A Generalized Deadlock-Free Process Calculus

Eijiro Sumii
Naoki Kobayashi

University of Tokyo

Merit and Demerit of Concurrent Languages

Compared with sequential languages...

- Merit: more expressive power
  - Inherently concurrent application (e.g. GUI)
  - Parallel/distributed computation

Merit and Demerit of Concurrent Languages

Compared with sequential languages...

- Merit: more expressive power
  - Inherently concurrent application (e.g. GUI)
  - Parallel/distributed computation
- Demerit: more complicated behavior
  - Non-determinism (possibility of various results)
  - Deadlock (failure of due communication)

Merit and Demerit of Concurrent Languages

Compared with sequential languages...

- Merit: more expressive power
  - Inherently concurrent application (e.g. GUI)
  - Parallel/distributed computation
- Demerit: more complicated behavior
  - Non-determinism (possibility of various results)
  - Deadlock (failure of due communication)

➡ **Errors & inefficiencies**

Example of Complication (1/2)

In ML:

$\texttt{f : int->int} \vdash \texttt{f(3) : int}$

- eventually returns a unique result
  (unless ‘infinite loop’ or ‘side effect’)

Example of Complication (1/2)

In CML:

$\texttt{f : int->int} \vdash \texttt{f(3) : int}$

- may return:
  - different results in parallel (→ non-determinism

```
fun f(i) =
  let
    val c : int chan = channel()
  in
    (spawn(fn () => send(c, i + 1));
     spawn(fn () => send(c, i + 2));
     recv(c))
```

Example of Complication (1/2)

In CML:

$$\texttt{f : int->int} \vdash \texttt{f(3) : int}$$

- may return:
  - different results in parallel (→ non-determinism
  - no result at all (→ deadlock)

```
fun f(i) =
  let
    val c : int chan = channel()
  in
    recv(c)
  end
```

Example of Complication (2/2)

Mutex channel `m : unit chan`

- correct use:
  - receive once, send once

`recv(m);` ***CriticalSection*** `; send(m, ())`

Example of Complication (2/2)

Mutex channel `m : unit chan`

- correct use:
  - receive once, send once
- incorrect use:
  - receive once, send never (→ deadlock)

```
recv(m); CS; ()
```

  - receive once, send twice (→ non-determinism)

```
recv(m); CS; send(m, ()); send(m, ())
```

Example of Complication (2/2)

Mutex channel `m, n : unit chan`

- correct use:
  - receive once, send once
- incorrect use:
  - receive once, send never (→ deadlock)
  - receive once, send twice (→ non-determinism)
  - use in various order (→ deadlock)

```
spawn(fn () => recv(m); recv(n); …);
  spawn(fn () => recv(n); recv(m); …)
```

Possible Approaches

- Provide higher-level constructs

  e.g.:
  - parallel functions
  - binary semaphores
  - concurrent objects

Possible Approaches

- Provide higher-level constructs

  e.g.:
  - parallel functions
  - binary semaphores
  - concurrent objects

  ✗ “chaos” outside them

  ✗ complicated syntax & semantics

Possible Approaches

- Provide higher-level constructs
  - ✗ “chaos” outside them
  - ✗ complicated syntax & semantics
- Enrich channel types:
  control communication
  with a static type system

Possible Approaches

- Provide higher-level constructs
  - ✗ “chaos” outside them
  - ✗ complicated syntax & semantics
- Enrich channel types:
  control communication
  with a static type system

$\Uparrow$

Our approach

Outline

- Introduction
- Basic Ideas
- The Type System
- Related Work
- Conclusion

Target Language

Asynchronous variant of Milner’s π-calculus

`new x in P`	**(channel creation)**
`x![y]`	**(output)**
`x?[y].P`	**(input)**
`P | Q`	**(parallel execution)**
`def x[y]=P in Q`	**(process definition)**
`if x then P else Q`	**(conditional branch)**

Target Language

Asynchronous variant of Milner’s π-calculus

`new x in P`	(channel creation)
`x![y]`	(output)
`x?[y].P`	(input)
`P	Q`	(parallel execution)
`def x[y]=P in Q`	(process definition)
`if x then P else Q`	(conditional branch)

- `x?[y].P | x![z]` ® `P{z/y}`
- `def x[y]=P in x![z]`
  ® `def x[y]=P in P{z/y}`

Outline

- Introduction
- Basic Ideas
  - Usages & Usage Calculus
    $\Rightarrow$ "In what way each channel may be used"
  - Time Tags & Time Tag Ordering
    $\Rightarrow$ "In what order those channels may be used"
- The Type System
- Related Work
- Conclusion

Usages (1/2): Input/Output

- *U* (usage) **:=**

O	(output)
I . *U*	(input + sequential execution)
U | *V*	(parallel execution)
Æ	(none)

✓ `x:[]/(O|I)` ⊢
`x![] | x?[]`

✗ `x:[]/(O|I)` ⊢
`x![] | x![] | x?[] | x?[]`

Usages (1/2): Input/Output

- *U* (usage) `:=`
  - `O` (output)
  - `I` . *U* (input + sequential execution)
  - *U* | *V* (parallel execution)
  - Æ (none)

✓ `y:[]/(O|O|I.I)` ⊢
`y![] | y![] | y?[].y?[]`

✗ `y:[]/(O|O|I.I)` ⊢
`y![] | y![] | y?[] | y?[]`

Usages (2/2): Obligation and Capability

- U (usage) **:=**
  - **O**$_a$ (output)
  - **I**$_a$ **.** U (input + sequential execution)
  - **…**
- a (attributes) **:=**
  - (none)
  - **o** (obligation: “***must*** be performed”)
  - **c** (capability: “***can*** be performed successfully”)
  - **co** (both)

Usages (2/2): Obligation and Capability

`x:[int]/Oo`

"***must*** send an integer value to **`x`**"

✓ `x:[int]/Oo` ⊢ `x![3]`

✗ `x:[int]/Oo` ⊢ `0`

Usages (2/2): Obligation and Capability

`y:[int]/Ic`

“***can*** receive an integer value from **`y`** successfully”

✓ $\mathtt{y{:}[int]/Ic} \vdash \mathtt{y?[v].0}$

$\Uparrow$

eventually reduces to **`0`**

(by communication with an external process)

✓ $\mathtt{y{:}[int]/Ic} \vdash \mathtt{0}$

Usages (2/2): Obligation and Capability

What to Ensure:

- An obligation must be fulfilled eventually
- A capability can be used successfully

$\Uparrow$

Otherwise “deadlock”

Reliability of Usages & the Usage Calculus

- `new x:[int]/Ic in x?[v].P`

Reliability of Usages & the Usage Calculus

✗ `new x:[int]/Ic in x?[v].P`

“For every **`I/O`** with capability,
a corresponding **`O/I`** with obligation”

✓
```
new x:[int]/(Ic|Oo)
          in (x?[v].P | x![3])
```

Reliability of Usages & the Usage Calculus

"For every **I/O** with capability,
a corresponding **O/I** with obligation"

```
✗ new x:[]/(Oo|Ic|Ic)
      in (x![] | x?[].P | x?[].Q)
  ® new x:[]/Ic in x?[].Q
```

Reliability of Usages & the Usage Calculus

"For every **`I/O`** with capability, a corresponding **`O/I`** with obligation"

```
✗  new x:[]/(Oo|Ic|Ic)
        in (x![] | x?[].P | x?[].Q)
  ® new x:[]/Ic in x?[].Q
```

```
Oo|Ic|Ic ® Ic
```

Outline

- Introduction
- Basic Ideas
  - Usages & Usage Calculus
    $\Rightarrow$ "In what way each channel may be used"
  - Time Tags & Time Tag Ordering
    $\Rightarrow$ "In what order those channels may be used"
- The Type System
- Related Work
- Conclusion

Dependency between Obligation and Capability

- ✓ `x:[int]/Oo` ⊢ `x![3]`
- ✗ `y:[]/I, x:[int]/Oo` ⊢ `y?[].x![3]`
- ✓ `y:[]/Ic, x:[int]/Oo` ⊢ `y?[].x![3]`

Dependency between Obligation and Capability

$t<s$

“a capability with **t** may be used before an obligation with **s** is fulfilled”

✓ $\mathtt{y:[]/Ic}^{t}\mathtt{,\ x:[int]/Oo}^{s}\mathtt{;}\ t<s \vdash \mathtt{y?[].x![3]}$

✗ $\mathtt{y:[]/Ic}^{t}\mathtt{,\ x:[int]/Oo}^{s}\mathtt{;}\ \text{Æ} \vdash \mathtt{y?[].x![3]}$

✗ $\mathtt{y:[]/Ic}^{t}\mathtt{,\ x:[int]/Oo}^{s}\mathtt{;}\ s<t \vdash \mathtt{y?[].x![3]}$

Preventing & Detecting Cycles in the Dependency

G = `c:[]/(Oo`^{s}`|Ic`^{s}`), d:[]/(Oo`^{t}`|Ic`^{t}`)`

✓ G; `s<t` $\vdash$ `c?[].d![] | ...`

✗ G; `s<t` $\vdash$ `d?[].c![] | ...`

Preventing & Detecting Cycles in the Dependency

G = `c:[]/(Oo`^{s}`|Ic`^{s}`), d:[]/(Oo`^{t}`|Ic`^{t}`)`

✓ G; `s<t` $\vdash$ `c?[].d![] | ...`

✗ G; `s<t` $\vdash$ `d?[].c![] | ...`

✓ G; `t<s` $\vdash$ `d?[].c![] | ...`

✗ G; `t<s` $\vdash$ `c?[].d![] | ...`

Preventing & Detecting Cycles in the Dependency

G = `c:[]/(Oo`^{s}`|Ic`^{s}`), d:[]/(Oo`^{t}`|Ic`^{t}`)`

✗ G; `s<t` $\vdash$ `c?[].d![] | d?[].c![]`

✗ G; `t<s` $\vdash$ `c?[].d![] | d?[].c![]`

G; `s<t,t<s` $\vdash$ `c?[].d![] | d?[].c![]`

Outline

Type Judgment

$G;\ \prec\ \vdash\ \mathbf{P}$

- G : type environment
  (mapping from variables to types)
- $\prec$: time tag ordering
  (binary relation on time tags)

P uses communication channels according to:

- the usage specified by Γ
- the order specified by $\prec$

Example of Typing Rules

T-Out (simplified):

$$\frac{\begin{array}{c} t \text{ includes obligations} \Rightarrow \mathbf{a} \text{ includes capability} \\ \mathbf{s} \;\; \mathbf{<} \;\; \text{time tags on obligations included in } t \\ G \text{ includes no obligation} \end{array}}{G \;\mathbf{+}\; \mathtt{x:[}t\mathtt{]/}\mathbf{O}_{\mathbf{a}}^{\mathbf{s}} \;\mathbf{+}\; \mathtt{y:}t\mathtt{;} \;\; \mathbf{<} \vdash \mathtt{x![y]}}$$

Example of Typing

```
ret:[int]/Oo^u; Æ ⊢
   def fib[i:int,r:[int]/Oo^s] =
      if i<2
         then r![1]
         else
            new c:[int]/(Oo^t|Oo^t|Ic^t.Ic^t)
               in (fib![i-1,c] | fib![i-2,c]
                         | c?[j].c?[k].r![j+k])
   in fib![10,ret]
```

Example of Typing

```
ret:[int]/Oo^u; Æ ⊢
   def fib[i:int,r:[int]/Oo^s] =
      if i<2
         then r![1]
         else
            new c:[int]/(Oo^t|Oo^t|Ic^t.Ic^t)
               in (fib![i-1,c] | fib![i-2,c]
                        | c?[j].c?[k].r![j+k])
   in fib![10,ret]
```

Example of Typing

```
ret:[int]/Oo^u; Æ ⊢
   def fib[i:int,r:[int]/Oo^s] =
      if i<2
         then r![1]
         else
            new c:[int]/(Oo^t|Oo^t|Ic^t.Ic^t)
               in (fib![i-1,c] | fib![i-2,c]
                        | c?[j].c?[k].r![j+k])
   in fib![10,ret]
```

Example of Typing

```
ret:[int]/Oo^u; Æ ⊢
  def fib[i:int,r:[int]/Oo^s] =
    if i<2
      then r![1]
      else
        new c:[int]/(Oo^t|Oo^t|Ic^t.Ic^t)
          in (fib![i-1,c] | fib![i-2,c]
                  | c?[j].c?[k].r![j+k])
  in fib![10,ret]
```

Outline

Correctness of the Type System

No immediate deadlock:

Well-typed processes are not in deadlock

Correctness of the Type System

No immediate deadlock:

Well-typed processes are not in deadlock

+

Subject reduction:

Well-typedness is preserved by reduction

Correctness of the Type System

No immediate deadlock:

Well-typed processes are not in deadlock

+

Subject reduction:

Well-typedness is preserved by reduction

$\Downarrow$

Deadlock-freedom:

Well-typed processes never fall into deadlock throughout reduction

Correctness of the Type System

Deadlock-freedom:

(the case of an output obligation)

- $G + \mathbf{x}:[t]/\mathbf{O}_{\mathbf{o}}^{\mathbf{t}}; \prec \vdash \mathbf{P}$
- Every usage in $G + \mathbf{x}:[t]/\mathbf{O}_{\mathbf{o}}^{\mathbf{t}}$ is reliable
- $\prec^{+}$ is a strict partial order
    - **P** will eventually perform output on **x**

(unless ‘infinite loop’)

Expressiveness of the Calculus

Expressive enough to encode:

- Parallel functions
- Typical concurrent objects
- Various semaphores

Expressiveness of the Calculus

Expressive enough to encode:

- Parallel functions
- Typical Concurrent Objects
- Various Semaphores

Too conservative to express:

- Case-by-case dependency

  c:[]/($I_{co}^{s}|O_{co}^{s}$), d:[]/($I_{co}^{t}|O_{co}^{t}$);

  $s<t, t<s \vdash$

  c![] | d![] |

  if … *then* c?[]…..d?[].… *else* d?[].….c?[].

Outline

- Introduction
- Basic Ideas
- The Type System
  - Type Judgment & Typing Rules
  - Correctness & Expressiveness
  - Type Checking
- Related Work
- Conclusion

Issues in Type Checking

- Usages of channels:
  must be explicitly specified by programmers
- Reliability of usages:
  can be automatically checked
  (by a co-inductive method)
- Time tag ordering:
  can be automatically inferred
  (by generation & satisfaction of constraints)

Outline

- Introduction
- Basic Ideas
- The Type System
- Related Work
- Conclusion

Related Work (1/4)

[Kobayashi 97]

Partially deadlock-free typed process calculus

- **In what way each channel may be used**
  - Linear Channels (used just once for communication)
  - Mutex Channels (used like binary semaphores)
  - Replicated Input Channels (used for process definitio
- **In what order those channels may be used**
  - Time tags and their ordering

Related Work (2/4)

[Pierce & Sangiorgi 93]

I/O Types:

In what direction a channel may be used

(for input, for output, or for both)

$$\texttt{c:-}[\mathit{int}] \Leftrightarrow \texttt{c:}[\mathit{int}]\texttt{/!O}$$

[Kobayashi & Pierce & Turner 96]

Linear Types:

How many times a channel may be used

(once or unlimitedly)

$$\texttt{c:}\uparrow^{1}[\mathit{int}] \Leftrightarrow \texttt{c:}[\mathit{int}]\texttt{/(O..|T..)}$$

Related Work (3/4)

[Yoshida 96]

Graph Types:

In what order processes perform input/output on channels

— Only ‘capability + obligation’;

cannot express ‘capability without obligation’ and ‘obligation without capability’

Related Work (4/4)

[Boudol 97]

Hennessy-Milner logic with recursion:

On what channels processes are ready to receive values

— Deadlock-freedom only for output;

cannot guarantee deadlock-freedom for input

Outline

- Introduction
- Basic Ideas
- The Type System
- Related Work
- Conclusion

Conclusion (1/2): Summary

Static type system that prevents deadlock:

- Usages & Usage Calculus

  “In what way each channel is used”

  +

- Time Tags & Time Tag Ordering

  “In what order those channels are used”

Conclusion (2/2): Future Work

- Develop a (partial) type inference algorithm
- Apply to practical concurrent languages
- Utilize for compile-time optimization

Prototype type checker available at:

`http://www.is.s.u-tokyo.ac.jp`
`/~sumii/pub/`