多段階 let 挿入を行うコード生成言語の設計

システム情報工学研究科 コンピュータサイエンス専攻
博士課程前期 1 年 201520621 大石純平
指導教員 亀山幸義

2015 年 10 月 22 日

1 はじめに

コード生成法は，プログラムの生産性・保守性と実行性能の高さを両立させられるプログラミング手法として有力なものである．本研究室では，コード生成法をサポートするプログラム言語の信頼性・安全性を高める研究を行ってきている．すなわち，プログラム生成を行うことによって生成されるプログラムが安全に実行されることを，プログラムの生成段階より早い段階，すなわちプログラム生成を行うプログラムのコンパイル段階で検査することのできる言語体系およびシステムを構築することを目標としている．

ここでいう安全性は，構文的に正しいプログラムであること，文字列同士の加算や乗算を決して行わない等の通常の型安全性を満たすことのほか，自由変数やプログラム特化後において利用できない変数に依存したプログラムを生成しないという，変数や変数スコープに関する性質を含む概念である．

この研究での大きな課題は，従来のコード生成のためのプログラミング言語の多くが，純粋なラムダ計算に基づく関数型プログラミング言語を基礎としており，効率の良いコードを生成する多くの技法をカバーしていないことである．これを克服する体系，すなわち，効率良いプログラムを記述するための表現力を高めつつ，安全性が担保された体系が求められている．

本研究の目的は，特に多段階 let 挿入といわれる技法をカバーしつつ，安全性が厳密に保証される計算体系の理論を構築し，さらにそれを実現する処理系を実装することを目的とする．このため，比較的最近になって理論的性質が解明されつつある shift0/reset0 というコントロールオペレータに着目し，これをコード生成の体系に追加して得られた体系を構築して，上記の課題を解決することを狙いとする．

2 準備

2.1 マルチステージプログラミング

マルチステージプログラミングとはプログラムを生成する段階や，生成したプログラムを実行する段階など，複数のステージを持つプログラミングの手法である．プログラムを計算対象のデータとして扱うことで，プログラムの効率や，保守性，再利用性の両立が期待できる．例えば生成元のプログラムから，何らかの目的に特化したプログラムを生成を行い，保守や改変をしたい時は，生成元のプログラムに対して行えばよいので，生成後のコードについては手を加える必要が無い．そのような，マルチステージプログラミングを効果的に行うためには，言語レベルで，プログラムを生成，実行などが行える機構を備えることが望ましい．そのような言語として，本研究では，MetaOCaml というマルチステージプログラミングに対応した OCaml の拡張言語を用いる．

2.2 shift0/reset0

継続を扱う命令としてコントロールオペレータというものがある．継続とは，計算のある時点における残りの計算のことである．本研究では，shift0/reset0 というコントロールオペレータを用いる．reset0 は継続の範囲を限定する命令であり，shift0 はその継続を捕獲するための命令である．

shift/reset[1] では，複数の計算エフェクトを含んだプログラムは書くことができない．しかし，階層化 shift/reset や shift0/reset0 はこの欠点を克服している．階層化 shift/reset[1] は，最大レベルの階層を固定する必要があるが，shift0/reset0 では，shift0，reset0 というオペレータだけで，階層を表現する事ができるという利点がある．また，shift0/reset0 は shift/reset よりも単純な CPS 変換で意味が与えられていて，純

粋なラムダ式で表せるために形式的に扱いやすいという利点がある．reset0 は reset0(M) というように表し，継続の範囲を M に限定するという意味となる．shift0 は shift0(fun k -> M) というように表し，直近の reset0 によって限定された継続を k に束縛し，M を実行するという意味となる．以下で，shift0/reset0 の例を掲載する．

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

この例は，let 挿入を shift0/reset0 により可能にする例である．shift0 によって，まず let x = 5 in k(x) が実行される．ここで，k には，直近の reset0 によって限定された継続である reset0 (3 + []) が捕獲されている．その k に x を適用させることで，3 + x が得られる．すると，let x = 5 in k (x) は let x = 5 in 3 + x に評価される．ここで，見方を変えると，reset0 により限定された継続を shift0 内部の k に捕獲したというよりは，let x = 5 in 3 + x が shift0 によるスコープの外部に出てきたとも見える．このように，shift0/reset0 を使うことで，let 挿入が実現できることが分かる．

2.3 shift0/reset0 による多段階 let 挿入

shift/reset は，直近の reset による限定継続のスコープからひとつ上のスコープまでしか継続を捕獲することができないが，shift0/reset0 においては，直近の reset0 内のスコープだけでなく，遠くの reset0 で限定された継続を捕獲することができる．そのことによって，本研究の肝である多段階の let 挿入[1] が可能となる．以下でその例を見ていく．

```
  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)
```

ここで， access2 は以下の様に宣言されている．

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

次に，上記のプログラムがどのように評価されるかを見ていく．

このプログラムに期待しているのは，access2 に定義されている assert 文を適切な位置へ挿入したいということである．そうすることで，無駄な配列外アクセスを二重ループの内側へ入る前に止めることができる．

access2 には入れ子になった shift0 の中身に assert 文が定義されている．それによって，直近の reset0 の限定継続でなく，もう一つ先の reset0 の場所へスコープを飛び越えて assert 文が挿入されていることが分かる．このような操作は，shift/reset では不可能であり，階層的な shift0/reset0 であるからできることである．

3 目的

プログラムによるプログラムの動的な生成を行い，保守性と性能の両立をはかりたい．また，生成するプログラムだけでなく，生成されたプログラムも型の整合性が静的に（生成前に）保証するようにしたい．

コード生成のアプローチとしては，コード生成のプログラムは，高レベルの記述つまり，高階関数，代数的データ型などを利用し，注腸的なアルゴリズムの記述を行う．それによって生成されたコードは低レベルの記述がなされており高性能な実行が可能となる．また，特定のハードウェアや特定のパラメータを仮定したコードなので，様々な環境に対して対応できる．

つまり，生成元のプログラムは抽象度を上げた記述をすることで，色々な状況 (特定のハードウェア，特定のパラメータ) に応じたプログラムを生成することを目指す．そのようにすることで，生成後のコードには手を加えることなく，生成前のプログラムに対してのみ保守や改変をすれば良い．また，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

[1] let 挿入も assert 挿入も本質は変わらないので，ここでは assert 挿入も let 挿入と呼んでいる．

ドに型エラーは絶対に起こらないことが，型システムにより，保証される．

しかし，コード生成において以下の様な信頼性への大きな不安がある．

- 構文的，意味的に正しくないプログラムを生成しやすい
- デバッグが容易ではない
- 効率のよいコード生成に必要な計算エフェクト（今回の場合だと限定継続のひとつである shift0/reset0）を導入すると，従来理論ではコード生成プログラムの安全性は保証されない

効率のよいコードの生成を行うためには例えば，ネストしたループの順序の入れ替えやループ不変項，共通項のくくりだしなどを行う必要がある．それらを実現するためには，コード生成言語に副作用が必要である．

本研究は，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を行う．コントロールオペレータとして前述の shift0/reset0 を用い，コード生成に必要な多段階 let 挿入等の副作用のあるコード生成技法を表現可能にし，生成されるコードの性質を静的に検査するために，型システムを構築し，型安全性を保証する．

4　研究項目

表現力と安全性を兼ね備えたコード生成の体系としては，2009 年の亀山らの研究 [2] が最初である．彼らは，MetaOCaml において shift/reset とよばれるコントロールオペレータを使うスタイルでのプログラミングを提案するとともに，コントロールオペレータの影響が変数スコープを越えることを制限する型システムを構築し，安全性を厳密に保証した．Westbrook ら [3] は同様の研究を Java のサブセットを対象におこなった．須藤ら [4] は，書換え可能変数を持つコード生成体系に対して，部分型付けを導入した型システムを提案して，安全性を保証した．これらの体系は，安全性の保証を最優先した結果，表現力の上での制限が強くなっている．特に，let 挿入とよばれるコード生成技法をシミュレートするためには，shift/reset が必要であるが，複数の場所への let 挿入を許すためには，複数の種類の shift/reset を組み合わせる必要がある．この目的のため，階層的 shift/reset やマルチプロンプト shift/reset といった，shift/reset を複雑にしたコントロールオペレータを考えることができるが，その場合の型システムは非常に複雑になることが予想され，安全性を保証するための条件も容易には記述できない，等の問題点がある．

本研究では，このような問題点を克服するため，shift/reset の意味論をわずかに変更した shift0/reset0 というコントロールオペレータに着目する．このコントロールオペレータは，長い間，研究対象となってこなかったが 2011 年以降，Materzok らは，部分型付けに基づく型システムや，関数的な CPS 変換を与えるなど，簡潔で拡張が容易な理論的基盤をもつことを解明した [5, 6]．特に，shift0/reset0 は shift/reset と同様のコントロールオペレータでありながら，階層的 shift/reset を表現することができる，という点で，表現力が高い．本研究では，これらの事実に基づき，これまでの shift/reset を用いたコード生成体系の知見を，shift0/reset0 を用いたコード生成体系の構築に活用するものである．

5　本研究の手法

shift0/reset0 を持つコード生成言語の型システムの設計を行い，深く入れ子になった内側からの，let 挿入，assertion 挿入の関数プログラミング的実現を目指すのだが，shift0/reset0 は shift/reset より強力であるため，型システムが非常に複雑である．また，コード生成言語の型システムも一定の複雑さを持っている．そのためにそれらを単純に融合させることは困難である．そこで，本研究では，shift0/reset0 の型システムを let 挿入等に絞るように単純化する．その型システムに対して，コード生成言語の型システムを融合させる．型システムの安全性の保証に関しては，Kameyama+ 2009[2] と Sudo+ 2014[4] の手法を利用する．

5.1　本研究の型システム

Sudo らの型システムのアイディアを利用し変数スコープを表す変数 a1,a2,... を使って型の中で変数ス

コープを表すことで，変数スコープの包含関係を変数 a1,a2,... に対する順序で表現する．以下で例を見ていく．

```
let f x = < ~x + 7>
==> f : int code^a1 -> int code^a1
```

上記の例は，f の引数の変数スコープと返り値のスコープが同じである事をゼロレベルを表す a1 という注釈によって表現されている．

```
let g = <fun y -> y + 7>
==> g : (int -> int) code^a2
```

g は (int -> int) code という型であるが，1 レベル (一番外のスコープ) より 1 つ深いために a2 という注釈が加えられている．

```
let h = fun x -> <fun y -> ~x + y>
==> h : int code ^a1 -> (int -> int) code^a2
```

h は int code -> (int -> int) code という型を持つが，引数には a1 返り値には a2 がそれぞれ割り振られている．a2 は a1 よりも 1 つ深いスコープに属する．

また，let 挿入において shift0/reset0 が型安全性を保つ条件を見つけることを例によって見てみる．

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

a2 にある let を reset0 の場所へ移動をしたいとする．そのためには，a1 と a2 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．また，a1 で生成される変数は a2 で使えないように，条件を付ければよい．

また，型安全性 (型システムの健全性; Subject Reduction 等の性質) の厳密な証明を与える．

定理 5.1. 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．)

6 まとめと今後

多段階 let 挿入が shit0/reset0 で記述可能なことを実例によって提示した．また，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アイデアに基づいて，コード生成言語の型システムを構築することを提案した．Sudo らのコード生成言語の型システム [4] 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および型検査器等の実装を行う

参考文献

[1] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pp. 151–160, New York, NY, USA, 1990. ACM.

[2] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: Staging with delimited control. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pp. 111–120, New York, NY, USA, 2009. ACM.

[3] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 400–411, New York, NY, USA, 2010. ACM.

[4] 須藤悠斗, Oleg Kiselyov, 亀山幸義. コード生成のための自然演繹. 日本ソフトウェア科学会第 31 回大会, 2014 年 9 月, 名古屋大学. PPL4-4, 9 ページ.

[5] Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. *SIGPLAN Not.*, Vol. 46, No. 9, pp. 81–93, September 2011.

[6] Marek Materzok and Dariusz Biernacki. A dynamic interpretation of the cps hierarchy. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, Vol. 7705 of *Lecture Notes in Computer Science*, pp. 296–311. Springer Berlin Heidelberg, 2012.