Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness

Naoki Kobayashi
University of Tokyo

Increasing Importance of Safety of Concurrent Programs

- **Why Safety?**
  - More programs exchanged via networks, which are
    - Executed before enough tests (e.g. applets)
    - Written by others who cannot be trusted
    - Hard to debug (The whole system cannot be tested)
- **Why Safety of Concurrent Programs?**
  - Inherent concurrency of distributed systems
  - More bugs introduced by concurrency
  - Safety of sequential programs subsumed

Type-Based Approach to Program Safety

- Successful for sequential programs
  - ML, Java, ...
  - TAL, Java bytecode, ...
  - exception/region analysis, information flow

⇒Similar approach to safety of concurrent programs? But, ...

✗ Conventional type systems are insufficient

e.g. CML: Evaluation of "functions" may get stuck

- fun f(n:int) = let val ch = channel() in accept(ch)+n end

val f = fn: int →int

⇒Need for more advanced type systems!

Previous Type Systems

- I/O channel types [Pierce&Sangiorgi 93]
  - Whether channels are used for input/output
- Linear channel types [Kobayashi, Pierce & Turner 96]
  - How often channels are used
- Deadlock-freedom (in a broad sense) [Yoshida 96; Kobayashi&Sumii 97,98; Puntigam 98]
  - Certain communications do not get blocked forever, unless the program diverges
- Race detection [Abadi&Flanagan 99, 2000]
- Information flow [Honda, Vasconcelos & Yoshida 2000; Hennessy & Riely 2000]
- ...

Deadlock-Freedom [Kobayashi 97, 98] Implies ...

- Evaluation never gets stuck (a là typed functional languages)
- A client's request is eventually accepted by a server, and a reply comes back.
  - lack of message-not-understood errors
- A process can eventually acquire a lock, and then it will release the lock.
- No process forgets to synchronize with other processes (e.g., via barrier synchronization primitives)

... Provided that the whole program does not diverge

This Talk

- **Type systems for stronger guarantees of the success of certain communications:**
  - **Deadlock-Freedom**
    - **Communications eventually succeed *unless divergence***
  - **Livelock-Freedom**
    - **Communications eventually succeed *under any fair scheduling***
  - **Time-Boundedness**
    - **Communications succeed *within a certain number of reduction steps***

All of them can be obtained in a similar manner

Outline

- Target Language (Pi-Calculus)
  - syntax, semantics
  - deadlock and livelock
  - examples
- Type Systems
- Current and Future Work

Target Language: Pi-Calculus

P, Q (Processes) ::=

new x in P	(channel creation)
$x!^{T}[\, v_1, ..., v_n \,].\, P$	(output)
$x?^{T}[\, y_1, ..., y_n \,].\, P$	(input)
$P \mid Q$	(parallel execution)
$*P$	(replication)
if v then P else Q	(conditional branch)

$S, T, ... \in$ Tags

Used to state properties of each input/output

e.g. "Any output with tag T succeeds eventually, but not those with tag S."

"Any output with tag T succeeds within 10 reduction steps"

Target Language: Pi-Calculus

P, Q (Processes) ::=

new x in P	(channel creation)
$x!^{T}[\, v_1, ..., v_n \,].\, P$	(output)
$x?^{T}[\, y_1, ..., y_n \,].\, P$	(input)
$P \mid Q$	(parallel execution)
$*P$	(replication)
if v then P else Q	(conditional branch)

$S, T, ... \in$ Tags

Reduction:

$$x![v_1, ..., v_n\,].P \mid x?[y_1, ..., y_n\,].Q \rightarrow P \mid [v_1/y_1, ..., v_n/y_n]Q$$

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

Example: Function Server

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

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

$$*succ?[n, r].r![n+1] \mid succ![1,rep] \mid rep?[m].print![m]$$

server: $*succ?[n, r].r![n+1]$ — client: $succ![1,rep] \mid rep?[m].print![m]$

$$\rightarrow *succ?[n, r].r![n+1] \mid rep![2] \mid rep?[m].print![m]$$

$$\rightarrow *succ?[n, r].r![n+1] \mid print![2]$$

Deadlock/Livelock-Freedom

- P is *deadlock-free on* T if:
  - Whenever $P \rightarrow^* Q$ and $Q \equiv$ new $y_1,...,y_n$ in $(x!^T[v].R_1 \mid R_2)$, $Q \rightarrow R$ for some R. (Similar for input)
- P is *livelock-free on* T if
  - Whenever $P \rightarrow^* Q$ and $Q \equiv$ new $y_1,...,y_n$ in $(x!^T[v].R_1 \mid R_2)$, the process $x!^T[v].R_1$ is eventually reduced in any fair reduction sequence. (Similar for input.)

Examples

- $x?^T[\,].y![\,] \mid y?\,[\,].x![\,]$: deadlocked and livelocked
- $x?^S[\,].y!^S[\,] \mid y?^S\,[\,].x!^S[\,]$: deadlock/livelock-free on T
- $x?^T[n].0 \mid x![1] \mid *x?[n].x![n]$: deadlock/livelock-free
- $x?^T[n].0 \mid r![x] \mid *r?[y].r![y]$: deadlock-free but livelocked

Example: Concurrent Object

- **Counter object:**

$$state![1]$$
$$| * inc?[r].state?[n].(state![n+1] \mid r![\,])$$
$$| * read?[r].state?[n].(state![n] \mid r![n])$$

- **A caller:**

  new r **in** $(inc!^{T}[r] \mid r?[\,]....)$

  The request is eventually accepted

  (≈ Lack of message-not-understood errors)

Example: Lock

- Unlocked state = presence of a value
  Locked state = lack of a value)

  creation: new x in $(x![\,] \mid \ldots)$

  acquisition and release : $x?^{T}[\,].(\ldots x![\,])$

Lock can be eventually acquired.

(No process forgets to release the lock.)

Example: Barrier

- **Barrier:**

  $sync?[\,].sync?[\,].(resume![\,] \mid resume![\,])$

- **Process:**

  $....sync![\,].resume?^{T}[\,]....$

  Synchronization succeeds eventually.
  (No other process forgets to synchronize.)

Outline

- Target Language
- Type Systems
  - Universal type system
  - Analysis of deadlock, livelock, and time-boundedness
  - Limitations and extensions
- Current and Future Work

Types

- **Extend channel types with information on how channels are used**

$\tau ::= b$ (base types)

$[\tau_1, ..., \tau_n]/U$ (type of channels used to transmit a tuple of values of types $\tau_1, ..., \tau_n$)

U (usages) ::=

0 (not used at all)

$I^T.U$ (used for input with tag T, and then as U)

$O^T.U$ (used for output with tag T, and then as U)

$U_1 \| U_2$ (used according to U_1 and U_2 in parallel)

$\mu\alpha.U$ (recursion: e.g., $\mu\alpha.I^T.\alpha$)

$<T>U$ (used as U after a communication with tag T)

$U_1 \& U_2$ (used according to either U_1 or U_2)

Examples (Type Judgment)

- x: [Int]/($O^{T1} \| I^{T2}$) , y: [Int]/($<T2>O^{T3}$)
  $\vdash x!^{T1}[3] \mid x?^{T2}[n].y!^{T3}[n]$

- y: [Int]/($<S>O^{T}$) $\vdash$ new x in $(x?^{S}[n].y!^{T}[n])$

- x : [[Int] /O^{S}]/ $I^{T} \vdash x?^{T}[r].r!^{S}[3]$

- f : [[Int] /$<T1>\mathrm{O}^{T2}$]/ $\mu\alpha.(\alpha \| I^{S})$ $\vdash P$

  P waits to receive a channel on f. Then, it waits for an event $T1$ to happen and sends an integer through the received channel.

Typing Rule: Input

$$\frac{\Gamma,\ x{:}[\tau]/U,\ y{:}\,\tau\ \vdash P}{<T>\Gamma,\ x{:}[\tau]\ /I^{T}.U \vdash x?^{T}[y].P}$$

where

$$<T>(x_1{:}[\tau_1]/U_1, \ldots, x_n{:}[\tau_n]/U_n)$$
$$= x_1{:}[\tau_1]/<T>U_1, \ldots, x_n{:}[\tau_n]/<T>U_n$$

Typing Rule: Parallel Composition

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

where

$$(x_1{:}[\tau_1]/U_1, \ldots, x_n{:}[\tau_n]/U_n) \,\|\, (x_1{:}[\tau_1]/V_1, \ldots, x_n{:}[\tau_n]/V_n)$$
$$= x_1{:}[\tau_1]/(U_1 \,\|\, V_1), \ldots, x_n{:}[\tau_n]/(U_n \,\|\, V_n)$$

Type Preservation

- **Reduction of type environment**
  - $\Gamma, x\text{: } [\tau]/\ (I^S.U \parallel O^T.V \parallel W) \rightarrow \Gamma, x\text{: } [\tau]/\ (U \parallel V \parallel W)$
  - $\Gamma, x\text{: } [\tau]/\ (<T>U) \rightarrow \Gamma, x\text{: } [\tau]/\ U$
- **Subject Reduction**

$$\begin{array}{ccc} \Gamma & \vdash & P \\ \downarrow & & \downarrow \\ \exists\Delta & \vdash & Q \end{array}$$

Type environment simulates the process behavior.

Type Preservation: Example

x: [Int]/(O^{T1}|| I^{T2}) ,

y: [Int]/(<$T2$>O^{T3}) $\vdash$ $x!^{T1}[3] \mid x?^{T2}[n].y!^{T3}[n]$

x: [Int]/0 , y: [Int]/O^{T3} $\vdash$ $y!^{T3}[3]$

Common Principles

- $\Gamma \vdash P$ means
  "Γ simulates the behavior of P"

$\Rightarrow$ ***To check properties of P, check the corresponding properties of Γ***

Analysis of Deadlock

$$P \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \quad \textit{deadlock}$$

$$\perp \qquad \perp \qquad \perp \qquad\qquad \perp$$

$$\Gamma \rightarrow \Gamma_1 \rightarrow \Gamma_2 \rightarrow \ldots \rightarrow \Gamma_n \quad \textit{deadlock}$$

If $\Gamma \vdash P$ and Γ is *deadlock-free*,

then P is deadlock-free!

(Actually, we need to check the type bindings of channels created inside P as well)

Analysis of Deadlock: Example

$x?^{T1}[n].y!^{T2}[n] \mid y?^{T3}[n].x!^{T4}[n]$

Its type environment:

x: [Int]/($I^{T1} \| <T3>O^{T4}$) , y: [Int]/($<T1>\mathrm{O}^{T2} \| I^{T3}$)

says that the process may be deadlocked on $T1$

- **The usage** $I^{T1} \| <T3>O^{T4}$ **of** x **implies ...**

  An event with tag $T3$ **must precede the input with tag** $T1$

- **The usage** $<T1>\mathrm{O}^{T2} \| I^{T3}$ **of** y **implies ...**

  $T1$ **must precede** $T3$

"Deadlock-Freedom" of Type Environment

- "Γ *is deadlock-free on* T" if there exists a *well-founded relation* $<$ on tags such that
  - If $\Gamma \rightarrow^* (\Delta, x{:}[\tau]/\ (O^{T'}.U \,\|V))$ and $T' \leq T$, then
    - $V = I^S.V_1 \,\|\, V_2$, or
    - $V = \langle S \rangle V_1 \,\|\, V_2$ *and* $S < T'$
  - Similarly for the input case.

"$S<T$*" :There may be a causal order* $S \rightarrow T$

Analysis of Livelock

$$P \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \rightarrow \ldots \textit{ livelock}$$

$$\perp \quad \perp \quad \perp \quad \perp$$

$$\Gamma \rightarrow \Gamma_1 \rightarrow \Gamma_2 \rightarrow \ldots \rightarrow \Gamma_n \rightarrow \ldots \textit{ livelock}$$

If $\Gamma \vdash P$ and Γ is livelock-free,
then P is livelock-free!

"Livelock-Freedom" of Type Environment

- "Γ *is livelock-free on* T" (Γ does not livelock on an event T) if there exists a *well-founded relation* $<$ on tags such that
  - If $\Gamma \rightarrow^* (\Delta, x{:}[\tau]/ (O^{T'}.U \,||V))$ and $T' \leq T$, then
    - $V = \langle S_1 \rangle ... \langle S_n \rangle \, I^{T''}.W_1 \,||\, W_2$ *and* $S_1, ..., S_n < T'$
  - Similar for the input case

Analysis of Time-Boundedness

$$x?^{T}[y].Q|R \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_T \quad \textit{time-out}$$

$$\bot \qquad \bot \qquad \bot \qquad \bot$$

$$\Gamma \quad \rightarrow \Gamma_1 \rightarrow \Gamma_2 \rightarrow \ldots \rightarrow \Gamma_T \quad \textit{time-out}$$

T: The number of parallel reduction steps required before the input succeeds

If $\Gamma \vdash P$ and Γ is time-bounded,
then P is time-bounded!

Time-Boundedness

- **"Timed" usage reduction**

  e.g. $I^{S1}.U \parallel O^{S2}.V \parallel O^{T}.W \rightarrow U \parallel V \parallel O^{T-1}.W$

- **"Timed reduction" of type environment**

  - $x_1{:}[\tau_1]/U_1, \ldots, x_n{:}[\tau_n]/U_n \rightarrow x_1{:}[\tau_1]/V_1, \ldots, x_n{:}[\tau_n]/V_n$
    if $U_i \rightarrow V_i$ **for each** i

- **"Γ is time-bounded" if,**

  - $\Gamma \rightarrow^* (\Delta, x{:}[\tau]/ (O^0.U \parallel V))$ **implies**
    $V = I^{T'}.W_1 \parallel W_2$ **with** $W_2 \neq O^T.W_3 \parallel W_4$
  - **Similar for the input case**

Variants

- Analysis of deadlock/livelock/time-boundedness of a process under:
  - FIFO scheduling, or
  - Priority scheduling

$\Rightarrow$Check how the process's type environment is reduced under the corresponding scheduling policy

Limitations (1/2)

- Cannot express value-dependent behavior

  e.g. $fact$: $[Nat, [Nat]/<T>O]/ \mu\alpha.(\alpha \parallel I^S)$

  $\vdash * fact?[n,r].$

  (if $n=0$ then $r![1]$

  else new s in $fact![n-1,s]|s?[m].r![m \times n]$)

  If T expresses a time-bound, it depends on n.

$\Rightarrow$Use *dependent types*

$fact$: $[\Sigma n{:}Nat.[Nat]/<2 \times n>O]/ \mu\alpha.(\alpha \parallel I^S)$

Limitations(2/2)

- Cannot express ordering of communications on different channels

  e.g. One-place buffer with an interface [*put*, *get*]

  Services *put* and *get* are available only alternately.

  ✘ *put*: $[\tau]/\mu\alpha.I^S.<T>\alpha$, *get*: $[\sigma]/\mu\alpha.<S>I^T.\alpha$

⇒ Extend type environments and use CCS-like process terms

$\mu\alpha.put?^S[\tau].get?^T[\sigma].\alpha$

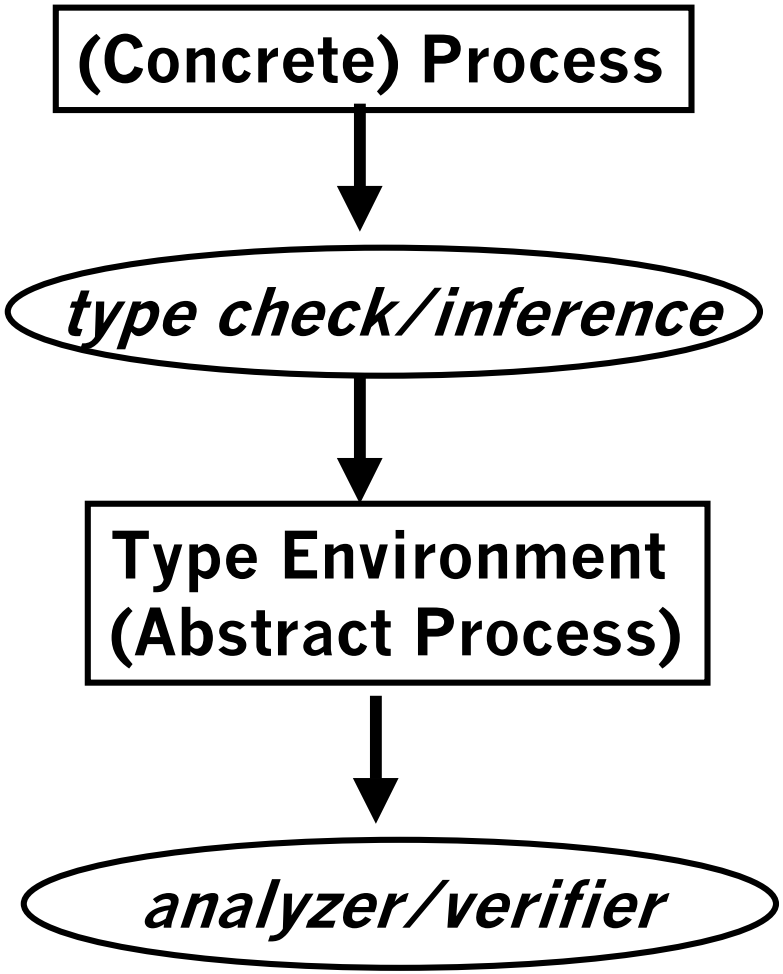
Conclusion

- **The safe usage of various construct for concurrent programming can be enforced uniformly via guarantees of**
  - **deadlock-freedom**
  - **livelock-freedom**
  - **time-boundedness**

  of pi-calculus processes
- **Type systems for those properties can be obtained based on the common principle:**

  "type environments as abstract processes"

General View of Our Approach



Current and Future Work

- A *generic* type system for pi-calculus, based on "type environments as abstract processes"
  - Various type systems obtained as instances
  - (Core of) type soundness proofs and type check/inference algorithms are shared.
- Applications to:
  - Concurrent languages (CML, Pict, Java,..)
  - Lower-level language (bytecode or assembly language with concurrency primitives)

Many engineering issues are left.