# 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 2 年 201520621 大石純平
指導教員 亀山幸義

2016 年 7 月 12 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．本研究は，コード生成法で必要とされる「多段階 let 挿入」等を簡潔に表現できるコントロールオペレータである shift0/reset0 を持つコード生成言語とその型システムを構築し，生成されたコードの型安全性を保証する．

多段階 let 挿入は，入れ子になった for ループを飛び越えたコード移動を許す仕組みであり，ループ不変式の移動などのために必要である．

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，安全性が保証された体系が求められている．

本研究の目的は，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．

コード生成言語の型安全性に関して，破壊的代入を持つ体系に対する須藤らの研究 [1] 等があるが，本研究は，彼らの環境識別子に対する小さな拡張により，shift0/reset0 に対する型システムが構築できることを示す．

## 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．そのような言語のことをコード生成言語と呼ぶ．

### 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[2] 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[2] 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．我々の言語体系において，コードを扱う reset0 は $\underline{\textbf{reset0}}\ M$ というように表し，これは，継続の範囲を $M$ に限定するという意味である．コードを扱う shift0 は $\underline{\textbf{shift0}}\ k \rightarrow M$ というように表し，これは，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である．つまり，shift0 と reset0 は対応関係にあり，reset0 で切り取った継続を，shift0 に

よって，$k$ へと束縛し，その継続を使うことができるようになる．

shift/reset[2] 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 let 挿入が可能となる．

以下で，我々の言語体系における shift0/reset0 による多段階 let 挿入の例を掲載する．

$$
\begin{aligned}
e = \ & \textbf{reset0}\ \ \underline{\textbf{clet}}\ x_1 = \%3\ \underline{\textbf{in}} \\
& \textbf{reset0}\ \ \underline{\textbf{clet}}\ x_2 = \%5\ \underline{\textbf{in}} \\
& \textbf{shift0}\ k_2\ \rightarrow\ \textbf{shift0}\ k_1\ \rightarrow\ \underline{\textbf{clet}}\ y = t\ \underline{\textbf{in}} \\
& \underline{\textbf{throw}}\ k_1\ (\underline{\textbf{throw}}\ k_2\ (x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y))
\end{aligned}
$$

まず，**reset0** によって，切り取られた継続 $\underline{\textbf{clet}}\ x_2 = \%5\ \underline{\textbf{in}}$ が，**shift0** によって，$k_2$ へと捕獲され，次に，**reset0** によって，切り取られた継続 $\underline{\textbf{clet}}\ x_2 = \%3\ \underline{\textbf{in}}$ が，**shift0** によって，$k_1$ へと捕獲される．

わかりやすいところまで計算を進めると以下のようになり，

$$
\begin{aligned}
e \rightsquigarrow^* & \ \underline{\textbf{clet}}\ y = t\ \underline{\textbf{in}} \\
& \underline{\textbf{throw}}\ k_1\ (\underline{\textbf{throw}}\ k_2\ (x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y))
\end{aligned}
$$

$\underline{\textbf{clet}}\ y = t\ \underline{\textbf{in}}$ がトップに挿入されたことが分かる．**throw** は，切り取られた継続を引数に適用するための演算子である．つまり，

$$
\begin{aligned}
e \rightsquigarrow^* & \ \underline{\textbf{clet}}\ y = t\ \underline{\textbf{in}} \\
& \underline{\textbf{clet}}\ x_1 = \%3\ \underline{\textbf{in}} \\
& \underline{\textbf{clet}}\ x_2 = \%5\ \underline{\textbf{in}} \\
& (x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y)
\end{aligned}
$$

となり，$\underline{\textbf{clet}}\ y = t\ \underline{\textbf{in}}$ が二重の **clet** を飛び越えて，挿入された事が分かる．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，抽象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．

# 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 [3] 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．須藤ら [1] 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 [4, 5]．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須藤らの研究 [1] を元に行い，深く入れ子になった内側からの，let 挿入等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，コード生成の言語に shift0/reset0 を組み合わせた言語を設計し，その言語によって書かれたプログラムの安全性は，型システムで安全なコードには型がつくように，安全でないコードには型がつかないように，型システムを構築する．環境識別子 Environment Classifier による型による変数のスコープのアイデア [1, 6] を拡張することによって，shift0/reset0 の型安全性を保証する．

# 6　本研究の型システム

## 6.1　型付け例

$$
\begin{aligned}
e = \ & \underline{\textbf{reset0}}\ \ \underline{\textbf{clet}}\ x_1 = \%3\ \underline{\textbf{in}} \\
& \underline{\textbf{reset0}}\ \ \underline{\textbf{clet}}\ x_2 = \%5\ \underline{\textbf{in}} \\
& \underline{\textbf{shift0}}\ k_2\ \rightarrow\ \underline{\textbf{shift0}}\ k_1\ \rightarrow\ \underline{\textbf{clet}}\ y = t\ \underline{\textbf{in}} \\
& \underline{\textbf{throw}}\ k_1\ (\underline{\textbf{throw}}\ k_2\ (x_1 \pm x_2 \pm y))
\end{aligned}
$$

$e$ の計算を進めると以下のようになる．

$$
\begin{aligned}
e \rightsquigarrow^* \ & \underline{\textbf{clet}}\ y = t\ \underline{\textbf{in}} \\
& \underline{\textbf{reset0}}\ \ \underline{\textbf{clet}}\ x_1 = \%3\ \underline{\textbf{in}} \\
& \underline{\textbf{reset0}}\ \ \underline{\textbf{clet}}\ x_2 = \%5\ \underline{\textbf{in}} \\
& (x_1 \pm x_2 \pm y)
\end{aligned}
$$

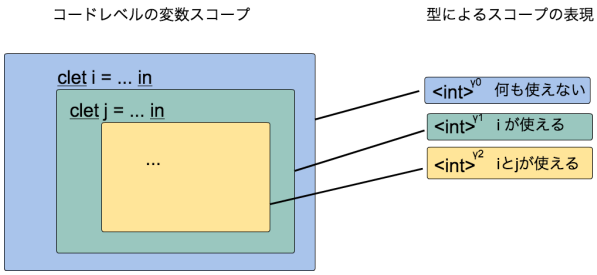
$t = \%7$ のとき $e_2$ は型が付く

$t = x_2$ か $t = x_1$ のとき $e_2$ は型が付かない

$t$ によって，安全なコードか安全でないコードかが変わるので，それを型で判断したい．このような型システムを構築することを考える．

## 6.2　環境識別子 EC によるスコープ表現 [1]

コードレベルの変数スコープと，型によるスコープの表現を表したのが下の図である．




$\gamma$ は変数のスコープを表し，そのスコープ内で使える自由変数の集合と思ってもらえば良い．$\gamma$ には，包含関係があり，それを $\gamma_1 \geq \gamma_0$ というような順序で表す．直感的には $\gamma_0$ より $\gamma_1$ のほうが使える自由変数が多いという意味である．

このように EC を型に付加することで，その型がどのスコープに存在するかどうかが分かる．

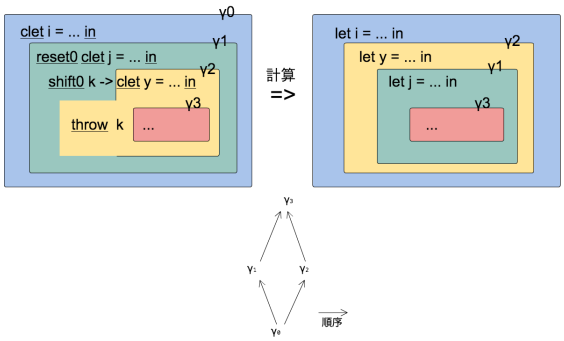
## 6.3　EC の洗練化

今までは，プログラムがネストしていけば，その分だけ，使える自由変数が増えていたのだが，shift0/reset0 が導入されたことにより，計算の順序が変わることがあり，単純にネストした分だけ使える自由変数が増えていくという訳にはいかない．

下の図は，

$$
\begin{aligned}
& \underline{\textbf{clet}}\ i = \ldots\ \underline{\textbf{in}} \\
& \underline{\textbf{reset0}}\ \ \underline{\textbf{clet}}\ j = \ldots\ \underline{\textbf{in}} \\
& \textbf{shift0}\ k\ \rightarrow\ \underline{\textbf{clet}}\ y = t\ \underline{\textbf{in}} \\
& \textbf{throw}\ k \ldots
\end{aligned}
$$

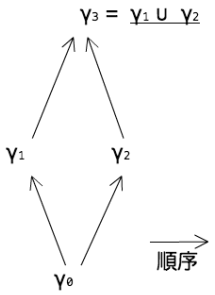
のスコープに色を付け，計算結果とともに載せた図である．また，下のグラフは EC の順序を視覚的に表している．




計算を進めると，reset0 によって，切り取られた継続が shift0 によって $k$ へ捕獲され， throw によって，使用されることにより，計算の順序が変わり，計算後では $\gamma_1$ と $\gamma_2$ のスコープの包含関係が入れ替わっている事がわかる．

### 6.4 EC のジョイン

今までの，EC では，計算の順序が変わることが無かったので，継続を任意の場所に貼り付けることを安全に行うことができなかった．しかし，EC のジョインというものを考えることで，shift0/reset0 を用いた let 挿入のコード生成の型安全性の保証を行うことができる．




上図は下のことを視覚的に表したグラフである．

- $\gamma_1$ 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

$\cup$ という概念を追加することで，shift0/reset0 を用いた let 挿入のコード生成の型安全性の保証を行うことができる．

## 7 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．コード生成言語の型システム [1] に shift0 reset0 を組み込んだ型システムの設計を行い，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，設計した型システムの健全性の証明 (subject reduction 等) を行い，および型検査器等の実装を行う．

## 参考文献


[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pp. 26–37, New York, NY, USA, 2003. ACM.