

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

2015/10/22

- ① 研究の背景
- ② 研究の目的
- ③ 研究の内容
- ④ まとめと今後

2 / 33

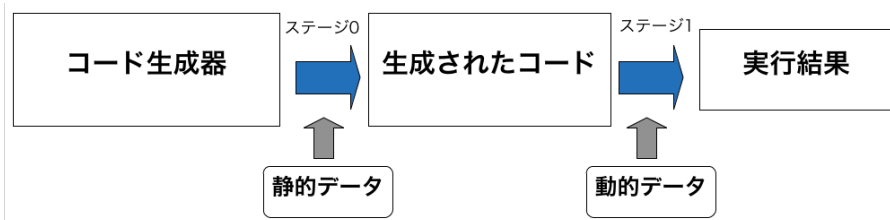
アウトライン

- ① 研究の背景
段階的計算 (コード生成)
効率の良いコードの生成例
段階的計算の課題
- ② 研究の目的
- ③ 研究の内容
- ④ まとめと今後

アウトライン

- ① 研究の背景
段階的計算 (コード生成)
効率の良いコードの生成例
段階的計算の課題
- ② 研究の目的
- ③ 研究の内容
- ④ まとめと今後

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
 - 「保守性・再利用性の高さ」と「実行性能の高さ」の両立
- ⇒ 段階的計算をサポートするプログラム言語

5 / 33

アウトライン

① 研究の背景

段階的計算 (コード生成)
効率の良いコードの生成例
段階的計算の課題

② 研究の目的

③ 研究の内容

④ まとめと今後

6 / 33

例 1: 行列の積

$n \times n$ 行列の積: $C = A B$

```
for i = 1 to n
  for k = 1 to n
    for j = 1 to n
      c[i][j] += a[i][k] * b[k][j]
```

ループの展開:

```
for i = 1 to n
  for k = 1 to n
    for j = 1 to n step 4
      c[i][j] += a[i][k] * b[k][j]
      c[i][j+1] += a[i][k] * b[k][j+1]
      c[i][j+2] += a[i][k] * b[k][j+2]
      c[i][j+3] += a[i][k] * b[k][j+3]
```

7 / 33

共通項のくり出し

```
for i = 1 to n
  for k = 1 to n
    for j = 1 to n step 4
      c[i][j] += a[i][k] * b[k][j]
      c[i][j+1] += a[i][k] * b[k][j+1]
      c[i][j+2] += a[i][k] * b[k][j+2]
      c[i][j+3] += a[i][k] * b[k][j+3]
```

```
for i = 1 to n
  for k = 1 to n
    let t = a[i][k] in
    for j = 1 to n step 4
      c[i][j] += t * b[k][j]
      c[i][j+1] += t * b[k][j+1]
      c[i][j+2] += t * b[k][j+2]
      c[i][j+3] += t * b[k][j+3]
```

8 / 33

let 挿入 [Danvy 1996]

let 挿入

```
for i = 1 to n
  for k = 1 to n
    let t = a[i][k] in
      for j = 1 to n step 4
        c[i][j] += t * b[k][j]
        c[i][j+1] += t * b[k][j+1]
        c[i][j+2] += t * b[k][j+2]
        c[i][j+3] += t * b[k][j+3]
```

誤りのある let 挿入

```
for i = 1 to n
  let t = a[i][k] in      --- k is not bound
  for k = 1 to n in
    for j = 1 to n step 4
      c[i][j] += t * b[k][j]
      c[i][j+1] += t * b[k][j+1]
```

9/33

例2: assert 挿入

```
let main a i =
  sort a;      -- some complex computation
  assert (1 <= i && i <= length a);
  a[i];
```

時間のかかる計算 (sort) の前に境界チェックを行いたい。

```
let main a i =
  assert (1 <= i && i <= length a);
  sort a;      -- some complex computation
  a[i];
```

例2: モジュラーな配列アクセス

```
let access a i =
  assert (1 <= i && i <= length a);
  a[i];

let main a i =
  sort a;      -- some complex computation
  access a i;
```

assert e は e の条件をチェック ⇒ 適切なエラー処理をする。
上記を展開すると ...

```
let main a i =
  sort a;      -- some complex computation
  access a i;
```

```
let main a i =
  sort a;      -- some complex computation
  assert (1 <= i && i <= length a);
  a[i];
```

10/33

例3: 深いネストでの assert 挿入

```
for j = 1 to n
  for k = 1 to n
    c[j][k] := (access a j) * (access b k)
```

異なる地点へ、assert を挿入したい。

```
for j = 1 to n
  assert (1 <= j && j <= length a);
  for k = 1 to n
    assert (1 <= k && k <= length b);
    c[j][k] := a[j] * b[k]
```

安全性の静的保証

$C = A \cdot B$

↓ : コード生成 (コード変換, 最適化を含む)

```
for i = 1 to n
  for k = 1 to n
    let t = a[i][k]
    for j = 1 to n step 4
      c[i][j] += t * b[k][j]
      c[i][j+1] += t * b[k][j+1]
      c[i][j+2] += t * b[k][j+2]
      c[i][j+3] += t * b[k][j+3]
```

安全性の静的保証

動的に生成されたコードのデバッグは困難

⇒ コード生成の前に安全性を保証したい

13 / 33

コード生成における課題

生成されたコードの信頼性 (正しさ)

- 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

従来研究

- 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保証
- let 挿入等を実現する **計算エフェクトを含む場合の安全性保証は研究途上**

15 / 33

アウトライン

① 研究の背景

段階的計算 (コード生成)

効率の良いコードの生成例

段階的計算の課題

② 研究の目的

③ 研究の内容

④ まとめと今後

14 / 33

アウトライン

① 研究の背景

② 研究の目的

③ 研究の内容

④ まとめと今後

16 / 33

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: より簡潔でより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
- 種々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

17 / 33

アウトライン

- ① 研究の背景
- ② 研究の目的
- ③ 研究の内容
コントロールオペレータ
本研究の着想
- ④ まとめと今後

18 / 33

アウトライン

- ① 研究の背景
- ② 研究の目的
- ③ 研究の内容
コントロールオペレータ
本研究の着想
- ④ まとめと今後

19 / 33

コントロールオペレータ

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):
- call/cc (第一級継続):
- shift/reset (限定継続):
 - 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

20 / 33

コントロールオペレータ shift0/reset0

```
reset0 (3 + (shift0 k -> 5 + k(7)))
```

```
reset0 (3 + (shift0 k -> 5 + k(7)))  
⇒ (5 + k(7))
```

```
reset0 (3 + (shift0 k -> 5 + k(7)))  
⇒ (5 + k(7)) where k = reset0 (3 + [])
```

```
reset0 (3 + (shift0 k -> 5 + k(7)))  
⇒ (5 + k(7)) where k = reset0 (3 + [])  
⇒ (5 + (3 + 7)) ⇒ 15
```

21 / 33

shift0/reset0 による let 挿入

```
reset0 (3 + (shift0 k -> let x = 5 in k(x)))
```

```
reset0 (3 + (shift0 k -> let x = 5 in k(x)))  
⇒ (let x = 5 in k(x))
```

```
reset0 (3 + (shift0 k -> let x = 5 in k(x)))  
⇒ (let x = 5 in k(x))  
   where k = reset0 (3 + [])
```

```
reset0 (3 + (shift0 k -> let x = 5 in k(x)))  
⇒ (let x = 5 in k(x))  
   where k = reset0 (3 + [])  
⇒ (let x = 5 in 3 + x)
```

22 / 33

shift0/reset0 による多段階 let 挿入

```
for j = 1 to n  
  for k = 1 to n  
    c[j][k] := (access a j) * (access b k)
```

```
for j = 1 to n  
  reset0(  
    for k = 1 to n  
      reset0(  
        c[j][k] := (access2 a j) * (access1 b k)))
```

⇒

```
for j = 1 to n  
  assert (1 <= j && j <= length a);  
  for k = 1 to n  
    assert (1 <= k && k <= length b);  
    c[j][k] := a[j] * b[k]
```

23 / 33

shift0/reset0 による多段階 let 挿入

```
for j = 1 to n reset0(  
  for k = 1 to n reset0(  
    c[j][k] := (access2 a j) * (access1 b k)))
```

```
let access2 a j =  
  shift0 k1 -> shift0 k2 ->  
    assert (1 <= j && j <= length a);  
    k2 (k1 (a[j]))
```

```
k1 = reset0( c[j][k] := [ ] * (access1 b k) )  
k2 = reset0( for k = 1 to n [ ] )
```

24 / 33

shift0/reset0 による多段階 let 挿入

```
for j = 1 to n  reset0(  
  for k = 1 to n  reset0(  
    c[j][k] := (access2 a j) * (access1 b k)))
```

```
for j = 1 to n  
  assert (1 <= j && j <= length a);  
  k2 (k1 (a[j]))
```

```
for j = 1 to n  
  assert (1 <= j && j <= length a);  
  for k = 1 to n  
    c[j][k] := [] * (access1 b k)
```

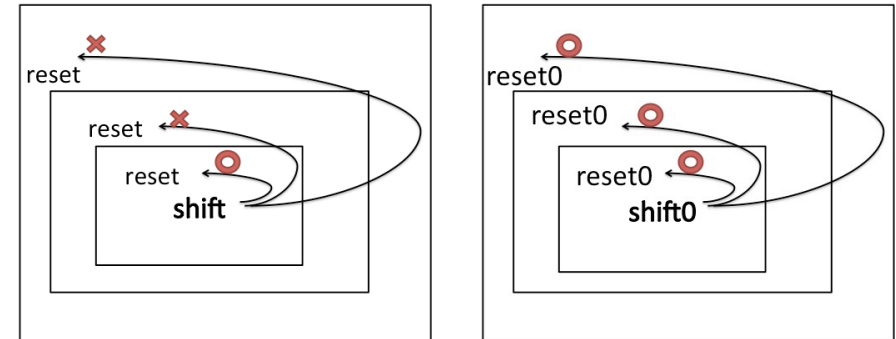
25 / 33

shift0/reset0 と shift/reset の違い

shift0 は、直近の reset0 より、遠くの reset0 までアクセスできる。

shift0/reset0: $\langle K[S_0 f.e] \rangle \rightsquigarrow e\{f/\lambda x.\langle K[x] \rangle\}$

shift/reset: $\langle K[S f.e] \rangle \rightsquigarrow \langle e\{f/\lambda x.\langle K[x] \rangle\} \rangle$



26 / 33

アウトライン

① 研究の背景

② 研究の目的

③ 研究の内容

コントロールオペレータ
本研究の着想

④ まとめと今後

本研究の着想

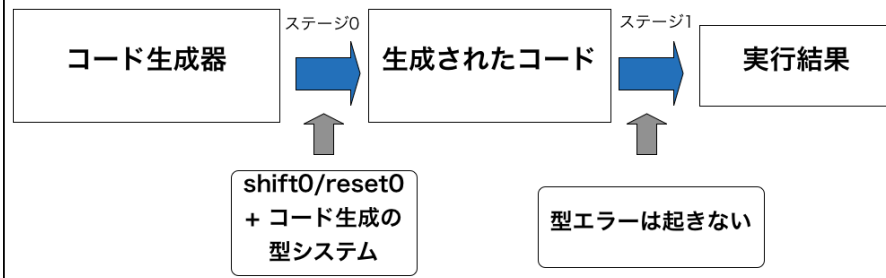
- shift/reset では、多段階の let 挿入は実現できない ⇒ shift0/reset0 では実現可能
- 階層化 shift/reset では実現可能 ⇒ 型システムが複雑、シンプルな意味論がない

shift0/reset0 は単純な CPS 変換, CPS 意味論をもち、多段階 let 挿入が実現できるところに着目 ⇒ コード生成への応用

27 / 33

28 / 33

型システム



- MSP での型システムの健全性とは、プログラム生成前に、型検査に通っていれば、生成後のコードに型エラーは絶対に起きないようなシステムである。
- 計算エフェクト（コントロールオペレータ）などがあるコード生成言語の型安全性は難しい課題であり多数の研究がある。

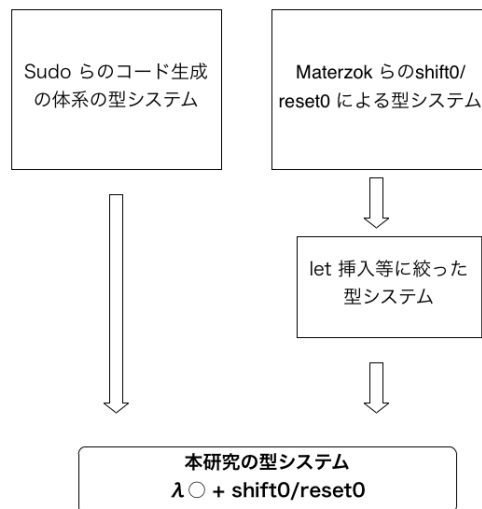
29 / 33

研究項目

1. shift0/reset0 を持つプログラム生成のための体系の構築
 - 型システムと操作的意味論の構築
 - 型の健全性の証明
2. shift0/reset0 を持つプログラム生成のための言語の設計と実装
 - 抽象機械による実装
 - 効率のよいコード生成プログラムの例の作成

30 / 33

本研究の手法



31 / 33

アウトライン

- ① 研究の背景
- ② 研究の目的
- ③ 研究の内容
- ④ まとめと今後

32 / 33

まとめと今後

- 多段階 let 挿入が shift0/reset0 で記述可能なことを見た.
shift0/reset0 を導入した言語を考えると従来より, 簡潔で,
検証しやすい体系ができるということを提案した.
- Sudo らのコード生成言語の型システムを利用し,
shift0/reset0 を組み込んだ体系について検討中である.
- 今後, 型システムの設計を完成させ健全性の証明を行う.

33 / 33

アウトライン

⑤ コード生成言語
安全性
型システム

⑥ 本研究の型システム

35 / 33

APPENDIX

アウトライン

⑤ コード生成言語
安全性
型システム

⑥ 本研究の型システム

36 / 33

コード生成言語の安全性

```
let rec power n x =  
  if n = 0 then 1  
  else x * power (n-1) x  
in power 3 2 ==> 8
```

```
let rec gen_power n x =  
  if n = 0 then <1>  
  else < ~x * ~(gen_power (n-1) x)>  
in gen_power 3 <2> ==> <2 * 2 * 2>
```

```
let rec wrong n x =  
  if n = 0 then <1>  
  else < ~x && ~(wrong (n-1) x)> ==> error
```

(wrong 3 <2>) を実行することはない.

37 / 33

アウトライン

⑤ コード生成言語 安全性 型システム

⑥ 本研究の型システム

コード生成言語の型システム

```
let rec power n x =  
  if n = 0 then 1  
  else x * power (n-1) x  
==> power : int -> int -> int
```

```
let cde = < 3 + 5>  
==> cde : int code
```

```
let f x = < ~x + 7>  
==> f : int code -> int code
```

```
let rec gen_power n x =  
  if n = 0 then <1>  
  else < ~x * ~(gen_power (n-1) x)>  
==> gen_power : int -> int code -> int code
```

生成されたコードの内部の型付けも行われる.

39 / 33

アウトライン

⑤ コード生成言語

⑥ 本研究の型システム

38 / 33

40 / 33

本研究の型システム(1)

1. Sudo らの型システムのアイデアを利用: 変数スコープを表す変数 a_1, a_2, \dots を使って, 型の中で変数スコープを表す.

```
let f x = <~x + 7>
==> f : int codea1 -> int codea1
```

```
let g = <fun y -> y + 7>
==> g : (int -> int) codea2
```

```
let h = fun x -> <fun y -> ~x + y>
==> h : int codea1 -> (int -> int) codea2
      where a2 < a1
```

変数スコープの包含関係を, 変数 a_1, a_2, \dots に対する順序で表現

41 / 33

本研究の型システム(2)

2. let 挿入において shift0/reset0 が型安全性を保つ条件を見つける

```
... (reset0 ( ... (shift0 k -> ... (k ...)))
a0          a1          a2          a3
```

let 挿入: a_2 にある let を reset0 の場所へ移動する.

⇒ a_1 と a_2 の場所を交換しても, 型エラーが起きない条件を同定し, その条件のもとで, 型安全性を証明すればよい.

⇒ a_1 で生成される変数は a_2 で使えないよう, 条件を付ければよい.

42 / 33

本研究の型システム(3)

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.)

43 / 33