

多段階 let 挿入を行うコード生成言語の 設計

大石純平

筑波大学 大学院
プログラム論理研究室

2016/7/12

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

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

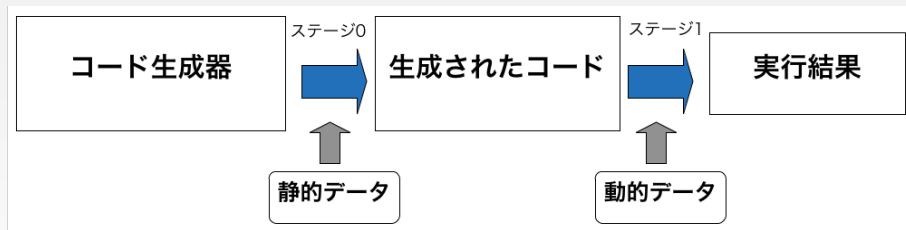
- 効率的なコードの生成
- 安全性の保証

⇒ 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

段階的計算 (Staged Computation)



- コード生成ステージとコード実行ステージ
- ⇒ 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

$\text{power } x \ n = x$	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

$\text{gen_power } x \ 8 = x * x * x * x * x * x * x * x$

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

- コード生成プログラムが、安全なコードのみを生成する事を静的に保証
- 安全なコード: 構文, 型, 変数束縛が正しいプログラム

- コード生成プログラムが、安全なコードのみを生成する事を静的に保証
- 安全なコード: 構文, 型, 変数束縛が正しいプログラム

しかし多段階 `let` 挿入等を実現する計算エフェクトを含む場合のコード生成の安全性保証は研究途上

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

多段階 let 挿入, let 挿入の例

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

多段階 let 挿入



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

— t が i にも j にも依存しない式

多段階 let 挿入, let 挿入の例

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

普通の let 挿入



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

— t が i にのみ依存し j には依存しない式

多段階 let 挿入, let 挿入の例

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

多段階 let 挿入でも let 挿入でもない



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

— t が i, j に依存した式

プログラミング言語におけるプログラムを制御するプリミティブ

- exception (例外): C++, Java, ML
- call/cc (第一級継続): Scheme, SML/NJ
- shift/reset (限定継続): Racket, Scala, OCaml
 - 1989 年以降多数研究がある
 - コード生成における let 挿入が実現可能
- shift0/reset0
 - 2011 年以降研究が活発化.
 - コード生成における多段階 let 挿入が可能

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

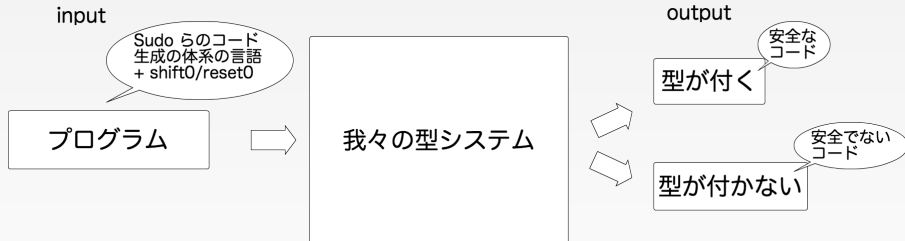
本研究: 簡潔で強力なコントロールオペレータに基づくコード生成体系の構築

- 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



コード生成前

```
cfor  $i = 0$  to  $n$  in  
  cfor  $j = 0$  to  $m$  in  
    clet  $y = t$  in  
       $(a[i][j] = b[i] + y)$ 
```

コード生成前

```
reset0 cfor  $i = 0$  to  $n$  in  
  reset0 cfor  $j = 0$  to  $m$  in  
    shift0  $k_2 \rightarrow$  shift0  $k_1 \rightarrow$  clet  $y = t$  in  
      throw  $k_1$  (throw  $k_2$  ( $a[i][j] = b[i] + y$ ))
```

shift0/reset0 による多段階 let 挿入

コード生成前

```
reset0 cfor  $i = 0$  to  $n$  in  
  reset0 cfor  $j = 0$  to  $m$  in  
    shift0  $k_2 \rightarrow$  shift0  $k_1 \rightarrow$  clet  $y = t$  in  
      throw  $k_1$  (throw  $k_2$  ( $a[\underline{i}][\underline{j}] = b[\underline{i}] + y$ ))
```

```
 $k_1 =$  cfor  $i = 0$  to  $n$  in  
 $k_2 =$  cfor  $j = 0$  to  $m$  in
```

コード生成前

clet $y = t$ in
throw k_1 (throw k_2 ($a[\underline{i}][\underline{j}] = b[\underline{i}] + y$))

$k_1 =$ cfor $i = 0$ to n in

$k_2 =$ cfor $j = 0$ to m in

コード生成前

```
clet  $y = t$  in  
  cfor  $i = 0$  to  $n$  in  
    cfor  $j = 0$  to  $m$  in  
       $(a[i][j] = b[i] + y)$ 
```


型が付く例/付かない例

コード生成前

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

型が付く例/付かない例

コード生成前

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

$e \rightsquigarrow^* \text{clet } y = t \text{ in}$
 cfor $i = 0$ to n in
 cfor $j = 0$ to m in
 $(a[\underline{i}][\underline{j}] = b[\underline{i}] + y)$

型が付く例/付かない例

コード生成前

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

```
 $e \rightsquigarrow^*$  clet  $y = t$  in  
  cfor  $i = 0$  to  $n$  in  
    cfor  $j = 0$  to  $m$  in  
      ( $a[\underline{i}][\underline{j}] = b[\underline{i}] + y$ )
```

$t = \%7$ のとき e は型が付く

$t = a[\underline{i}][\underline{j}]$ や $t = b[\underline{j}]$ のとき e は型が付かない

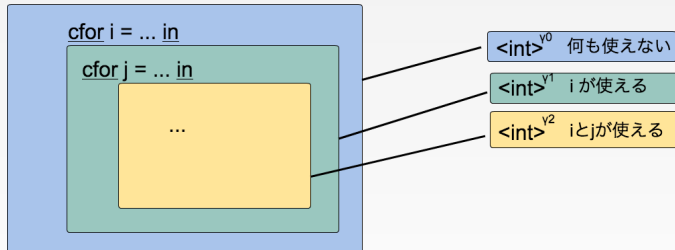
**安全なコードにのみ型をつける
にはどうすればよいか**

環境識別子 EC によるスコープ表現

[Taha+2003] [Sudo+2014]

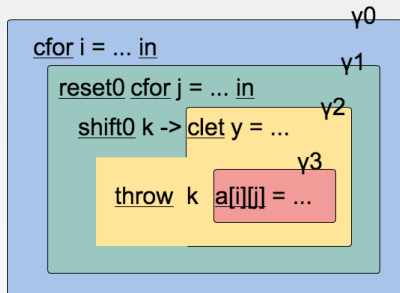
コードレベルの変数スコープ

型によるスコープの表現

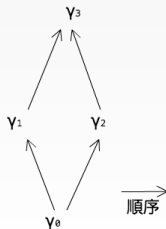
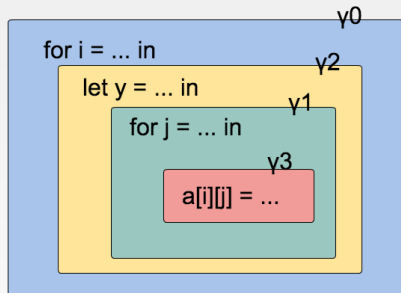


γ_i ...Refined Environment Classifier

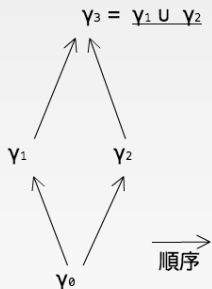
ECの洗練化（本研究）



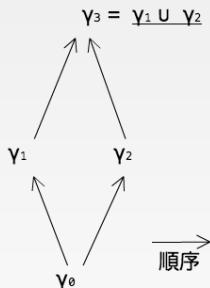
計算
 \Rightarrow



ECのジョイン

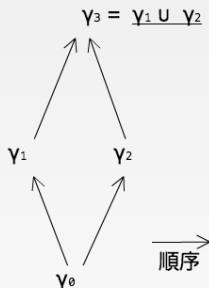


ECのジョイン

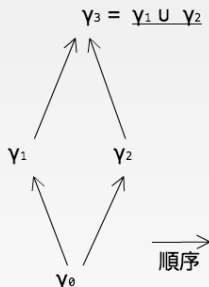


- γ_1 のコードレベル変数は γ_2 では使えない

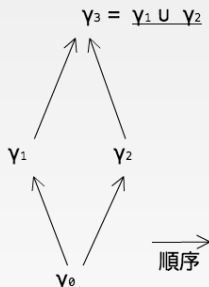
ECのジョイン



- γ_1 のコードレベル変数は γ_2 では使えない
- γ_2 のコードレベル変数は γ_1 では使えない



- γ_1 のコードレベル変数は γ_2 では使えない
- γ_2 のコードレベル変数は γ_1 では使えない
- γ_1, γ_2 のコードレベル変数は γ_3 で使える



- γ_1 のコードレベル変数は γ_2 では使えない
 - γ_2 のコードレベル変数は γ_1 では使えない
 - γ_1, γ_2 のコードレベル変数は γ_3 で使える
- ⇒ Sudo らの体系に \cup を追加

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行った
- その型システムによって型が付く場合と付かない場合の例をみた

今後の課題

- 設計した型システムの健全性の証明 (Subject reduction 等) を行い, 実装を完成させる

APPENDIX

アウトライン

⑥ コード生成+shift0/reset0 の型システム

⑦ 健全性の証明

型システムの一部

Typing rule for code-level let (derived rule):

$$\frac{\Gamma \vdash e_1 : \langle t_1 \rangle^\gamma \quad \Gamma, \gamma_1 \geq \gamma, x : \langle t_1 \rangle^{\gamma_1} \vdash e_2 : \langle t_2 \rangle^{\gamma_1}}{\Gamma \vdash \textbf{clet } x = e_1 \textbf{ in } e_2 : \langle t_2 \rangle^\gamma} \quad (\gamma_1 \text{ is eigen var})$$

Typing rule for code-level reset0:

$$\frac{\Gamma \vdash e : \langle t \rangle^\gamma}{\Gamma \vdash \textbf{reset0 } e : \langle t \rangle^\gamma}$$

Typing rule for code-level shift0:

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

Typing rule for code-level throw:

$$\frac{\Gamma, \gamma_3 \geq \gamma_1, \gamma_3 \geq \gamma_2 \vdash e : \langle t_1 \rangle^{\gamma_3} \quad \Gamma \models \gamma_2 \geq \gamma_0}{\Gamma, k : (\langle t_1 \rangle^{\gamma_1} \Rightarrow \langle t_0 \rangle^{\gamma_0}) \vdash \textbf{throw } k e : \langle t_0 \rangle^{\gamma_2}} \quad (\gamma_3 \text{ is eigen var})$$

⑥ コード生成+shift0/reset0 の型システム

⑦ 健全性の証明

健全性の証明 (Subject Reduction)

3. 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

$\Gamma \vdash M : \sigma$ 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, $\Gamma \vdash N : \sigma$ 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.)