

Systemy typów dla języków programowania

(Część II: Programowanie współbieżne - DRAFT)

E-mail: Pawel.T.Wojciechowski [at] put.poznan.pl

10 stycznia 2008

Literatura

Prace:

Naoki Kobayashi

oraz

Robin Milner, Benjamin C. Pierce, ...

Dedukcja typu w ML

```
let f x = x + 1;;  
val f : int -> int = <fun>
```

lub

```
let f = fun x -> x + 1;;  
val f : int -> int = <fun>
```

Błąd typowania:

```
f "ala";;
```

*This expression has type string but is here used
with type int*

Typowanie w programowaniu współbieżnym?

Program w Concurrent ML:

```
let f (x:int) =  
  let c = channel () in  
    recv c + x;;  
val f : int -> int = <fun>
```

f 1

Funkcja f tworzy nowy kanał komunikacyjny c i stara się odebrać komunikat

Czy powyższy program jest prawidłowy (tj. powinien być typowalny)?

Typowanie w programowaniu współbieżnym?

Program w Concurrent ML:

```
let f (x:int) =  
  let c = channel () in  
    recv c + x;;  
val f : int -> int = <fun>
```

```
f 1  
??
```

Wywołanie `f` jest zablokowane na zawsze, gdyż brakuje procesu nadawcy

Rachunek π (*ang.* π -calculus) [Milner 1992]

Procesy

$P ::= 0$	pusty proces
$x![v_1 \dots v_n].P$	wyślij komunikat
$x?[y_1 : t_1 \dots y_n : t_n] = P$	odbierz komunikat
$(P \mid Q)$	procesy równoległe
$\text{new } x : t P$	nowy kanał komunikacyjny
$*P$	replikuj process
$\text{if } v \text{ then } P \text{ else } Q$	warunek (rozgałęzienie)

Wartości

$v ::= x, y, ..$
true
false

Program *ping* (1/2)

```
new ping : t
  ( *ping?r = r![]
  | new reply : t' (ping!reply
                    | reply?_ = P))
```

Program *ping* (2/2)

new ping : t

(**ping?r = r![]*
| **new reply : t'** (*ping!reply | reply?_ = P*)

→ (**ping?r = r![]*
| *r![] {reply/r} | reply?_ = P*)

→ (**ping?r = r![]*
| *reply![] | reply?_ = P*)

→ (**ping?r = r![]*
| *P*

Procesy replikowane

$$\text{new } p : t \\ (*p?[x_1 \dots x_n] = P \\ | Q)$$

Inna notacja:

$$\text{def } p [x_1:t_1 \dots x_n:t_n] = P \\ Q$$

Konkretny program, np:

$$\text{new } c : t$$
$$\text{def } p [x:t \ y:t'] = x!y \\ p![c \ 2]$$

Procesy replikowane

Co robi poniższy proces ?

```
def p [n:t r:t'] =  
  if n < 0 then  
    ()  
  else  
    (r![] | p![(n-1) r])
```

Obiekty

Obiekt udostępniający dwie metody *set* i *get*, odpowiednio aktualizujące i pobierające stan obiektu (typu *int*)

new s : t

(s!0

| * set?[v r] = s?_ = (s!v | r![])

| * get?r = s?x = (s!x | r!x)

Zamki (*ang.* locks)

Obiekt udostępniający dwie metody *set* i *get*, odpowiednio aktualizujące i pobierające stan obiektu (typu *integer*)

new s : t

(*s!0*

| * *set?[v r] = s?_ = (s!v | r![])*

| * *get?r = s?x = (s!x | r!x)*)

lock?_ =

new r : t

(*get!r | r?x =*

new r' : t' (set![(x + 1) r'] | r'?_ = lock![]))

Adnotacje typów

```
new p : [t1 ... tn]  
  ( *p?[x1:t1 ... xn:tn] = P  
  | Q)
```

Czy typy we wzorcu (po *p?) są potrzebne? A w notacji niżej?

```
def p [x1:t1 ... xn:tn] = P  
Q
```

Konkretny program, np:

```
new c:int chan
```

```
def p [x:int chan y:int] = x!y  
p![c 2]
```

Adnotacje typów

```
def p [n:int r:[] chan] =  
  if n < 0 then  
    ()  
  else  
    (r![] | p![(n-1) r])
```

Jaki jest typ kanału komunikacyjnego p ?

Prosty system typów

Chcemy wykryć proste błędy

Przykład 1

```
new ping : t  
  ( *ping?r = r![]  
    | (..) ping!true)
```

Przykład 2

```
new ping : t  
  ( *ping?r = r![]  
    | (..) ping![x y])
```

(Większość współbieżnych języków progr. wykryje te błędy.)

Typy w rachunku π

Sklasyfikujemy wartości zgodnie z ich typami oraz kanały zgodnie z typami komunikatów; dodajmy typ procesów:

Typy

$t ::= \mathit{bool}$	wartości Boolowskie
$ [t_1 \dots t_n] \mathit{chan}$	kanały komunikacyjne
$\sigma ::= t$	
$ \mathit{proc}$	procesy

Ponizsze programy się nie skompilują

Przykład 1

```
new ping : [[] chan] chan  
  ( *ping?r = r![]  
  | (..) ping!true)
```

Przykład 2

```
new ping : [[] chan] chan  
  ( *ping?r = r![]  
  | (..) ping![x y])
```

Stwierdzenie typowalności

Zachodzi

$$\Gamma \vdash A : \sigma$$

jeśli wyrażenie A (ewaluowane do wartości lub procesu) jest dobrze typowane oraz ma typ σ

Prosty system typów dla rachunku π (1/2)

$$\text{(bool)} \quad \frac{b \in \{\text{true}, \text{false}\}}{\emptyset \vdash b : \text{bool}}$$

$$\text{(par)} \quad \frac{\Gamma \vdash P : \text{proc} \quad \Gamma \vdash Q : \text{proc}}{\Gamma \vdash P \mid Q : \text{proc}}$$

$$\text{(var } \succ) \quad x : t \vdash x : t$$

$$\text{(weak)} \quad \frac{\Gamma' \vdash A : \sigma \quad \Gamma \leq \Gamma'}{\Gamma \vdash A : \sigma}$$

$$\text{(new)} \quad \frac{\Gamma, x : t \vdash P : \text{proc} \quad t \text{ jest typem kanałów}}{\Gamma \vdash (\text{new } x : t) P : \text{proc}}$$

$$\text{(zero)} \quad \emptyset \vdash \mathbf{0} : \text{proc}$$

$$\text{(rep)} \quad \frac{\Gamma \vdash P : \text{proc}}{\Gamma \vdash *P : \text{proc}}$$

$\Gamma \leq \Gamma'$ gdy $\text{dom}(\Gamma) \supseteq \text{dom}(\Gamma')$ i $\forall x \in \text{dom}(\Gamma') \quad \Gamma(x) = \Gamma'(x)$

Prosty system typów dla rachunku π (2/2)

$$\begin{array}{l} \text{(out)} \quad \frac{\Gamma \vdash x : [t_1 \dots t_n] \text{ chan} \quad \Gamma \vdash v_i : t_i \text{ for each } i = 1..n \quad \Gamma \vdash P : \text{proc}}{\Gamma \vdash x![v_1 \dots v_n].P : \text{proc}} \\ \\ \text{(in)} \quad \frac{\Gamma \vdash x : [t_1 \dots t_n] \text{ chan} \quad \Gamma, y_1 : t_1, \dots, y_n : t_n \vdash P : \text{proc}}{\Gamma \vdash x?[y_1 : t_1 \dots y_n : t_n] = P : \text{proc}} \\ \\ \text{(if)} \quad \frac{\Gamma \vdash v : \text{bool} \quad \Gamma \vdash P : \text{proc} \quad \Gamma \vdash Q : \text{proc}}{\Gamma \vdash \text{if } v \text{ then } P \text{ else } Q : \text{proc}} \end{array}$$

Co jeszcze moglibyśmy zweryfikować?

Czy poniższy program jest poprawny?

```
new ping : [[] chan] chan  
( *ping?r = r?_ = 0  
| (..) ping!reply)
```

Czy poniższy program jest poprawny?

```
new ping : [[] chan] chan  
( *ping?r = r?[] = 0  
| (..) ping!reply)
```

prawidłowy program *ping* wygląda inaczej

```
new ping : [[] chan] chan  
( *ping?r = r![]  
| (..) ping!reply)
```

System typów wejścia/wyjścia

Typy

$t ::= \text{bool}$ wartości Boolowskie
| $[t_1 \dots t_n] \text{chan}_M$ kanały komunikacyjne

$\sigma ::= t$
| proc procesy

$M ::= ! | ? | !?$ tryb

Tryb M wskazuje które akcje mają zastosowanie dla danego kanału

Poniższy program się nie skompiluje

```
new ping : [[] chan!] chan?  
  ( *ping?r = r?[] = 0  
  | (..) ping!reply)
```

ale następujący program tak

```
new ping : [[] chan!] chan?  
  ( *ping?r = r![]  
  | (..) ping!reply)
```

Podtypowanie

Piszemy $t_1 \prec t_2$ kiedy wartość typu t_1 może być użyta jako wartość typu t_2 , np.

$$[t_1 \dots t_n] \text{ chan}! ? \prec [t_1 \dots t_n] \text{ chan} ?$$

oraz

$$[t_1 \dots t_n] \text{ chan}! ? \prec [t_1 \dots t_n] \text{ chan} !$$

$\Gamma \leq \Gamma'$ gdy

$$\text{dom}(\Gamma) \supseteq \text{dom}(\Gamma') \text{ oraz } \forall x \in \text{dom}(\Gamma') \Gamma(x) \prec \Gamma'(x)$$

System typów wejścia/wyjścia

$$\text{(m-out)} \quad \frac{\Gamma \vdash x : [t_1 \dots t_n] \text{ chan}_! \quad \Gamma \vdash v_i : t_i \text{ for each } i = 1..n \quad \Gamma \vdash P : \text{proc}}{\Gamma \vdash x![v_1 \dots v_n].P : \text{proc}}$$

$$\text{(m-in)} \quad \frac{\Gamma \vdash x : [t_1 \dots t_n] \text{ chan}_? \quad \Gamma, y_1 : t_1, \dots, y_n : t_n \vdash P : \text{proc}}{\Gamma \vdash x?[y_1 : t_1 \dots y_n : t_n] = P : \text{proc}}$$

Pozostałe reguły wnioskowania jak na slajdach 20 i 21

Nieco bardziej złożony system typów

Poniższe programy są problematyczne

Przykład 1

```
new ping : [[] chan!] chan?  
  ( *ping?r = if b then r![] else 0  
    | (..) ping!reply)
```

Przykład 2

```
new ping : [[] chan!] chan?  
  ( *ping?r = (r![] | r![])  
    | (..) ping!reply)
```

Liniovyy system typów (1/3)

Typy

$t ::= \text{bool}$ wartości Boolowskie
| $[t_1 \dots t_n] \text{chan}_{(?m_1, !m_2)}$ kanały komunikacyjne

$\sigma ::= t$
| proc procesy

$m ::= 0 \mid 1 \mid \omega$ krotność

Krotność M wskazuje ile razy kanał może być zastosowany:

0 kanał nie może być użyty

1 kanał powinien być użyty raz

ω kanał może być użyty dowolną liczbę razy

Ponizsze programy się nie skompilują

Przykład 1

```
new ping : [[] chan(?0,!1)] chan(?ω,!0)  
  ( *ping?r = if b then r![] else 0  
  | (..) ping!reply)
```

Przykład 2

```
new ping : [[] chan(?0,!1)] chan(?ω,!0)  
  ( *ping?r = (r![] | r![])  
  | (..) ping!reply)
```

Liniowy system typów (2/3)

$$(l\text{-par}) \quad \frac{\Gamma \vdash P : \text{proc} \quad \Delta \vdash Q : \text{proc}}{\Gamma \mid \Delta \vdash P \mid Q : \text{proc}}$$

$$(l\text{-rep}) \quad \frac{\Gamma \vdash P : \text{proc}}{\omega\Gamma \vdash *P : \text{proc}}$$

Liniowy system typów (3/3)

$$\text{(l-out)} \quad \frac{\Gamma_i \vdash v_i : t_i \text{ for each } i = 1..n \quad \Gamma \vdash P : \text{proc}}{(x : [t_1 \dots t_n] \text{ chan}_{(?^0, !^1)}) \mid \Gamma_1 \mid \dots \mid \Gamma_n \mid \Gamma \vdash x![v_1 \dots v_n].P : \text{proc}}$$

$$\text{(l-in)} \quad \frac{\Gamma, y_1 : t_1, \dots, y_n : t_n \vdash P : \text{proc}}{(x : [t_1 \dots t_n] \text{ chan}_{(?^1, !^0)}) \mid \Gamma \vdash x?[y_1 : t_1 \dots y_n : t_n] = P : \text{proc}}$$

$$\text{(l-if)} \quad \frac{\Gamma \vdash v : \text{bool} \quad \Delta \vdash P : \text{proc} \quad \Delta \vdash Q : \text{proc}}{\Gamma \mid \Delta \vdash \text{if } v \text{ then } P \text{ else } Q : \text{proc}}$$

Relacja uporządkowania $m_1 \leq m_2$

Zachodzi częściowe uporządkowanie:

$$\omega \leq 0 \quad \wedge \quad \omega \leq 1$$

Relacja podtypowania \prec

$$(I\text{-if}) \quad \frac{m_1 \leq m'_1 \quad m_2 \leq m'_2}{[t_1 \dots t_n] \text{chan}_{(?m_1, !m_2)} \prec [t_1 \dots t_n] \text{chan}_{(?m'_1, !m'_2)}}$$

Przykład:

Kanał typu $[] \text{chan}_{(?^\omega, !^\omega)}$

może być używany jako kanał typu $[] \text{chan}_{(?^1, !^0)}$

ale

kanał typu $[] \text{chan}_{(?^0, !^1)}$

(którym *musi* być wysłany komunikat)

nie może być używany zamiast $[] \text{chan}_{(?^0, !^0)}$

(którym *nie można* wysyłać komunikatów)

Definicja $\Gamma \leq \Gamma'$

$\Gamma \leq \Gamma'$ iff

(1) $dom(\Gamma) \supseteq dom(\Gamma')$

(2) $\forall x \in dom(\Gamma') \quad \Gamma(x) \prec \Gamma'(x)$

(3) $\forall x \in dom(\Gamma) \setminus dom(\Gamma')$

$$\Gamma(x) = \mathit{bool} \vee \Gamma(x) = [t_1 \dots t_n] \mathit{chan}_{(?m_1, !m_2)}$$

gdzie $m_1 \leq 0 \wedge m_2 \leq 0$

Przykład: czy zachodzi

$$x : t, y : [] \mathit{chan}_{(?^0, !^1)} \prec x : t \quad ?$$

Definicja $\Gamma \leq \Gamma'$

$\Gamma \leq \Gamma'$ iff

(1) $dom(\Gamma) \supseteq dom(\Gamma')$

(2) $\forall x \in dom(\Gamma') \quad \Gamma(x) \prec \Gamma'(x)$

(3) $\forall x \in dom(\Gamma) \setminus dom(\Gamma')$

$$\Gamma(x) = \mathit{bool} \vee \Gamma(x) = [t_1 \dots t_n] \mathit{chan}_{(?m_1, !m_2)}$$

gdzie $m_1 \leq 0 \wedge m_2 \leq 0$

Przykład: Czy zachodzi

$$x : t, y : [] \mathit{chan}_{(?0, !1)} \prec x : t \quad ?$$

nie!

Definicja $\Gamma \mid \Delta$

$$(\Gamma \mid \Delta)(x) = \begin{cases} \Gamma(x) \mid \Delta(x) & \text{if } x \in \text{dom}(\Gamma) \cup \text{dom}(\Delta), \\ \Gamma(x) & \text{if } x \in \text{dom}(\Gamma) \setminus \text{dom}(\Delta), \\ \Delta(x) & \text{if } x \in \text{dom}(\Delta) \setminus \text{dom}(\Gamma), \end{cases} \quad (1)$$

$$\text{bool} \mid \text{bool} = \text{bool}$$

$$\begin{aligned} ([t_1 \dots t_n] \text{chan}_{(?m_1, !m_2)}) \mid ([t_1 \dots t_n] \text{chan}_{(?m'_1, !m'_2)}) \\ = [t_1 \dots t_n] \text{chan}_{(?m_1+m'_1, !m_2+m'_2)} \end{aligned}$$

$$m_1 + m_2 = \begin{cases} m_2 & \text{if } m_1 = 0, \\ m_1 & \text{if } m_2 = 0, \\ \omega & \text{w przeciwnym razie} \end{cases} \quad (2)$$

Definicja operacji $\omega\Gamma$

$$(\omega\Gamma)(x) = \omega(\Gamma(x))$$

$$\omega bool = bool$$

$$\omega([t_1 \dots t_n] \text{chan}_{(?m_1, !m_2)}) = [t_1 \dots t_n] \text{chan}_{(? \omega m_1, ! \omega m_2)}$$

$$\omega m = \begin{cases} 0 & \text{if } m = 0, \\ \omega & \text{w przeciwnym razie} \end{cases} \quad (3)$$

Ale czy wszystko OK?

Niestety nie...

Liniowy system typów wcale nie gwarantuje, że na przykład

x : chan_(?0,!1)

będzie użyty dokładnie raz do wysłania komunikatu!

Problem

Poniższy program jest dobrze typowany w środowisku typowania $x : [] \text{chan}_{(?^0,!^1)}$, a jednak trudno uznać go za poprawny!

```
(new y)(new z)(y?_ = z![] | z?_ = (y![] | x![]))
```

Dlaczego?

Problem z wielokrotnym użyciem kanałów

Poniższe programy są błędne:

```
lock?_ =  
  <sekcja krytyczna>  
  (lock![] | lock![])
```

```
lock?_ = <sekcja krytyczna> if b then lock![] else 0
```

Liniowy system typów nie gwarantuje poprawnego użycia zamków

Zamek *lock* może być używany wielokrotnie, a więc ma typ [] *chan*(? ω ,! ω), tj. może być używany w arbitralny sposób

Typy opisujące sposób użycia kanałów

Typy

$t ::= \mathit{bool}$	wartości Boolowskie
$\quad \quad [t_1 \dots t_n] \mathit{chan}_U$	kanały komunikacyjne
$\sigma ::= t$	
$\quad \quad \mathit{proc}$	procesy

Sposób używania kanału

$$U ::= 0 \mid \rho \mid ?.U \mid !.U \mid (U_1 \mid U_2) \mid U_1 \& U_2 \mid \mu \rho. U$$

Sposób użycia (zastosowanie) kanałów

$U ::= 0$	kanał nie może być użyty
ρ	
$?U$	najpierw odbierz komunikat
$!U$	najpierw wyślij komunikat
$(U_1 \mid U_2)$	zgodnie z U_1 i U_2 (także równolegle)
$U_1 \& U_2$	zgodnie z albo U_1 lub U_2
$\mu\rho.U$	rekurencyjnie jako $[\mu\rho.U/\rho]U$

np. $\mu\rho.(0 \& (!\rho))$ opisuje kanał używany sekwencyjnie do wysłania dowolnej liczby komunikatów

$\mu\rho.(?!\rho)$ to ?

Skrótowe oznaczenia $*U$ i ωU

Kanał który może być używany zgodnie z U dowolną liczbę razy ma typ:

$$*U \equiv \mu\rho.(0 \ \& \ (U \mid \rho))$$

Kanał który powinien być używany zgodnie z U nieskończenie często ma typ:

$$\omega U \equiv \mu\rho.(U \mid \rho)$$

Inne skróty: $?$ zamiast $? \cdot 0$, oraz $!$ zamiast $! \cdot 0$

Zakodowanie typów liniowych

Możemy zakodować typy liniowe za pomocą typów sposobu użycia kanałów:

$$(?^{m_1}, !^{m_2}) \equiv m_1? \mid m_2!$$

(krotność: $m ::= 0 \mid 1 \mid \omega$)

gdzie

$$1U \equiv U$$

$$0U \equiv 0$$

$$\omega U \equiv \mu\rho.(U \mid \rho)$$

Wymuszenie prawidłowego zastosowania zamków

Zamek *lock* powinien mieć typ $! \mid *?.!$, wtedy poniższe (błędne) programy nie są typowalne:

$$\begin{aligned} lock?_ = \\ & \langle \text{sekcja krytyczna} \rangle \\ & (lock![] \mid lock![]) \end{aligned}$$
$$lock?_ = \langle \text{sekcja krytyczna} \rangle \text{ if } b \text{ then } lock![] \text{ else } 0$$

Przypomnijmy, że:

$$*U \equiv \mu\rho.(0 \ \& \ (U \mid \rho))$$

Typy dla wolności od zakleszczenia

Problem zakleszczenia

Systemy typów zaprezentowane dotychczas nie gwarantują, że serwer “ping” da w końcu odpowiedź (w przypadku braku awarii), lub że zamek zostanie w końcu zwrócony, etc.

np. system typów ze sposobem użycia kanałów akceptuje proces

$$lock?_ = (new\ x)(new\ y)(x?_ = y![] \mid y?_ = (lock![] \mid x![]))$$

który nie zwalnia zamka z powodu wystąpienia *zakleszczenia* (*ang.* deadlock) w kanałach x i y

Tak się dzieje gdyż system typów uwzględnia komunikację w kanałach ale nie zależności między *różnymi* kanałami (x i y)

Procesy wolne od zakleszczenia

Proces $x!v.P$ wysyła v do x i po tym jak v zostało odebrane przez jakiś proces, wykonuje P (jeśli $P = 0$ to piszemy $x!v$)

Wolność od zakleszczenia: jeśli akcja wysłania komunikatu została kiedykolwiek wykonana, v będzie w końcu odebrane przez jakiś proces lub cały program “pójdzie w innym kierunku” (*ang.* diverge)

Proces $x?y = P$ czeka na otrzymanie v z x a po otrzymaniu v wykonuje $P[v/y]$

Wolność od zakleszczenia: jeśli akcja odebrania komunikatu została kiedykolwiek wykonana, proces będzie w końcu w stanie otrzymać komunikat z x lub cały program “pójdzie w innym kierunku” (*ang.* diverge)

Systemy typów dla wolności od zakleszczenia

Pomysł polega na rozszerzeniu typów opisujących kanały komunikacyjne o:

- **sposób użycia kanału** (*ang.* channel usage), który opisuje jak często i w jakim porządku kanał ma być stosowany do wysyłania / odbierania komunikatów (jak w poprzednim systemie typów)
- **zdolność i zobowiązanie** (*ang.* capability and obligation) do wykonania każdej akcji wejścia/wyjścia; informacje te opisują zależności *między* kanałami

Przykłady typów zastosowań kanałów

Zastosowanie x w procesie $x?y \mid x!1 \mid x!2$ można wyrazić jako $? \mid ! \mid !$, czyli x jest używane jeden raz do odbioru i dwa razy do wysłania, szeregowo lub równoległe

Zastosowanie x w procesie $x?y = x!y$ można wyrazić jako $?.!$, czyli x jest najpierw użyte do odbioru, potem do wysłania komunikatu

Informacja o sukcesie lub niepowodzeniu

Zastosowanie zawiera pewną informację o tym czy każda akcja kończy się sukcesem czy nie, np.

x mający zastosowanie ? | ! | ! wskazuje że przynajmniej jedno z tych dwóch wysłań nie powiedzie się!

x mający zastosowanie ?.! (w całym procesie/programie) wskazuje, że ani akcja wejścia ani akcja wyjścia nie zakończy się sukcesem, gdyż obie akcje nie zachodzą równolegle

Zobowiązanie do i zdolność wykonania akcji

Sama informacja o sposobie użycia kanałów nie wystarcza do analizy zakleszczenia, np. nie można wykryć różnicy między zakleszczonym procesem

$$x?z = y!z \mid y?z = x!1$$

and nie-zakleszczonym procesem

$$x?z = y!z \mid x!1.y?z = 0$$

Potrzebna jest też informacja opisująca:

zdolność (*ang.* capability) do pomyślnego odbioru lub wysłania komunikatu

zobowiązanie (*ang.* obligation) do czekania na odbiór lub do wysłania komunikatu

Przykład zakleszczonego procesu

$$x?z = y!z \mid y?z = x!1$$

Aby lewy proces $x?z = y!z$ z sukcesem otrzymał komunikat na x , jakiś proces musi wypełnić *zobowiązanie* wysłania do x .

Jednakże prawy proces $y?z = x!1$ ma *zdolność* otrzymania komunikatu z y zanim wypełni to zobowiązanie.

Aby prawy proces był w stanie kontestować tę zdolność, lewy proces musi wypełnić swoje zobowiązanie wysłania komunikatu w y , niestety zanim będzie w stanie wypełnić to zobowiązanie, proces ten próbuje swojej zdolności do otrzymania komunikatu z x

Tak więc zależność między zdolnościami a zobowiązaniami tworzy cykl, więc komunikacja nie następuje

Poziom zdolności i zobowiązania

Aby uniknąć formowania się cykli, z każdą z akcji (? lub !) występujących w typie zastosowań kanałów skojarzone będą *poziomy* zobowiązania i zdolności z zakresu $\{0, 1, 2, \dots\} \cup \{\infty\}$

∞ oznacza, że akcja nie musi być wykonana

np. $?_{c=1}^{o=1}$ oznacza odebranie komunikatu na poziomach zobowiązania (obligation) i zdolności (capability) równych **1**; w skrócie piszemy $?_1^1$

Uwaga: poziomy zobowiązania i zdolności są generowane przez odpowiedni algorytm dedukcji typów (dla wolności od zakleszczenia), więc nie obarcza to programisty

Reguły weryfikacji wolności od zakleszczenia

Poziomy zdolności i zobowiązań nakładają następujące reguły na zachowanie procesu i jego środowiska:

A **Zobowiązanie (obligation)** na poziomie n ($n \neq \infty$) musi być wypełnione używając wyłącznie **zdolności (capability)** na poziomie mniejszym niż n

np. założmy, że x ma sposób użycia $?_0^0$ a y ma $?_1^1$;
wtedy procesy $x?z = y!z$ oraz $x?z = 0 \mid y!1$ są do przyjęcia, ale $y!1.x?z$ już nie;

ostatni proces charakteryzuje się zdolnością na poziomie 1 przed wypełnieniem zobowiązania na poziomie niższym

B Dla akcji o poziomie **zdolności** n ($n \neq \infty$), musi istnieć ko-akcja o poziomie **zobowiązań** mniejszym lub równym n (tak aby gwarantować, że zdolność może być ostatecznie kontestowana)

Wolność od zakleszczenia

Reguły A i B gwarantują, że nie ma cyklicznych zależności między zdolnościami i zobowiązaniami o skończonych poziomach, a więc gwarantują wolność od zakleszczenia dla każdej akcji o skończonym poziomie zdolności

Przykład weryfikacji wolności od zakleszczenia

$$x?z = y!z \mid y?z = x!z$$

Przyjmijmy, że zastosowania x i y są odpowiednio $?_{c_{x1}}^{o_{x1}} \mid !_{c_{x2}}^{o_{x2}}$ oraz $?_{c_{y1}}^{o_{y1}} \mid !_{c_{y2}}^{o_{y2}}$, gdzie zdolności c_{x1} i c_{y1} są skończone

Reguła A implikuje: $c_{x1} < o_{y2}$ oraz $c_{y1} < o_{x2}$

Reguła B implikuje:

$$o_{x2} \leq c_{x1},$$

$$o_{x1} \leq c_{x2},$$

$$o_{y2} \leq c_{y1} \text{ oraz } o_{y1} \leq c_{y2}$$

Stąd otrzymujemy: $c_{x1} < o_{y2} \leq c_{y1} < o_{x2} \leq c_{x1}$, co jest sprzecznością

Wada powyższego systemu typów

Opisany system typów nie radzi sobie z procesami rekurencyjnymi!

Dla przykładu weźmy program obliczający silnię:

```
*silnia?[n r] =  
  if n == 0  
    then r!1  
    else (new r') (silnia![n - 1 r']  
                  | r'?m = r!(m * n))
```

Weryfikacja rekurencji zawodzi

```
*silnia?[n r] = if n == 0 then r!1  
               else (new r') (silnia![n - 1 r'] | r'?m = r!(m * n))
```

Argument r dla $silnia$ ma typ $int\ chan_{t_o}^{t_c}$, co oznacza, że kanał jest używany do wysyłania wartości typu int oraz poziomy zobowiązania i zdolności do wykonania tej akcji są odpowiednio t_o i t_c . Ponieważ r' jest też wysłany do $silnia$, więc ma ten sam typ $int\ chan_{t_o}^{t_c}$.

Ale wtedy zgodnie z regułą B poziom zdolności do wykonania akcji wejścia na r' w $r'?m = \dots$ musi być większy lub równy t_o . A więc podproces $r'?m = r!(m * n)$ nie spełnia reguły A (jeśli t_o nie jest ∞).

Rozwiązanie problemu rekurencji

Aby usunąć tę wadę, możemy osłabić regułę A jak niżej:

A' **Zobowiązanie** na poziomie n dla kanału x musi być wypełnione używając wyłącznie **zdolności** na poziomie mniejszym *lub równym* n , oraz jeśli poziom zdolności jest równy n , to wtedy ta zdolność musi dotyczyć kanału, który został utworzony później niż x

np. w programie 'silnia' poziom zobowiązania zwrócenia wartości używając r oraz poziom odbioru wartości z r' są takie same, ale ponieważ r' został utworzony bardziej niedawno, więc $r' \leq m = r!(m * n)$ zgadza się z regułą A'

Reguła A' jest dostateczna aby zapobiec zakleszczeniu przez uniknięcie cyklicznych zależności między różnymi kanałami

Analiza statyczna w regule A'

Potrzebna jest analiza statyczna aby estymować informację o tym które kanały zostały utworzone bardziej ostatnio

Rozwiązanie: prosta analiza syntaktyczna, która konkluduje, że w procesie $(\text{new } x) P$ kanał x został utworzony później (tj. bardziej ostatnio) niż jakikolwiek inny *wolny* kanał w P

Powyższa procedura jest dostateczna w przypadku typowania procesów rekurencyjnych takich jak w programie 'silnia'

Sposoby użycia kanałów do analizy zakleszczenia

$U ::= 0$	kanał nie może być użyty
$\alpha_{t_c}^{t_o}.U$	odbierz/wyślij komunikat, potem jak U
$(U_1 \mid U_2)$	zgodnie z U_1 i U_2 (także równoległe)
$*U$	zgodnie z U , nieskończenie wiele procesów
$\uparrow^t U$	jak U ale poziom zobowiązań poniesiony do t
$U_1 \& U_2$	zgodnie z albo U_1 lub U_2
ρ	zmienna sposobu użycia (zob. niżej)
$\mu\rho.U$	rekurencyjnie jako $[\mu\rho.U/\rho]U$
$\alpha ::= ? \mid !$	

gdzie t należy do przedziału $\mathbf{Nat} \cup \{\infty\}$

$\forall t \in \mathbf{Nat} \cup \{\infty\} \ t \leq \infty$ oraz $\infty + t == t + \infty == \infty$

Znaczenie poziomu zdolności/zobowiązań

Jeśli poziom t_o jest skończony, to kanał o zastosowaniu $\alpha_{t_c}^{t_o}$ musi być użyty do akcji α , podczas gdy jeśli t_o jest ∞ , to akcja nie musi być wykonana

Jeśli t_c jest skończony, to akcja w końcu się powiedzie jeżeli jest kiedykolwiek wykonana i cały program (proces) nie “pójdzie w innym kierunku” (*ang.* diverge); jeśli t_c jest ∞ , to nie ma takiej gwarancji

np. kanał o zastosowaniu $?_0^\infty .!_0^\infty$ może (nie musi) być użyty do odbioru, oraz jeśli został użyty do odbioru i odbiór zakończył się sukcesem, to *musi* być użyty do wysłania komunikatu

Podnoszenie zobowiązań

$\uparrow^t U$ podnosi poziomy zobowiązań w U (oprócz tych chronionych przez $?$ i $!$) w taki sposób, że zobowiązania wejścia i wyjścia stają się większe lub równe t

np. $\uparrow^1 (?_0^0.!_0^0)$ jest równoważne $?_0^1.!_0^0$

Przykłady typowania wolnego od zakleszczenia

Kanały liniowe, które są użyte *dokładnie raz* mają typ $?_{n_2}^{n_1} \mid !_{n_4}^{n_3}$

Kanały affine, które mogą być użyte *najwyżej jeden raz* mają typ $?_{\infty}^{\infty} \mid !_{\infty}^{\infty}$

Referencje – kanał x zawierający bieżącą wartość referencji jako komunikat, z operacją odczytu $x?y = (x!y \mid \dots)$ i zapisu $x?y = (x!v \mid \dots)$ ma typ $!_{\infty}^0 \mid *?_0^{\infty} .!_{\infty}^0$

Semafory binarne – kanał przechowujący co najwyżej jeden komunikat, z operacjami podniesienia semafora jako akcji pobrania komunikatu i zwolnienia semafora jako akcji wysłania z powrotem komunikatu do kanału, ma typ $!_{\infty}^0 \mid *?_n^{\infty} .!_{\infty}^n$

(n kontroluje które zamki powinny być pobrane wpierw)

Zamki wolne od zakleszczenia (1/2)

$lock?_ = (new\ x)(new\ y)(x?_ = y![] \mid y?_ = (lock![] \mid x![]))$

Założmy zastosowania x i y odpowiednio $?_{t_1}^{t_0} \mid !_{t_0}^{t_1}$ oraz $?_{t_3}^{t_2} \mid !_{t_2}^{t_3}$

Z procesu $x?_ = y![]$ mamy $t_1 < t_3$

Z procesu $y?_ = (lock![] \mid x![])$ mamy $t_3 < t_1$

Czyli musi zachodzić przypadek $t_1 = t_3 = \infty$

(bo z definicji zachodzi $t < t$ wyłącznie jeśli $t = \infty$)

Ponieważ zwolnienie zamka $lock$ jest strzeżone przez wejście na y , to poziom zobowiązań na zwolnieniu zamka musi być także ∞ (wg. reguł systemu typów), co oznacza, że zamek może nie być zwolniony

Zamki wolne od zakleszczenia (2/2)

Zastosowanie (typ) zamka to $!_{\infty}^0 \mid *?_t^{\infty} .!_{\infty}^t$

Część $!_{\infty}^0$ oznacza, że wartość musi być umieszczona do kanału natychmiast (tak aby symulować otwarty zamek)

Część $*?_t^{\infty}$ oznacza, że dowolne akcje mogą być wykonane zanim nastąpi pobranie zamka i że po tym jak proces spróbuje pobrać zamek, proces ten może w końcu zamek pobrać

Część $!_{\infty}^t$ oznacza, że po tym jak proces pobrał zamek, to ma on zobowiązanie aby na poziomie t aby zamek zwolnić

Przykłady zamków wolnych od zakleszczenia

Przypuśćmy, że zamki l_1 i l_2 mają zastosowania, odpowiednio $*?_1^\infty .!_\infty^1$ oraz $*?_2^\infty .!_\infty^2$

Wtedy możliwe jest pobranie najpierw zamka l_2 a potem l_1 przed zwolnieniem l_2 :

$$l_2?_{} = l_1?_{} = (l_1![] \mid l_2![])$$

ale nie jest możliwe pobranie zamków l_1 i l_2 w odwrotnym porządku:

$$l_1?_{} = l_2?_{} = (l_1![] \mid l_2![])$$