チャネル使用法計算に基づく部分的デッドロック・フリー型付きプロセス計算の拡張

東京大学 理学部 情報科学科
小林研究室　住井 英二郎

並行プログラミングの重要性

- 並列・分散システム
- 本質的に並行性のあるアプリケーション

⇓

並行プログラミング：
複数のプロセスが通信しながら計算する

- 並行プログラミング言語
  CML [Reppy 91], Pict [Pierce & Turner 97]
- 並行プロセス計算
  π-calculus [Milner 93], HACL [Kobayashi 96]

並行言語を構成する要素

- プロセスの実行
  - `a . P`　a を行ってから P をする（逐次実行）
  - `P | Q`　P をしながら Q をする（並行実行）
- 通信チャネルを通じたデータの受け渡し
  - `new c`　新たなチャネル c を作る
  - `c![v]`　チャネル c に値 v を送る
  - `c?[v]`　チャネル c から値 v を受け取る

```
new c.(c![123] | c?[v].print![v])
                          → print![123]
```

並行プログラミングの問題点

- 非決定性：実行によって動作が相違
- デッドロック：期待した動作をせずに停止

プログラムについての推論を困難にする
　— バグの原因、最適化の障害などになる

並行プログラムに特有の問題
　⇒ 逐次プログラムを対象とした
　　型システムや解析では解決できない

既存の研究

- 以前の研究
  - 通信のトポロジを静的に固定

  ⇒ 一般の並行プログラムには適用できない
- 部分的デッドロック・フリー型付きプロセス計算 [Kobayashi 97]
  - 型システムを利用してデッドロックを防止

デッドロック・フリー計算 (1/3)

デッドロックの原因 (1) :
　複数のチャネルの使用順

```
x?[v].y![3]  |  y?[w].x![7]
```

⇓

チャネルの使用順を型システムで管理

$x{:}[int]^{s}, y{:}[int]^{t}\,;\, s < t \vdash x?[v] \,.\, y![3]$

$x{:}[int]^{s}, y{:}[int]^{t}\,;\, t < s \vdash y?[w] \,.\, x![7]$

$x{:}[int]^{s}, y{:}[int]^{t}\,;\, s < t, t < s \vdash x?[v] \,.\, y![3] \,|\, y?[w] \,.\, x![7]$

⇒ 型エラー

デッドロック・フリー計算 (2/3)

デッドロックの原因 (2)：
単独のチャネルの使用法

```
new c . c?[v] . …
```

⇓

チャネルの使用法を型システムで限定

- linear channel：一回の送受信に使用
- replicated input channel：プロセスの定義に使用
- mutex channel：二値セマフォとして使用

上の `c` はどれにも当てはまらない ⇒ 型エラー

デッドロック・フリー計算 (3/3)

デッドロックしない保証

複数のチャネルの使用順の管理

＋

単独のチャネルの使用法の管理

… ad hoc な限定

⇒ プログラマの自由を制限

今回の研究：目的とアイディア

型システムを拡張、
　デッドロックしない使用法を一般化

⇓

チャネル使用法計算を導入

- その項をチャネルに型として付加
- その簡約により使用法の正当性を判定

使用法計算：骨格

- 送信 `o`　受信 `I`　逐次実行 `.`　並行実行 `|`
- `o` と `I` が対応して簡約する（ → ）

- 送信一回と受信一回 (linear channel)

  `o | I` → ϕ

- 送信二回（並行）と受信二回（逐次）

  `o | o | I.I` → `o | I` → ϕ

使用法計算：能力と義務

capability と obligation の概念を導入

- capability（能力）
  I/O を実行すれば必ず成功する
  （実行しなくてもよい）
- obligation（義務）
  I/O を実行しなければならない
  （必ずしも成功しない）

使用法計算：正当性

使用法の正当性：

- capability には対応する obligation が存在する
- 任意に簡約しても同様の条件が成立する

- 正当な例

  mutex channel（二値セマフォ）

  $o_o \mid i_c.o_o \mid i_c.o_o \mid i_c.o_o \mid \ldots$

  $\rightarrow o_o \mid i_c.o_o \mid i_c.o_o \mid \ldots$

- 正当でない例

  $o_o \mid i_c \mid i_c \rightarrow i_c$

使用法計算：従来の使用法の表現

- linear channel

$$\mathtt{O}_{\mathtt{co}} \mid \mathtt{I}_{\mathtt{co}} \rightarrow \phi$$

- replicated input channel

$$!\mathtt{I}_{\mathtt{o}} \mid !\mathtt{O}_{\mathtt{c}} \rightarrow !\mathtt{I}_{\mathtt{o}} \mid !\mathtt{O}_{\mathtt{c}}$$

- mutex channel

$$\mathtt{O}_{\mathtt{o}} \mid !(\mathtt{I}_{\mathtt{c}}.\mathtt{O}_{\mathtt{o}}) \rightarrow \mathtt{O}_{\mathtt{o}} \mid !(\mathtt{I}_{\mathtt{c}}.\mathtt{O}_{\mathtt{o}})$$

（ここでの !u は u | u | u | … をあらわす）

型システムの拡張：型付け規則

型付け規則の例（骨格）

- input guard

$$\frac{\boldsymbol{ob}(\Gamma) \Rightarrow \boldsymbol{cap}(a) \quad \Gamma + x{:}\mathrm{U},\, y{:}\tau \vdash P}{\Gamma + x{:}[\tau]/(\mathrm{I}_a \,.\, \mathrm{U}) \vdash x?[y] \,.\, P}$$

- asynchronous output

$$\frac{\boldsymbol{ob}(\tau) \Rightarrow \boldsymbol{cap}(a) \quad \neg\, \boldsymbol{ob}(\Gamma)}{\Gamma + x{:}[\tau]/\mathrm{O}_a + y{:}\tau \vdash x![y]}$$

型システムの拡張：型判定

型判定の例

- $x:[int]/O_o,\ y:[int]/O_\phi \vdash x![5]$
  - ⇒ OK：x へ送信する義務は遂行されており、
    y へ送信する義務はない
- $x:[]/I_\phi,\ y:[]/O_o \vdash x?[].y![]$
  - ⇒ Error：x から受信する能力がないので、
    y へ送信する義務が必ずしも遂行されない
- $x:[]/(O_o \mid I_c),\ y:[]/O_o \vdash x![] \mid x?[].y![]$
  - ⇒ OK：x から受信する能力があるので、
    y へ送信する義務は必ず遂行される

型システムの正しさ

- subject reduction

  プロセスを簡約しても、同様の型付けを維持できる

- no immediate deadlock（output の義務のケース）

  $\Gamma + x{:}[\tau]/\mathrm{O}_{\mathrm{o}} \vdash P$ のときに、P が簡約できなければ、
  P は実際に x への output を実行している

  $\Downarrow$

- deadlock-freedom（output の義務のケース）

  $\Gamma + x{:}[\tau]/\mathrm{O}_{\mathrm{o}} \vdash P$ のときに、P を簡約していけば、
  P は実際に x への output を実行する

型検査規則と型検査器

- 型検査のアルゴリズムを
  ルールの集合として記述
- 型検査器を SML で実装（1510 行）

結論と今後の課題

- 結論

  チャネル使用法計算の導入によって、
  デッドロックしないことを保証する
  型システムを一般化

- 今後の課題
  - 型システムの正当さの厳密な証明
  - 実際の並行プログラミング言語への適用
  - 最適化などへの応用

比較に利用するプログラム

```
def fib[n,r] =
  if (n<2) then
    r![1]
  else
    new c .
      (fib![n-1,c] |
       fib![n-2,c] |
       c?[x].c?[y].r![x+y])
```

今までの型付け

```
def fib[n:int,r:↑¹[int]] =
  if (n<2) then r![1]
  else new c:↕ω[int] .
         (fib![n-1,c] | fib![n-2,c] |
          c?[x].c?[y].r![x+y])
```

- c は二回の送受信に使用されているので
  「それ以外」のチャネルに分類せざるをえない

⇓

- deadlock-freedom は保証されない

新しい型付け

```
def fib[n:int,r:[int]/Oo] =
  if (n<2) then r![1]
  else new c:[int]/(Oo|Oo|Ic.Ic) .
         (fib![n-1,c] | fib![n-2,c] |
          c?[x].c?[y].r![x+y])
```

- r は一回の送信に使用　⇒　O_o
- c は二回の送受信に使用　⇒　$O_o|O_o|I_c.I_c$

⇓

- deadlock-freedom が保証される