

# 多段階 let 挿入を行うコード生成言語の 型システムの設計

大石純平  
亀山幸義

筑波大学 大学院 コンピュータ・サイエンス専攻

2016/9/9

# アウトライン

- ① 概要
- ② 研究の背景
- ③ 研究の目的
- ④ 研究の内容
- ⑤ まとめと今後の課題

# アウトライン

- ① 概要
- ② 研究の背景
- ③ 研究の目的
- ④ 研究の内容
- ⑤ まとめと今後の課題

プログラムを生成するプログラミング言語 (=コード生成言語)  
の安全性を保証する研究

プログラムを生成するプログラミング言語 (=コード生成言語)  
の安全性を保証する研究

- 効率的なコードの生成
- 安全性の保証

プログラムを生成するプログラミング言語 (=コード生成言語)  
の安全性を保証する研究

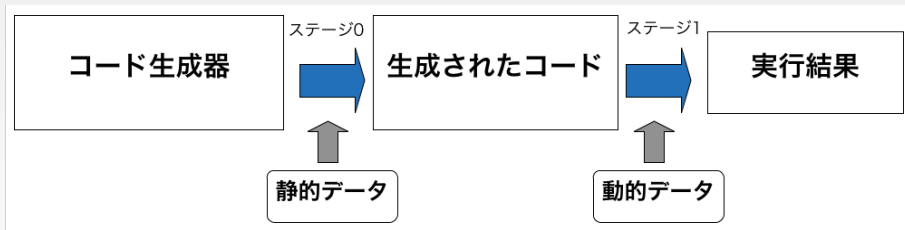
- 効率的なコードの生成
- 安全性の保証

⇒ 多段階 let 挿入を効率的かつ安全に扱うための型システムを  
構築

# アウトライン

- ① 概要
- ② 研究の背景**
- ③ 研究の目的
- ④ 研究の内容
- ⑤ まとめと今後の課題

# コード生成（段階的計算）



- コード生成段階とコード実行段階
  - 生成前の段階で、生成後のコードの安全性を保証する
- ⇒ 段階的計算をサポートするプログラム言語 ⇒ コード生成言語



# power 関数のコード化

```
power  $x$   $n$  =  $x$                                 if  $n = 1$   
               $x * \text{power } x (n - 1)$             if  $n > 1$ 
```

# power 関数のコード化

```
power  $x$   $n$  =  $x$                                 if  $n = 1$   
               $x * \text{power } x (n - 1)$             if  $n > 1$ 
```

$n = 8$  に特化したコードの生成を行う

```
gen_power  $x$  8 =  $x * x * x * x * x * x * x * x$ 
```

# power 関数のコード化

|                                 |            |
|---------------------------------|------------|
| $\text{power } x \ n = x$       | if $n = 1$ |
| $x * \text{power } x \ (n - 1)$ | if $n > 1$ |

$n = 8$  に特化したコードの生成を行う

$\text{gen\_power } x \ 8 = x * x * x * x * x * x * x * x$

$\text{gen\_power } x \ 8$  は  $\text{power } x \ 8$  より高速

- 関数呼び出しがない
- 条件式がない

# コード生成の利点と課題

## 利点

- 「保守性・再利用性の高さ」と「実行性能の高さ」の両立

# コード生成の利点と課題

## 利点

- 「保守性・再利用性の高さ」と「実行性能の高さ」の両立

## 課題

- パラメータに応じて、非常に多数のコードが生成される
  - 生成したコードのデバッグが容易ではない
- ⇒ コード生成の前に安全性を保証したい

# 多段階 let 挿入

- 入れ子になった for ループなどを飛び越えたコード移動を許す仕組み
- ループ不変式の移動によって、効率的なコード生成に必要なプログラミング技法

## 多段階 let 挿入, let 挿入の例

```
for  $i = 0$  to  $n$  do  
  for  $j = 0$  to  $m$  do  
    let  $y = t$  in  
       $a[i, j] \leftarrow b[i] + y$ 
```

## 多段階 let 挿入, let 挿入の例

```
for  $i = 0$  to  $n$  do  
  for  $j = 0$  to  $m$  do  
    let  $y = t$  in  
       $a[i, j] \leftarrow b[i] + y$ 
```

多段階 let 挿入



let  $y = t$  in      —  $t$  が  $i$  にも  $j$  にも依存しない式

```
for  $i = 0$  to  $n$  do  
  for  $j = 0$  to  $m$  do  
     $a[i, j] \leftarrow b[i] + y$ 
```



## 多段階 let 挿入, let 挿入の例

```
for  $i = 0$  to  $n$  do  
  for  $j = 0$  to  $m$  do  
    let  $y = t$  in  
       $a[i, j] \leftarrow b[i] + y$ 
```

普通の let 挿入



```
for  $i = 0$  to  $n$  do  
  let  $y = t$  in      —  $t$  が  $i$  にのみ依存し  $j$  には依存しない式  
    for  $j = 0$  to  $m$  do  
       $a[i, j] \leftarrow b[i] + y$ 
```

# 危険な例

# 危険な例

**for**  $i = 0$  **to**  $n$  **do**

**for**  $j = 0$  **to**  $m$  **do**

**let**  $y = a[i] + b[j]$  **in** —  $t$  が  $i, j$  に依存した式

$a[i, j] \leftarrow b[i] + y$

# 危険な例

```
for  $i = 0$  to  $n$  do  
  for  $j = 0$  to  $m$  do  
    let  $y = a[i] + b[j]$  in —  $t$  が  $i, j$  に依存した式  
     $a[i, j] \leftarrow b[i] + y$ 
```

多段階 let 挿入



```
let  $y = a[i] + b[j]$  in —  $t$  が  $i, j$  に依存した式  
  for  $i = 0$  to  $n$  do  
    for  $j = 0$  to  $m$  do  
       $a[i, j] \leftarrow b[i] + y$ 
```

# アウトライン

- ① 概要
- ② 研究の背景
- ③ 研究の目的**
- ④ 研究の内容
- ⑤ まとめと今後の課題

# 研究の目的

## 表現力と安全性を兼ね備えたコード生成言語の構築

- 表現力: 多段階 `let` 挿入, メモ化等の技法を表現
- 安全性: 生成されるコードの一定の性質を静的に検査

# 研究の目的

## 表現力と安全性を兼ね備えたコード生成言語の構築

- 表現力: 多段階 `let` 挿入, メモ化等の技法を表現
- 安全性: 生成されるコードの一定の性質を静的に検査

## 本研究: 簡潔で強力なコントロールオペレータに基づくコード生成体系の構築

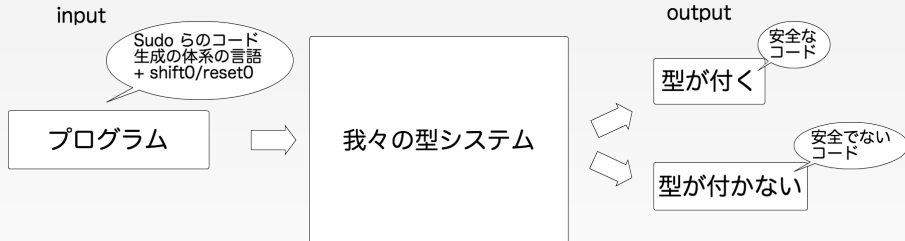
- コントロールオペレータ `shift0/reset0` を利用し, `let` 挿入などのコード生成技法を表現
- 型システムを構築して型安全性を保証

# アウトライン

- ① 概要
- ② 研究の背景
- ③ 研究の目的
- ④ 研究の内容**
- ⑤ まとめと今後の課題



# 本研究の手法



# コード生成器と生成されるコード

## コード生成器

```
... for  $i = 0$  to  $n$  do  
... for  $j = 0$  to  $m$  do  
... let  $y = t$  in  
...  $(a[i, j] \leftarrow b[i] + y)$ 
```

$\Rightarrow$

## 生成されるコード

```
let  $y = t$  in  
  for  $i = 0$  to  $n$  do  
    for  $j = 0$  to  $m$  do  
       $a[i, j] \leftarrow b[i] + y$ 
```

## shift0/reset0 による多段階 let 挿入

$\text{reset0}(E[\text{shift0 } k \rightarrow e]) \rightarrow e\{k \Leftarrow E\}$

### コード生成器

```
... for  $i = 0$  to  $n$  do  
  ... for  $j = 0$  to  $m$  do  
    ... let  $y = t$  in  
      ...  $(a[i, j] \leftarrow b[i] + y)$ 
```

## shift0/reset0 による多段階 let 挿入

$\text{reset0}(E[\text{shift0 } k \rightarrow e]) \rightarrow e\{k \Leftarrow E\}$

### コード生成器

**reset0** for  $i = 0$  to  $n$  do

**reset0** for  $j = 0$  to  $m$  do

**shift0**  $k_2 \rightarrow$  **shift0**  $k_1 \rightarrow$  let  $y = t$  in

$k_1$  ( $k_2$  ( $a[i, j] \leftarrow b[i] + y$ ))

# shift0/reset0 による多段階 let 挿入

$\text{reset0}(E[\text{shift0 } k \rightarrow e]) \rightarrow e\{k \Leftarrow E\}$

## コード生成器

**reset0** for  $i = 0$  to  $n$  do

**reset0** for  $j = 0$  to  $m$  do

**shift0**  $k_2 \rightarrow$  **shift0**  $k_1 \rightarrow$  let  $y = t$  in

$k_1$  ( $k_2$  ( $a[i, j] \leftarrow b[i] + y$ ))

$k_1 =$  for  $i = 0$  to  $n$  do

$k_2 =$  for  $j = 0$  to  $m$  do

## shift0/reset0 による多段階 let 挿入

$\text{reset0}(E[\text{shift0 } k \rightarrow e]) \rightarrow e\{k \leftarrow E\}$

### コード生成器

let  $y = t$  in

$k_1$  ( $k_2$  ( $a[i, j] \leftarrow b[i] + y$ ))

$k_1 =$  for  $i = 0$  to  $n$  do

$k_2 =$  for  $j = 0$  to  $m$  do

## shift0/reset0 による多段階 let 挿入

$\text{reset0}(E[\text{shift0 } k \rightarrow e]) \rightarrow e\{k \leftarrow E\}$

生成されるコード

```
let y = t in
  for i = 0 to n do
    for j = 0 to m do
       $a[i, j] \leftarrow b[i] + y$ 
```

**コード生成前の段階で，安全なコードかどうかを判断する**

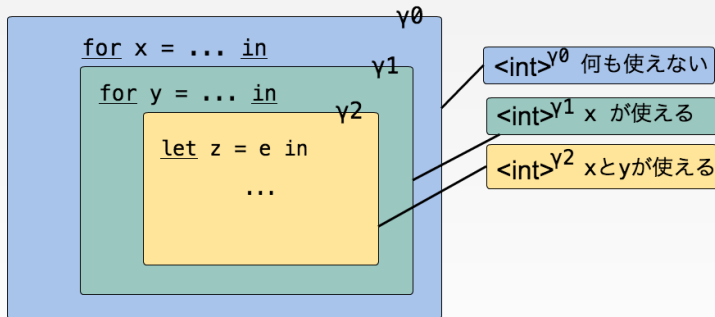


# 環境識別子 EC によるスコープ表現

[Taha+2003] [Sudo+2014]

コードレベルの変数スコープ

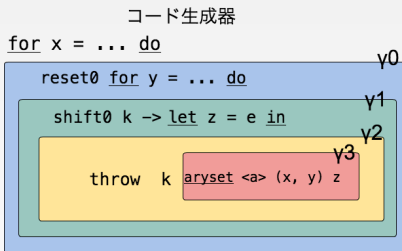
型によるスコープの表現



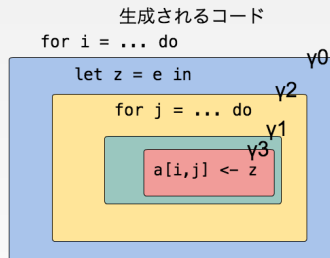
$\gamma_0 \longrightarrow \gamma_1 \longrightarrow \gamma_2$

$\gamma_i \dots$  Refined Environment Classifier

# ECの洗練化（本研究）

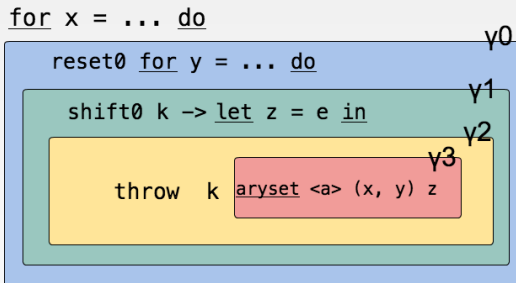


$y_0 \longrightarrow y_1 \longrightarrow y_2 \longrightarrow y_3$



# ECの洗練化（本研究）

コード生成器



$y_0 \longrightarrow y_1 \longrightarrow y_2 \longrightarrow y_3$

かつ

$y_0 \longrightarrow y_2 \longrightarrow y_1 \longrightarrow y_3$

# ECの洗練化（本研究）

コード生成器

for x = ... do

reset0 for y = ... do

shift0 k -> let z = e in

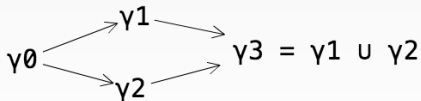
throw k aryset <a> (x, y) z

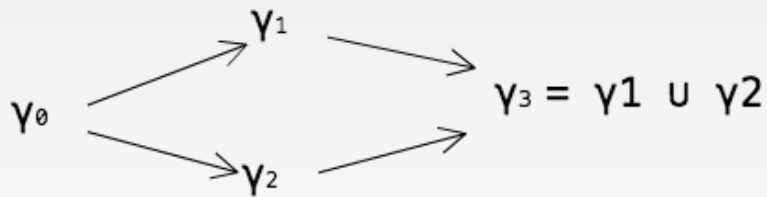
$\gamma_0$

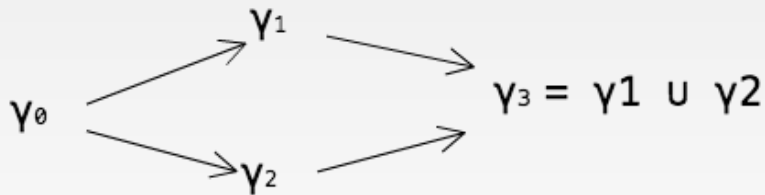
$\gamma_1$

$\gamma_2$

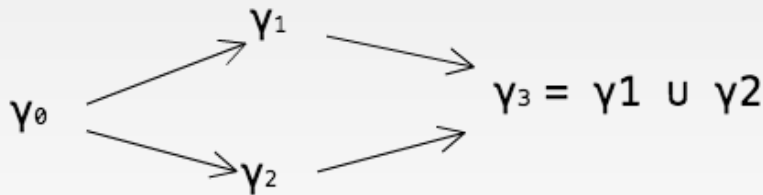
$\gamma_3$



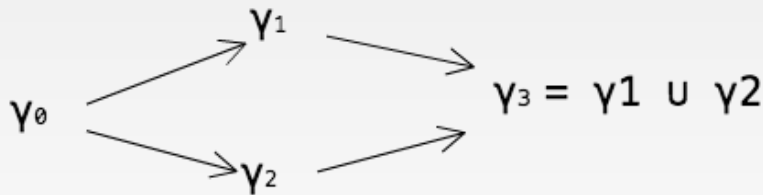




- $\gamma_1$  のコードレベル変数は  $\gamma_2$  では使えない

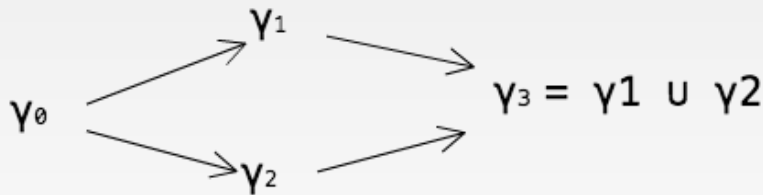


- $\gamma_1$  のコードレベル変数は  $\gamma_2$  では使えない
- $\gamma_2$  のコードレベル変数は  $\gamma_1$  では使えない



- $\gamma_1$  のコードレベル変数は  $\gamma_2$  では使えない
- $\gamma_2$  のコードレベル変数は  $\gamma_1$  では使えない
- $\gamma_1, \gamma_2$  のコードレベル変数は  $\gamma_3$  で使える





- $\gamma_1$  のコードレベル変数は  $\gamma_2$  では使えない
  - $\gamma_2$  のコードレベル変数は  $\gamma_1$  では使えない
  - $\gamma_1, \gamma_2$  のコードレベル変数は  $\gamma_3$  で使える
- ⇒ Sudo らの体系に  $\cup$  を追加

# コード生成+shift0/reset0 の型システム (の一部)

コードレベルのラムダ抽象:

$$\frac{\Gamma, \gamma_1 \geq \gamma, x : \langle t_1 \rangle^{\gamma_1} \vdash e : \langle t_2 \rangle^{\gamma_1}; \sigma}{\Gamma \vdash \underline{\lambda}x.e : \langle t_1 \rightarrow t_2 \rangle^{\gamma}; \sigma} \quad (\gamma_1 \text{ is eigen var})$$

reset0:

$$\frac{\Gamma \vdash e : \langle t \rangle^{\gamma}; \langle t \rangle^{\gamma}, \sigma}{\Gamma \vdash \mathbf{reset0} \ e : \langle t \rangle^{\gamma}; \sigma}$$

shift0:

$$\frac{\Gamma, k : (\langle t_1 \rangle^{\gamma_1} \Rightarrow \langle t_0 \rangle^{\gamma_0})\sigma \vdash e : \langle t_0 \rangle^{\gamma_0}; \sigma \quad \Gamma \models \gamma_1 \geq \gamma_0}{\Gamma \vdash \mathbf{shift0} \ k.e : \langle t_1 \rangle^{\gamma_1}; \langle t_0 \rangle^{\gamma_0}, \sigma}$$

throw:

$$\frac{\Gamma \vdash v : \langle t_1 \rangle^{\gamma_1 \cup \gamma_2}; \sigma \quad \Gamma \models \gamma_2 \geq \gamma_0}{\Gamma, k : (\langle t_1 \rangle^{\gamma_1} \Rightarrow \langle t_0 \rangle^{\gamma_0})\sigma \vdash \mathbf{throw} \ k \ v : \langle t_0 \rangle^{\gamma_2}; \sigma}$$

# 型が付く例/付かない例

## コード生成器

$e =$  reset0 for  $i = 0$  to  $n$  in  
    reset0 for  $j = 0$  to  $m$  in  
        shift0  $k_2 \rightarrow$  shift0  $k_1 \rightarrow$  let  $y = t$  in  
             $k_1$  ( $k_2$  ( $a[\underline{i}][\underline{j}] = b[\underline{i}] + y$ ))

# 型が付く例/付かない例

## コード生成器

```
e = reset0 for i = 0 to n in  
      reset0 for j = 0 to m in  
            shift0 k2 → shift0 k1 → let y = t in  
              k1 (k2 (a[i][j] = b[i] + y))
```

## 生成されるコード



```
e  $\rightsquigarrow^*$  let y = a[i][j] in  
      for i = 0 to n in  
        for j = 0 to m in  
          a[i][j] = b[i] + y
```



```
e  $\rightsquigarrow^*$  let y = 7 in  
      for i = 0 to n in  
        for j = 0 to m in  
          a[i][j] = b[i] + y
```

あれもこれもできる

# アウトライン

- ① 概要
- ② 研究の背景
- ③ 研究の目的
- ④ 研究の内容
- ⑤ **まとめと今後の課題**

# まとめと今後の課題

## まとめ

- コード生成言語の型システムに `shift0/reset0` を組み込んだ型システムの設計を完成させた.
- 安全なコードの場合に型が付くこと, 安全でないコードの場合には型が付かないように意図通りに型システムが設計できていることをみた

## 今後の課題

- 設計した型システムの健全性の証明 (Subject reduction 等)
- 型推論アルゴリズムの開発
- 言語の拡張
  - グローバルな参照 (OCaml の `let ref`)
  - 生成したコードの実行 (MetaOCaml の `run`)

# APPENDIX



## ⑥ 健全性の証明

# 健全性の証明 (Subject Reduction)

型安全性 (型システムの健全性; Subject Reduction 等の性質) を厳密に証明する.

## Subject Redcution Property

$\Gamma \vdash M : \tau$  が導ければ (プログラム  $M$  が型検査を通れば),  $M$  を計算して得られる任意の  $N$  に対して,  $\Gamma \vdash N : \tau$  が導ける ( $N$  も型検査を通り,  $M$  と同じ型, 同じ自由変数を持つ)