94])

τ ::= b | chan(U)

U ::= 0 not used

?[τ].U used for receiving a value of type τ, and then as U

![τ].U used for sending a value of type τ,

Example:

x: chan(?[bool].![int]) |-

x?[b].if b then x![1] else x![2]

Typing Rules

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

$$\frac{\Gamma,\ x{:}\mathrm{Chan}(U) \vdash P \qquad \tau_1 \leq \tau_2 \text{ if } U \rightarrow^* ![\tau_1].U_1 \mid ?[\tau_2].U_2 \mid U_3}{\Gamma \vdash \text{new } x \text{ in } P}$$

Typing Rules

Examples
- a bad usage:
  ![int].![bool] | ?[int].?[int]
- a good usage:
  ![int] | *?[int].![int]

$$\frac{\Gamma,\ x{:}Chan(U) \vdash P \qquad \tau_1 \leq \tau_2 \text{ if } U \rightarrow^* ![\tau_1].U_1 \mid ?[\tau_2].U_2 \mid U_3}{\Gamma \vdash \text{new } x \text{ in } P}$$

Usage with Varying Argument Types

(cf. session types [Takeuchi, Honda&Kubo, 94])

τ ::= b | chan(U)

U ::= 0	not used
$?[\tau].U$	used for receiving a value of type τ, and then as U
$![\tau].U$	used for sending a value of type τ, 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

Usage with Varying Argument Types

(cf. session types [Takeuchi, Honda&Kubo, 94])

τ ::= b | chan(U)

U ::= 0 not used

?[τ].U used for receiving a value of type τ, and then as U

![τ].U used for sending a value of type τ,

Session types
≈ Extended usage + labeled branch
+ restriction on parallel composition

CCS Types

[Igarashi&K. 2001, Chaki et al. 2002]

τ ::= b | $(x1,\ldots,xn)\Gamma$

Γ ::= 0 not used

$x?[\tau].\Gamma$ used for receiving a value of type τ through x, and then as Γ

$x![\tau].\Gamma$ used for sending a value of type τ through x, and then as Γ

$\Gamma_1 \mid \Gamma_2$ used as Γ_1 and Γ_2 in parallel

$\Gamma_1 \,\&\, \Gamma_2$ used as Γ_1 or Γ_2

$\Gamma_1 + \Gamma_2$ (external) choice

$\mu a.\Gamma$ recursion

CCS Types

[Igarashi&K. 2001, Chaki et al. 2002]

τ ::= b | $(x1,\ldots,xn)\Gamma$

Γ ::= 0 not used

x?[τ]. Γ used for receiving a value of type τ through x, and then as Γ

x![τ]. Γ used for sending a value of type τ through x, and then as Γ

Example:

x?[bool].y![int] |- x?[b].y![1]

r?[(x,y)x?[int].y![int]]
|- r?[w,z].w?n.z!(n+1)

Outline

- Type-based program analysis: general principles
- Types for pi-calculus
  - Syntax and desired properties
  - Demonstration of TyPiCal
  - Channel usage types
  - Types for deadlock-freedom
  - Further extensions
- Types for resource usage verification
- From behavioral types to higher-order model checking
- Conclusion

Resource Usage Verification for Functional Programs

[Igarashi&K. POPL02]

Is each resource (files, network, memory, ...) accessed in a valid manner?

```
let f(x) =
  if * then close(x)
  else read(x); f(x)
in
let y = open "foo"
in
     f (y)
```

Is the file "foo" accessed according to read* close?

Types

τ (types) ::=	b	base types
|	R(u)	resource types
|	$\tau_1 \to \tau_2$	function types
|	$\tau \times \tau$	
u (usages) ::=	0	cannot be used
|	a	accessed once by use$_a$
|	$u_1; u_2$	u_1 and then u_2
|	$u_1 \& u_2$	u_1 or u_2
|	ρ	usage variable
|	$\mu\rho.\, u$	recursion

Examples: usages

- $\mu\rho.(c \,\&\, (r;\, \rho))$: read-only file

- $\mu\rho.(0 \,\&\, (push;\rho;\, pop))$: stack

u (usages) ::= 0 cannot be used

| a accessed once by use_a

| $u_1; u_2$ u_1 and then u_2

| $u_1 \& u_2$ u_1 or u_2

| ρ usage variable

| $\mu\rho.\, u$ recursion

Example

```
let f(x: R(r*c)) =
  if * then close(x)
  else read(x); f(x)
in
let y: R(r*c) = open "foo"
in
     f (y)
```

Typing: resources

$$\frac{\mathrm{sem}(u) \subseteq \Phi}{\vdash \mathrm{new}^{\Phi}() : R(u)}$$

$$\frac{\Gamma \vdash M : R(a)}{\Gamma \vdash \mathrm{use}_a\ M : \mathrm{unit}}$$

Typing : let

$$\frac{\Gamma \vdash M:\tau \quad \Delta, x:\tau \vdash N:\sigma \quad \mathbf{rfree}(\tau)}{\Gamma ; \Delta \vdash \mathbf{let}\ x=M\ \mathbf{in}\ N : \sigma}$$

(rfree(τ) if τ does not contain resource types)

Example:

$$\frac{y: R(r) \vdash \mathbf{read}(y):\mathbf{unit} \quad y: R(c),\ x: \mathbf{unit} \vdash \mathbf{close}(y):\mathbf{unit}}{y: R(r;c) \vdash \mathbf{let}\ x= \mathbf{read}(y)\ \mathbf{in}\ \mathbf{close}(y) : \mathbf{unit}}$$

Outline

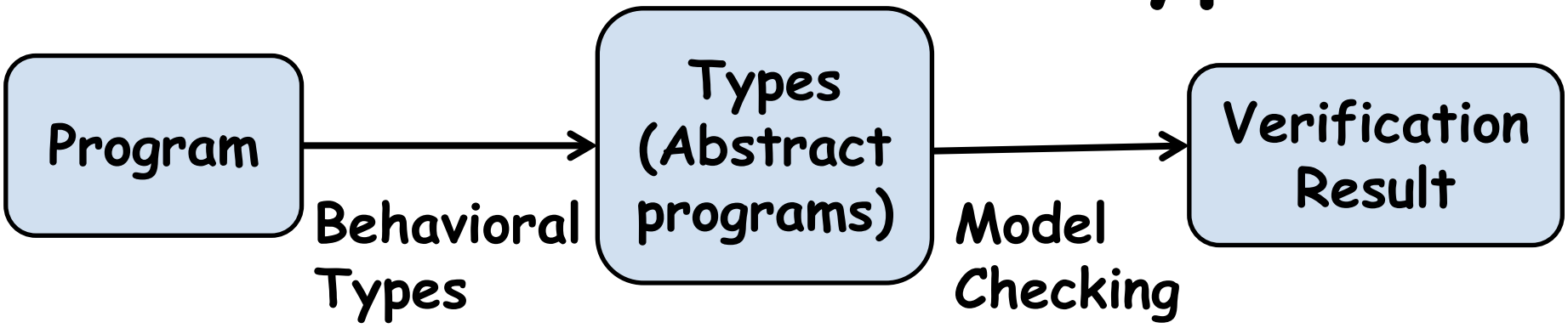
- Type-based program analysis: general principles
- Types for pi-calculus
  - Syntax and desired properties
  - Demonstration of TyPiCal
  - Channel usage types
  - Types for deadlock-freedom
  - Further extensions
- Types for resource usage verification
- From behavioral types to higher-order model checking
- Conclusion

Behavioral Types for Program Verification

- **Types** (or type environments) = abstract models of a program
- **Typing rules** establish a relation between a program and its abstraction
- **Type inference** computes an abstraction of a program
- **Extra work** (possibly **model checking**) is need to check that the abstraction (hence also the program) satisfies an expected property

$$\frac{\Gamma,\ x{:}Chan(U) \vdash P \qquad \tau_1 \leq \tau_2 \text{ whenever } U \rightarrow^* ![\tau_1].U_1 \mid ?[\tau_2].U_2 \mid U_3}{\Gamma \vdash \text{new } x \text{ in } P}$$

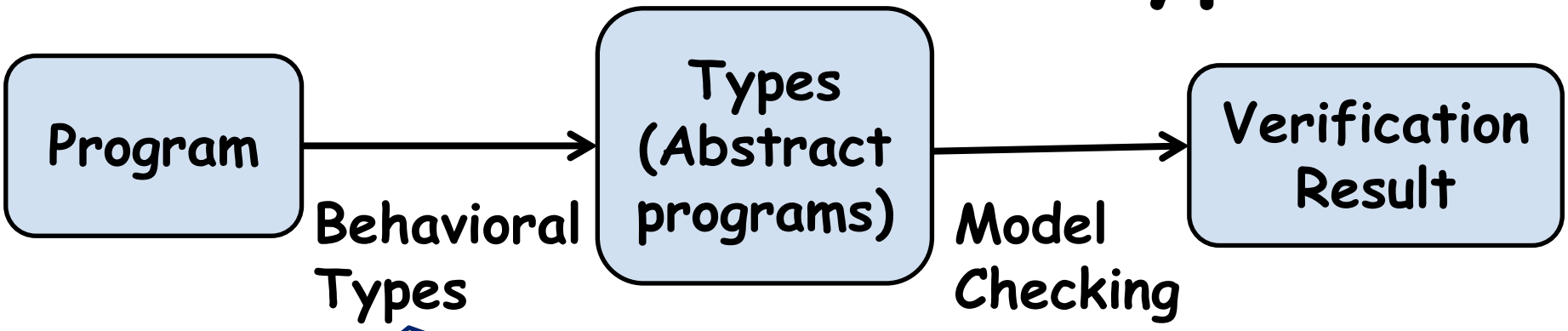
Role of Behavioral Types?



1. Map a program to a simpler model (e.g. CCS instead of π-calculus, first-order programs instead of higher-order ones)

2. Project the whole program behavior to each behavior of interest

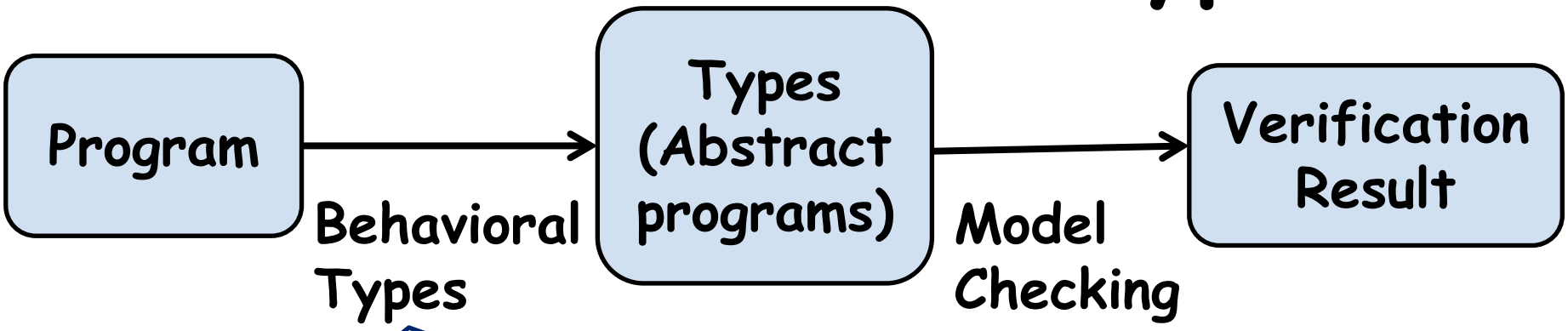
Role of Behavioral Types?



1. Map a program to a simpler model (e.g. CCS instead of π-calculus, first-order programs instead of higher-order ones)

   Is it essential to use *first-order* programs as abstract programs?

Role of Behavioral Types?



1. Map a program to a simpler model (e.g. CCS instead of π-calculus, first-order programs instead of higher-order ones)

   Is it essential to use *first-order* programs as abstract programs?
   No. (cf. Model checking of higher-order recursion schemes [Ong 2006])

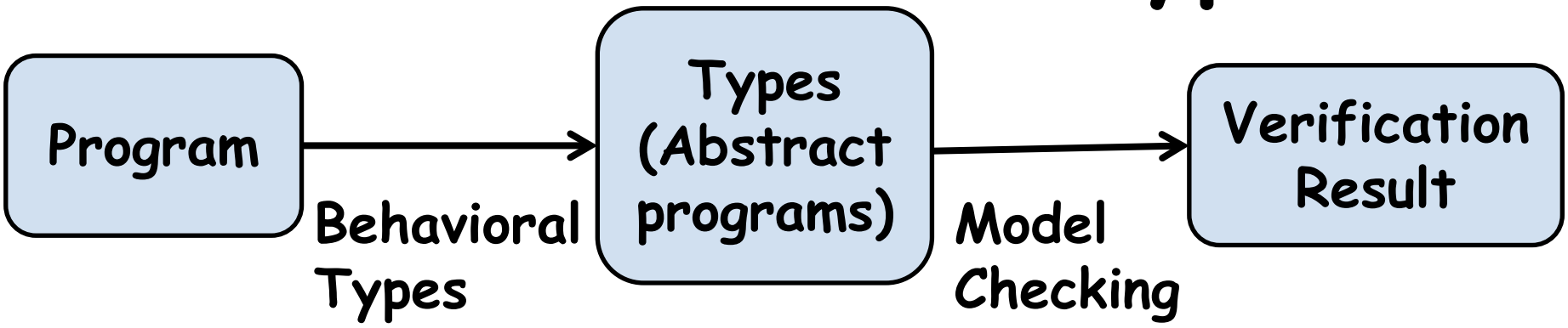
Higher-order programs as abstractions

```
let f(x, y) =
  if * then
     close(x); close(y)
  else
     read(x); write(y);
     f(x, y)
in
let x = new^{r*c}() in
let y = new^{w*c}() in
     f (x, y)
```

```
let f(x) =
  if * then close(x)
  else read(x); f(x) in
let x = new^{r*c}() in
     f (x)
```

```
let f(y) =
  if * then close(y)
  else write(y); f(y) in
let y = new^{w*c} () in
     f (y)
```

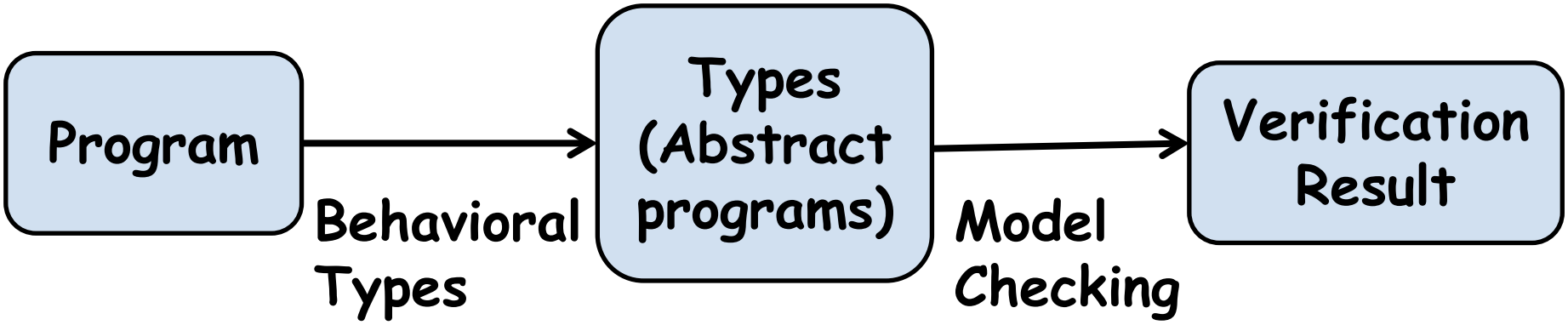
Role of Behavioral Types?

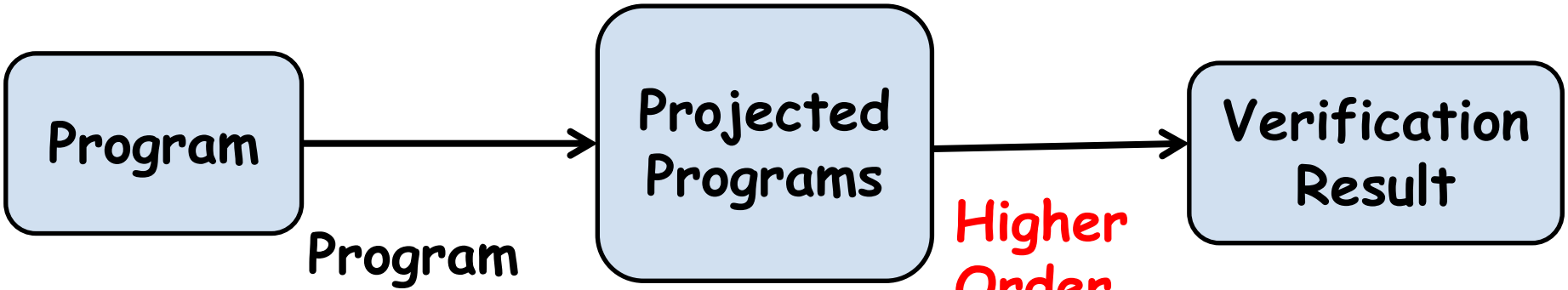


1. Map a program to a simpler model (e.g. CCS instead of π-calculus, first-order programs instead of higher-order ones)

2. Project the whole program behavior to each behavior of interest

Behavioral Types to HO Model Checking





Conclusion

- Tutorial on behavioral types for
  - channel usage
  - deadlock
  - resource access protocols
- From behavioral types to HO model checking