# 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

コード生成法は，プログラムの実行性能の高さと保守性・再利用性を両立できるプログラミング手法として有力なものである．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．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 Sudo らの研究等があるが，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した．

## 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静的に保証する言語体系および型システムを設計する。これにより、コード生成器のコンパイル段階、すなわち、実際にコードが生成されてコンパイルされるより遥かに前の段階でのエラーの検出が可能となるという利点がある。

コード生成における let 挿入は，生成されたコードを移動して効率良いコードに変形するための機能であり、ループ不変式を for ループの外側に移動したり、コードの計算結果を共有するなどのコード変換 (コード最適化) において必要な機能である。多段階 let 挿入は、入れ子になった for ループ等を飛び越えて、コードを移動する機能である。

本研究は，多段階 let 挿入を可能とするコード生成体系の構築のため、比較的最近になって理論的性質が解明された 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に対する型規則を適切に設計することにより、型安全性を厳密に保証し、上記の問題を解決した。

本研究に関連した従来研究としては、束縛子を越えない範囲でのコントロールオペレータを許した研究や、局所的な代入可能変数を持つ体系に対する須藤らの研究 [3]、後者を、グローバルな代入可能変数を持つ体系に拡張した研究 [1] などがある。しかし、いずれの研究でも 多段階の for ループを飛び越えた let 挿入は許していない。本研究は，須藤らの研究をベースに、shift0/reset0 を持つコード生成体系を設計した点に新規性がある。

## 2　コード生成と let 挿入

コード生成、すなわち、プログラムによるプログラム (コード) の生成の手法は、対象領域に関する知識、実行環境、利用可能な計算機リソースなどのパラメータに特化した (実行性能の高い) プログラムを生成する目的で広く利用されている。生成されるコードを文字列として表現する素朴なコード生成法では、

Type-Safe Code Generation with Multi-level Let-insertion
Junpei Ohishi, Yukiyoshi Kameyama, 筑波大学システム情報工学研究科コンピュータ・サイエンス専攻, Department of Computer Science, University of Tsukuba.

構文エラーなどのエラーを含むコードを生成してしまう危険があり、さらに、生成されたコードのディバッグが非常に困難であるという問題がある。

これらの問題を解決するため、コード生成器 (コード生成をするプログラム) を記述するためのプログラム言語の研究が行われており、特に静的な型システムのサポートを持つ言語として、MetaOCaml, Template Haskell, Scala LMS などがある。

本研究は、MetaOCaml などの値呼び関数型言語に基づいたコード生成言語を対象としているが、言語のプレゼンテーションでは、先行研究にならいコードコンビネータ (Code Combinator) 方式を使う。MetaML/MetaOCaml などにおける擬似引用 (Quasi-quotation) 方式は、コード生成に関する言語要素として「ブラケット (コード生成,quotation)」と「エスケープ (コード合成,anti-quotation)」を用いるのに対して、コードコンビネータ方式では、各演算子に対して、「コード生成版の演算子 (コードコンビネータ)」を用意してコード生成器を記述する。たとえば、加算 $e_1 + e_2$ に対して、コードコンビネータ版は $e_1 \underline{+} e_2$ というように、演算子名に下線をつけてあらわす。

本章では、例に基づいてコード生成器と let 挿入について説明する。対象言語の構文・意味論などの形式的体系の説明は後に行う.

### 2.1 コードコンビネータ方式のプログラム例

まず、(完成した) コードは、<3> や <3 + 5> のようにブラケットで囲んで表す。次のコードは、これらを生成するプログラムである。

$$(\underline{\mathbf{int}}\ 3) \rightsquigarrow^* \text{<3>}$$
$$(\underline{\mathbf{int}}\ 3)\ \underline{+}\ (\underline{\mathbf{int}}\ 5) \rightsquigarrow^* \text{<3} + 5\text{>}$$

$\underline{\mathbf{int}}$ は整数を整数のコードに変換し、$\underline{+}$ は、整数のコード 2 つをもらって、それらの加算をおこなうコードを生成するコードコンビネータである。なお、$\rightsquigarrow^*$ は 0 ステップ以上の簡約を表す。

$\underline{\lambda} x.e$ と $\cap$ はそれぞれラムダ抽象と関数適用のコードを生成する。

$$\underline{\lambda} x.x\underline{+}(\underline{\mathbf{int}}\ 3) \rightsquigarrow^* \text{<}\lambda x.x + 3\text{>}$$
$$(\underline{\lambda} x.x\underline{+}(\underline{\mathbf{int}}\ 3))\ \underline{@}\ (\underline{\mathbf{int}}\ 5) \rightsquigarrow^* \text{<}(\lambda u.u + 3)\ 5\text{>}$$

ラムダ抽象のコードコンビネータにおいて、$x$ は「(コードレベルの) 変数」そのものを表すのではなく、「変数のコード」をあらわす。上記の例の計算過程で、$x$ は <$u$>(ここで $u$ は新たに作成されたコードレベルの変数) に簡約され、計算が進む。

$\underline{\mathbf{let}}$ は let 式のコードを生成する。

$$\underline{\mathbf{let}}\ x = (\underline{\mathbf{int}}\ 3)\ \underline{\mathbf{in}}\ x\ \underline{+}\ (\underline{\mathbf{int}}\ 7)$$
$$\rightsquigarrow^* \textbf{<let } x = 3 \textbf{ in } x + 7\textbf{>}$$

実は、$\underline{\mathbf{let}}$ は、コードコンビネータとしてのラムダ抽象と適用によりマクロ定義され、上記の式は、以下の式と同じである。

$$(\underline{\lambda} x.\ x\ \underline{+}\ (\underline{\mathbf{int}}\ 7)\ )\ \underline{@}\ (\underline{\mathbf{int}}\ 3)$$
$$\rightsquigarrow^* \textbf{<let } x = 3 \textbf{ in } x + 7\textbf{>}$$

本研究の対象言語は、MetaML や MetaOCaml と同様、静的束縛の言語であり、以下の例では、束縛変数の名前が正しく付け換えられる。

$$\underline{\lambda} y.\textbf{let } x = y \textbf{ in } \underline{\lambda} y.\ x\ \underline{+}\ y \rightsquigarrow^* \text{<}\lambda y.\lambda y'.y\ +\ y'\text{>}$$

この例では、2 つのラムダ抽象が $y$ という変数をもっているが、これらは異なる束縛変数であるので、計算の過程で衝突が起きるときは名前換えが発生する。

### 2.2 コード生成における let 挿入

$\underline{\mathbf{for}}$ は for 式を生成するコードコンビネータである。ここで、(コードレベルの) 配列 $A$ の第 $n$ 要素に対する代入を $A[n] \leftarrow e$ と表し、$\underline{\mathbf{aryset}}\ a\ e_1\ e_2$ は対応するコードコンビネータであると仮定する。また、$A$ は適宜 $n$ 次元のものを考えることにする。

$$\underline{\mathbf{for}}\ x = (\underline{\mathbf{int}}\ 3)\ \underline{\mathbf{to}}\ (\underline{\mathbf{int}}\ 7)\ \underline{\mathbf{do}}$$
$$\underline{\mathbf{aryset}}\ \text{<}A\text{>}\ x\ (\underline{\mathbf{int}}\ 0)$$
$$\rightsquigarrow^* \textbf{<for } i = 3 \textbf{ to } 7 \textbf{ do } A[i] \leftarrow 0\textbf{>}$$

**for** を入れ子にすると、入れ子の for 式が生成できる。

$$\begin{array}{c}\underline{\textbf{for}}\ x = (\underline{\textbf{int}}\ 3)\ \underline{\textbf{to}}\ (\underline{\textbf{int}}\ 7)\ \underline{\textbf{do}}\\ \underline{\textbf{for}}\ y = (\underline{\textbf{int}}\ 1)\ \underline{\textbf{to}}\ (\underline{\textbf{int}}\ 9)\ \underline{\textbf{do}}\\ \underline{\textbf{aryset}}\ \texttt{<}A\texttt{>}\ (x,y)\ (\underline{\textbf{int}}\ 0)\\ \rightsquigarrow^{*}\ \textbf{<for}\ i = 3\ \textbf{to}\ 7\ \textbf{do}\\ \textbf{for}\ j = 1\ \textbf{to}\ 9\ \textbf{do}\\ A[i,j] \leftarrow 0\textbf{>}\end{array}$$

この二重ループの中で、複雑な計算をするループ不変式があったとする。たとえば、配列の初期値として 0 でなく、(何らかの複雑な) 計算結果を代入するが、その計算にはループ変数 $i, j$ を使わない場合を考える。それを $e$ とすると、

$$\begin{array}{c}\textbf{<for}\ i = 3\ \textbf{to}\ 7\ \textbf{do}\\ \textbf{for}\ j = 1\ \textbf{to}\ 9\ \textbf{do}\\ A[i,j] \leftarrow e\textbf{>}\end{array}$$

というコードの代わりに

$$\begin{array}{c}\textbf{<let}\ z = e\ \textbf{in}\\ \textbf{for}\ i = 3\ \textbf{to}\ 7\ \textbf{do}\\ \textbf{for}\ j = 1\ \textbf{to}\ 9\ \textbf{do}\\ A[i,j] \leftarrow z\textbf{>}\end{array}$$

というコードの方が実行性能が高くなることが期待できる。

このように、生成するコードの上部 (トップレベルに近い方) に let 式を挿入することができれば、早い段階で値を計算できたり、また、同一の部分式がある場合は計算結果を再利用できたり、という利点がある †1。

そこで、コード生成器に let 挿入の機能を組み込もう。let 挿入は部分計算の分野等で研究されており、CPS 変換あるいはコントロールオペレータを用いることで実現できることが知られている。本研究では、shift0/reset0 というコントロールオペレータを用いて let 挿入を実現する。

上記のコード生成器にコントロールオペレータを組みこんだものが次のプログラムである。

$$\begin{array}{l}\textbf{reset0}\ \ (\underline{\textbf{for}}\ x = (\underline{\textbf{int}}\ 3)\ \underline{\textbf{to}}\ (\underline{\textbf{int}}\ 7)\ \underline{\textbf{do}}\\ \qquad\quad (\underline{\textbf{for}}\ y = (\underline{\textbf{int}}\ 1)\ \underline{\textbf{to}}\ (\underline{\textbf{int}}\ 9)\ \underline{\textbf{do}}\\ \qquad\quad \textbf{shift0}\ k_1\ \rightarrow\ \underline{\textbf{let}}\ z = e\ \underline{\textbf{in}}\\ \qquad\quad \textbf{throw}\ k_1\ (\underline{\textbf{aryset}}\ \texttt{<}a\texttt{>}\ (x,y)\ z)))\end{array}$$

赤字の reset0,shift0,throw がコントロールオペレータであり、それらに対するインフォーマルな †2 計算規則は以下の通りである。

$$\begin{array}{l}\textbf{reset0}\ v \rightarrow v\\ \textbf{reset0}\ (E[\textbf{shift0}\ \ k\ \rightarrow\ \cdots(\textbf{throw}\ k\ e)\cdots])\\ \rightarrow \cdots(\textbf{reset0}(E[e]))\cdots\end{array}$$

ここで $v$ は値、$E$ は評価文脈である。2 行目では、reset0 と shift0 に挟まれた文脈が切り取られ、変数 $k$ に束縛され、**throw** $k$ $e$ の形の式の場所で利用される。ここで切り取られる文脈には、トップにあった **reset0** も含まれているため、簡約後のトップから **reset0** が消えている。よく知られている shift/reset では、この **reset0** が残る点が違っている。

上記のコード生成器をこの計算規則により計算すると、2 重の for 式に相当する文脈 $\underline{\textbf{for}}\ x = \cdots\ \underline{\textbf{to}}\ \cdots\ \underline{\textbf{do}}\ \underline{\textbf{for}}\ y = \cdots\ \underline{\textbf{to}}\ \cdots\ \underline{\textbf{do}}\ [\ ]$ が切り取られ **throw** の部分の $k_1$ で使われる。結果として、$\underline{\textbf{let}}\ z = e\ \underline{\textbf{in}}$ の部分が、この文脈の外側に移動する効果が得られ、let 挿入が実現できる。

上記の例では、一番外側まで let 挿入を行ったが、式 $e$ が $x$ に相当するループ変数を含むときは、一番外側まで持っていくことはできず、2 つの for 式の中間地点まで移動することになる。このためには、reste0 の設置場所を変更すればよい。

問題は、このように let 挿入をしたい式が複数ある場合である。「let 挿入をする先」に reset0 を 1 つ置くため、いくつかの let 挿入においては直近の reset0 まで移動するのではなく、2 つ以上先の (遠くの)reset0 まで let を移動したいことがある。これは、shift0/reset0 を入れ子にすることにより、以下のよ

†1 この変形・最適化は、コードを生成してから行なうのでよければ技術的に難しいものではない。しかし、コード生成においては、生成されるコード量の爆発が問題になることが多く、無駄なコードはできるだけ早い段階で除去したい、すなわち、コードを生成してから最適化するのではなく生成段階でコードを変形・最適化したいという強い要求がある。

†2 精密な意味論は後述する。

うに実現できる。

**reset0** (**for** $x = (\mathbf{int}\ 3)$ **to** (**int** 7) **do**
**reset0** (**for** $y = (\mathbf{int}\ 1)$ **to** (**int** 9) **do**
**shift0** $k_2$ → **shift0** $k_1$ → **let** $z = e$ **in**
**throw** $k_1$ (**throw** $k_2$ (**aryset** <$a$> $(x, y)$ $z$))))

青字のコントロールオペレータをいれた場合、let 挿入の「目的地」であるトップの位置 (赤字の reset0 で指定された位置) は、2 つ先の reset0 になってしまったが、これは、shift0 と throw をそれぞれ 2 回入れ子にすることにより実現できる。これが多段階 let 挿入である.

なお、このように直近の reset0 を越えた地点までの移動 (あるいは文脈の切り取り) は、shift/reset では実現できず、その拡張である階層的 shift/reset や shift0/reset0 が必要となる。本研究では、簡潔さのため、shift0/reset0 を用いることとした。

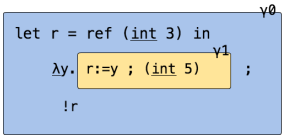
さて、以上のように shift0/reset0 を使うことにより多段階 let 挿入が実現できることがわかったが、自由な使用を許せば、危ないコード生成器を書けてしまう。上記の例では、項 $e$ がどのループ変数に依存するかによって、let をどこまで移動してよいかが変わってきた。例えば，トップレベルまで移動するコード生成器の場合、$e$ が <7> のときは型がつき，$x$ や $y$ のとき型が付かないようにしたい．このような精密な区別を実現する型システムを構築するのが本研究の目的である。

## 3 環境識別子による型システムの構築

### 3.1 先行研究のアイディア

Taha ら [2] は、純粋な (副作用のない) コード生成の言語の型安全性を保証するため、環境識別子 (Environment Classifier) を導入した。環境識別子 $\alpha$ は、コード生成のステージに対応し、「そのステージで使える (コードレベル) の変数とその型の集合 (あるいは型文脈)」を抽象的に表現した変数であり、$\langle \mathtt{int} \rangle^{\alpha}$ のように、コード型の一部として使用される。

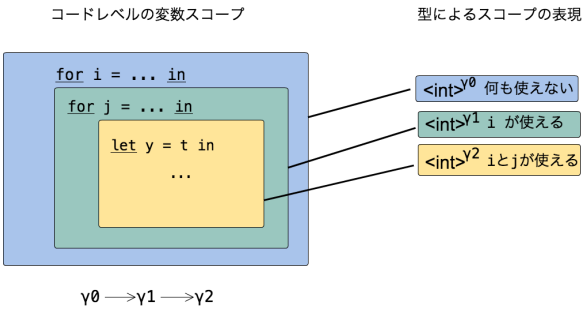
須藤ら [3] は、破壊的変数を持つコード生成言語に対する型安全性を保証するため、環境識別子を精密化した。本節では、須藤らのアイディアを解説する。以下の図は、彼らの言語における危険なプログラム例である。




須藤らの「危険な」例ここで、$r$ は、整数のコードを格納する参照 (破壊的セル) であり、$r := y$ と $!r$ はそれぞれ、$r$ への代入と $r$ の中身の読み出しを表す。上記のプログラムは、コードレベルのラムダ抽象 ($y$ に対するラムダ抽象) で生成されるコードレベル変数を $u$ とするとき、<$u$> を $r$ に格納し、$u$ のスコープ (黄色で示したもの) が終わったあとに取り出しているため、計算結果は、自由変数をもつコード <$u$> となり、危険である。

上記のようなプログラムを型エラーとするため、須藤らは、コードレベル変数のスコープごとに環境識別子を割り当てた。上記では外側のスコープ (青) が $\gamma_0$、内側のスコープ (黄) が $\gamma_1$ という環境識別子で表現される。$\gamma_0$ で有効なコードレベル変数はなく、$\gamma_1$ で有効なコードレベル変数は $y$(に対応して生成される変数) である。$\gamma_0$ のスコープは $\gamma_1$ のスコープを含む。言い換えれば、$\gamma_1$ で使える変数の方が $\gamma_0$ で使える変数の方が (同じか) 多い。このことを $\gamma_1 \geq \gamma_0$ と表すことにする。$r$ は $\langle \mathtt{int} \rangle^{\gamma_0}$ 型を持つ。$y$ は $\gamma_1$ で使える変数であるが、$\gamma_0$ では使えないため、$r$ に $y$ を代入することはできず、$r := y$ のところで型エラーとなる。

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あらわしたのが，以下の図である．for や let などコードレベルの束縛子があるたびに、新しいスコープが開かれ、使える変数が増えていくことが分かるだろう.



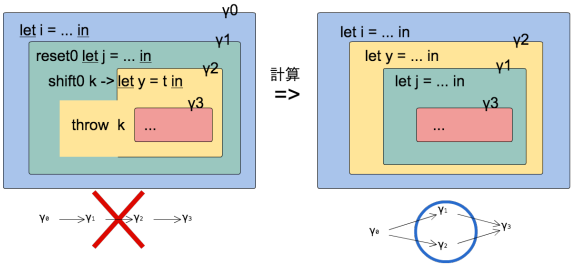

コードレベルの変数の型に、(精密化した) 環境識

別子を付与することで、その変数が使えるスコープがわかり、破壊的代入などの副作用があるプログラムにおいてもスコープや型の安全性を保つことができる。

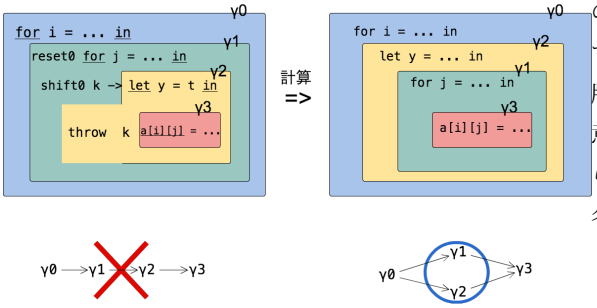
なお、須藤らの対象としていた言語が持っていた計算エフェクトは、「局所的なスコープをもつ参照」であり、現実の OCaml/MetaOCaml 等とは異なるものであった。同一の著者グループは、最近、精密化した環境識別子のアイディアを用いて、グローバルな参照を持つ言語に対するある種の型安全性が成立することを示している [1]。

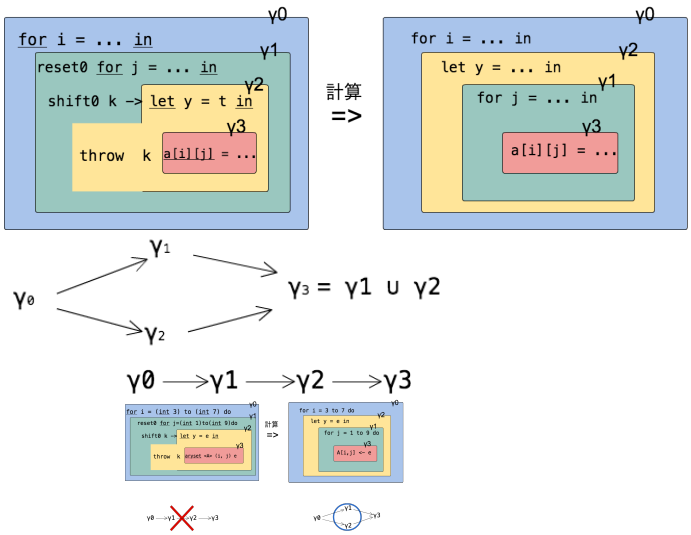
### 3.2 本研究: 環境識別子の拡張

本研究で扱う shift0/reset0 によるコントロールエフェクトは、須藤らによる精密化された環境識別子でも扱うことができない。本節では、その問題点を明らかにし、その問題の解決の鍵となる join (∪) の導入について述べる。




しかし，shift0/reset0 が導入されることにより，計算の順序が




## 4 本研究のアイディア

## 5 対象言語: 構文と意味論

本研究における対象言語は、ラムダ計算にコード生成機能とコントロールオペレータ shift0/reset0 を追加したものに型システムを導入したものである。

本稿では、最小限の言語のみについて考えるため、コード生成機能の「ステージ (段階)」は、コード生成段階 (レベル 0、現在ステージ) と生成されたコードの実行段階 (レベル 1、将来ステージ) の 2 ステージのみを考える。

言語のプレゼンテーションにあたり、先行研究にならいコードコンビネータ (Code Combinator) 方式を使う。MetaML/MetaOCaml などにおける擬似引用 (Quasi-quotation) 方式が「ブラケット (コード生成,quotation)」と「エスケープ (コード合成,anti-quotation)」を用いるのに対して、コードコンビネータ方式では、「ブラケット (コード生成,quotation)」のみを用い、そのかわりに、各種の演算子を 2 セットずつ用意する。たとえば、加算は $e_1 + e_2$ という通常版のほか、$e_1 \underline{+} e_2$ というコードコンビネータ版も用意する。後者は <3>$\underline{+}$<5> を計算すると <3 + 5> が得られる。このように、コードコンビネータは、演算子名に下線をつけてあらわす。

### 5.1 構文の定義

対象言語の構文を定義する。

変数は、レベル 0 変数 ($x$), レベル 1 変数 ($u$), (レベル 0 の) 継続変数 ($k$) の 3 種類がある。レベル 0 項 ($e^0$)、レベル 1 項 ($e^1$) およびレベル 0 の値 ($v$) を下の通り定義する。

$$
\begin{aligned}
c &::= i \mid b \mid \underline{\mathbf{int}} \mid \ \underline{@}\ \mid + \mid \underline{+} \mid \underline{\mathbf{if}} \\
v &::= x \mid c \mid \lambda x.e^0 \mid \texttt{<}e^1\texttt{>} \\
e^0 &::= v \mid e^0\ e^0 \mid \mathbf{if}\ e^0\ \mathbf{then}\ e^0\ \mathbf{else}\ e^0 \\
&\quad \mid \underline{\lambda} x.e^0 \mid \underline{\underline{\lambda}} u.e^0 \\
&\quad \mid \mathbf{reset0}\ e^0 \mid \mathbf{shift0}\ k \to e^0 \mid \mathbf{throw}\ k\ v \\
e^1 &::= u \mid c \mid \lambda u.e^1 \mid e^1\ e^1 \mid \mathbf{if}\ e^1\ \mathbf{then}\ e^1\ \mathbf{else}\ e^1
\end{aligned}
$$

ここで $i$ は整数の定数、$b$ は真理値定数である。

定数のうち、下線がついているものはコードコンビネータである。変数は、ラムダ抽象 (下線なし、下線つき、二重下線つき) および shift0 により束縛され、$\alpha$ 同値な項は同一視する。$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ および $\underline{\mathbf{let}}\ x = e_1\ \underline{\mathbf{in}}\ e_2$ は、それぞれ、$(\lambda x.e_2)e_1$ $(\underline{\lambda} x.e_2)\ \underline{@}\ e_1$ の省略形である。

### 5.2 操作的意味論

対象言語は、値呼びで left-to-right の操作的意味論を持つ。ここでは評価文脈に基づく定義を与える。

評価文脈を以下のように定義する。

$$
\begin{aligned}
E &::= [\ ] \mid E\ e^0 \mid v\ E \\
&\quad \mid \mathbf{if}\ E\ \mathbf{then}\ e^0\ \mathbf{else}\ e^0 \mid \mathbf{reset0}\ E \mid \underline{\underline{\lambda}} u.E
\end{aligned}
$$

コード生成言語で特徴的なことは、コードレベルのラムダ抽象の内部で評価が進行する点である。実際、上記の定義には、$\underline{\underline{\lambda}} u.E$ が含まれている。たとえば、$\underline{\underline{\lambda}} u.u\underline{+}[\ ]$ は評価文脈である。

この評価文脈 $E$ と次に述べる計算規則 $r \to l$ により、評価関係 $e \rightsquigarrow e'$ を次のように定義する。

$$\frac{r \to l}{E[r] \rightsquigarrow E[l]}$$

計算規則は以下の通り定義する。

$$
\begin{aligned}
(\lambda x.e)\ v &\to e\{x := v\} \\
\mathbf{if}\ \mathit{true}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 &\to e_1 \\
\mathbf{if}\ \mathit{else}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 &\to e_2 \\
\underline{\lambda} x.e &\to \underline{\underline{\lambda}} u.(e\{x := \texttt{<}u\texttt{>}\}) \\
\underline{\underline{\lambda}} y.\texttt{<}e\texttt{>} &\to \texttt{<}\lambda y.e\texttt{>} \\
\mathbf{reset0}\ v &\to v \\
\mathbf{reset0}(E[\mathbf{shift0}\ k \to e]) &\to e\{k \Leftarrow E\}
\end{aligned}
$$

ただし、4 行目の $u$ はフレッシュなコードレベル変数とし、最後の行の $E$ は穴の周りに **reset0** を含まない評価文脈とする。また、この行の右辺のトップレベルに **reset0** がない点が、shift/reset の振舞いとの違いである。すなわち、shift0 を 1 回計算すると、reset0 が 1 つはずれるため、shift0 を N 個入れ子にすることにより、N 個分外側の reset0 までアクセスすることができ、多段階 let 挿入を実現できるようになる。

上記における継続変数に対する代入 $e\{k \Leftarrow E\}$ は次の通り定義する。

$$
\begin{aligned}
(\mathbf{throw}\ k\ v)\{k \Leftarrow E\} &\equiv \mathbf{reset0}(E[v]) \\
(\mathbf{throw}\ k'\ v)\{k \Leftarrow E\} &\equiv \mathbf{throw}\ k'\ (v\{k \Leftarrow E\}) \\
&\qquad \text{ただし } k \neq k'
\end{aligned}
$$

上記以外の $e$ に対する定義は透過的である。

上記の定義の 1 行目で **reset0** を挿入しているのは **shift0** の意味論に対応しており、これを挿入しない場合は別のコントロールオペレータ (Felleisen の control/prompt に類似した control0/prompt0) の振舞いとなる。

コードコンビネータ定数の振舞い (ラムダ計算における $\delta$ 規則に相当) は以下のように定義する。

$$
\begin{aligned}
\underline{\mathbf{int}}\ n &\to \texttt{<}n\texttt{>} \\
\texttt{<}e_1\texttt{>}\ \underline{@}\ \texttt{<}e_2\texttt{>} &\to \texttt{<}e_1\ e_2\texttt{>} \\
\texttt{<}e_1\texttt{>}\ \underline{+}\ \texttt{<}e_2\texttt{>} &\to \texttt{<}e_1 + e_2\texttt{>} \\
\underline{\mathbf{if}}\ \texttt{<}e_1\texttt{>}\ \texttt{<}e_2\texttt{>}\ \texttt{<}e_3\texttt{>} &\to \texttt{<}\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\texttt{>}
\end{aligned}
$$

計算の例を以下に示す。

$$
\begin{aligned}
e_1 = \mathbf{reset0}\ \ & \underline{\mathbf{let}}\ x_1 = \%3\ \underline{\mathbf{in}} \\
& \mathbf{reset0}\ \ \underline{\mathbf{let}}\ x_2 = \%5\ \underline{\mathbf{in}} \\
& \mathbf{shift0}\ k\ \rightarrow\ \underline{\mathbf{let}}\ y = t\ \underline{\mathbf{in}} \\
& \mathbf{throw}\ k\ (x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y)
\end{aligned}
$$

$$
\begin{aligned}
&[e_1] \rightsquigarrow [\mathbf{reset0}(\underline{\mathbf{let}}\ x_1 = \%3\ \underline{\mathbf{in}} \\
&\quad \mathbf{reset0}\ \underline{\mathbf{let}}\ x_2 = \%5\ \underline{\mathbf{in}} \\
&\quad [\mathbf{shift0}\ k\ \rightarrow\ \underline{\mathbf{let}}\ y = t\ \underline{\mathbf{in}} \\
&\quad [\mathbf{throw}\ k\ (x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y)]])] \\
&\quad \rightsquigarrow [\underline{\mathbf{let}}\ y = t\ \underline{\mathbf{in}} \\
&\quad [\underline{\lambda}x.\mathbf{reset0}\ (\underline{\mathbf{let}}\ x_1 = \%3\ \underline{\mathbf{in}}\ \mathbf{reset0}\ (\underline{\mathbf{let}}\ x_2 = \%5\ \underline{\mathbf{in}}[x])(x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y)]] \\
&\quad \rightsquigarrow [\underline{\lambda}y.(\underline{\lambda}x.\mathbf{reset0}\ (\underline{\mathbf{let}}\ x_1 = \%3\ \underline{\mathbf{in}}\ \mathbf{reset0}\ (\underline{\mathbf{let}}\ x_2 = \%5\ \underline{\mathbf{in}}[x])(x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y)] \\
&\quad \rightsquigarrow [[\underline{\lambda}y.(\underline{\lambda}x.\mathbf{reset0}\ (\underline{\mathbf{let}}\ x_1 = \%3\ \underline{\mathbf{in}}\ \mathbf{reset0}\ (\underline{\mathbf{let}}\ x_2 = \%5\ \underline{\mathbf{in}}[x])(x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y)] \\
&\quad \rightsquigarrow [[\underline{\lambda}y_1.(\underline{\lambda}x.\mathbf{reset0}\ (\underline{\mathbf{let}}\ x_1 = \%3\ \underline{\mathbf{in}}\ \mathbf{reset0}\ (\underline{\mathbf{let}}\ x_2 = \%5\ \underline{\mathbf{in}}[x])(x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y)] \\
&\quad \rightsquigarrow
\end{aligned}
$$

let ref の e1 e2 の制限 scope extrusion 問題への対処 shift reset で同じようなことをかけるので，これについて考える

## 6 型システム

本研究での型システムについて述べる。

まず、基本型 $b$、環境識別子 (Environment Classifier)$\gamma$ を以下の通り定義する。

$$b ::= \mathtt{int} \mid \mathtt{bool}$$

$$\gamma ::= \gamma_x \mid \gamma \cup \gamma$$

$\gamma$ の定義における $\gamma_x$ は環境識別子の変数を表す。すなわち、環境識別子は、変数であるかそれらを $\cup$ で結合した形である。以下では、メタ変数と変数を区別せず $\gamma_x$ を $\gamma$ と表記する。また、$L ::= | \gamma$ は現在ステージと将来ステージをまとめて表す記号である。たとえば、$\Gamma \vdash^L e : t\ ;\ \sigma$ は、$L =$ のとき現在ステージの判断で、$L = \gamma$ のとき将来ステージの判断となる。

レベル 0 の型 $t^0$、レベル 1 の型 $t^1$、(レベル 0 の) 型の有限列 $\sigma$, (レベル 0 の) 継続の型 $\kappa$ を次の通り定義する。

$$
\begin{aligned}
t^0 &::= b \mid t^0 \xrightarrow{\sigma} t^0 \mid \langle t^1 \rangle^\gamma \\
t^1 &::= b \mid t^1 \rightarrow t^1 \\
\sigma &::= \epsilon \mid \sigma, t^0 \\
\kappa^0 &::= \langle t^1 \rangle^\gamma \overset{\sigma}{\Rightarrow} \langle t^1 \rangle^\gamma
\end{aligned}
$$

レベル 0 の関数型 $t^0 \xrightarrow{\sigma} t^0$ は、エフェクトをあらわす列 $\sigma$ を伴っている。これは、その関数型をもつ項を引数に適用したときに生じる計算エフェクトであり、具体的には、**shift0** の answer type の列である。前述したように shift0 は多段階の reset0 にアクセスできるため、$n$ 個先の reset0 の answer type まで記憶するため、このように型の列 $\sigma$ で表現している。

本稿の範囲ではないが、コントロールオペレータは現在ステージにあらわれ生成されるコードの中にはあらわれないレベル 1 の関数型は、エフェクトを表す列を持たない。また、本項では、shift0/reset0 はコードを操作する目的にのみ使うため、継続の型は、コードからコードへの関数の形をしている。ここでは、後の定義を簡略化するため、継続を、通常の関数とは区別しており、そのため、継続の型も通常の関数の型とは区別して二重の横線で表現している。

型判断は、以下の 2 つの形である。

$$\Gamma \vdash^L e : t;\ \sigma$$

$$\Gamma \models \gamma \geq \gamma$$

ここで、型文脈 $\Gamma$ は次のように定義される。

$$\Gamma ::= \emptyset \mid \Gamma, (\gamma \geq \gamma) \mid \Gamma, (x : t) \mid \Gamma, (u : t)^\gamma$$

型判断の導出規則を与える。まず、$\Gamma \models \gamma \geq \gamma$ の形に対する規則である。

$$\frac{}{\Gamma \models \gamma_1 \geq \gamma_1} \qquad \frac{}{\Gamma, \gamma_1 \geq \gamma_2 \models \gamma_1 \geq \gamma_2}$$

$$\frac{\Gamma \models \gamma_1 \geq \gamma_2 \quad \Gamma \models \gamma_2 \geq \gamma_3}{\Gamma \models \gamma_1 \geq \gamma_3}$$

$$\frac{}{\Gamma \models \gamma_1 \cup \gamma_2 \geq \gamma_1} \qquad \frac{}{\Gamma \models \gamma_1 \cup \gamma_2 \geq \gamma_2}$$

$$\frac{\Gamma \models \gamma_3 \geq \gamma_1 \quad \Gamma \models \gamma_3 \geq \gamma_2}{\Gamma \models \gamma_3 \geq \gamma_1 \cup \gamma_2}$$

次に、$\Gamma \vdash^L e : t;\ \sigma$ の形に対する導出規則を与える。まずは、レベル 0 における単純な規則である。

$$\frac{}{\Gamma, x:t \vdash x:t\ ;\ \sigma} \quad \frac{}{\Gamma, (u:t)^{\gamma} \vdash^{\gamma} u:t\ ;\ \sigma}$$

$$\frac{}{\Gamma \vdash^{L} c:t^{c}\ ;\ \sigma}$$

$$\frac{\Gamma \vdash^{L} e_1 : t_2 \to t_1; \sigma \quad \Gamma \vdash^{L} e_2 : t_2; \sigma}{\Gamma \vdash^{L} e_1\ e_2 : t_1; \sigma}$$

$$\frac{\Gamma,\ x:t_1 \vdash e:t_2\ ;\ \sigma}{\Gamma \vdash \lambda x.e : t_1 \xrightarrow{\sigma} t_2\ ;\ \sigma'} \quad \frac{\Gamma,\ (u:t_1)^{\gamma} \vdash^{\gamma} e:t_2\ ;\ \sigma}{\Gamma \vdash^{\gamma} \lambda x.e : t_1 \to t_2\ ;\ \sigma'}$$

$$\frac{\Gamma \vdash^{L} e_1 : \texttt{bool};\ \sigma \quad \Gamma \vdash^{L} e_2 : t; \sigma \quad \Gamma \vdash^{L} e_3 : t; \sigma}{\Gamma \vdash^{L} \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : t\ ;\ \sigma}$$

次にコードレベル変数に関するラムダ抽象の規則である。

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

$$\frac{\Gamma, \gamma_1 \geq \gamma, x:(u:t_1)^{\gamma_1} \vdash e:\langle t_2\rangle^{\gamma_1}; \sigma}{\Gamma \vdash \underline{\underline{\lambda}} u^{1}.e : \langle t_1 \to t_2\rangle^{\gamma}; \sigma}$$

コントロールオペレータに対する型導出規則である。

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

$$\frac{\Gamma,\ k:\langle t_1\rangle^{\gamma_1} \overset{\sigma}{\Rightarrow} \langle t_0\rangle^{\gamma_0} \vdash e:\langle t_0\rangle^{\gamma_0}; \sigma \quad \Gamma \models \gamma_1 \geq \gamma_0}{\Gamma \vdash \textbf{shift0}\ k \to e:\langle t_1\rangle^{\gamma_1}\ ;\ \langle t_0\rangle^{\gamma_0}, \sigma}$$

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

コード生成に関する補助的な規則として、Subsumption に相当する規則等がある。

$$\frac{\Gamma \vdash e:\langle t\rangle^{\gamma_1}; \sigma \quad \Gamma \models \gamma_2 \geq \gamma_1}{\Gamma \vdash e:\langle t\rangle^{\gamma_2}; \sigma}$$

$$\frac{\Gamma \vdash^{\gamma_1} e:t\ ;\ \sigma \quad \Gamma \models \gamma_2 \geq \gamma_1}{\Gamma \vdash^{\gamma_2} e:t\ ;\ \sigma}$$

$$\frac{\Gamma \vdash^{\gamma} e:t^{1}; \sigma}{\Gamma \vdash \texttt{<}e\texttt{>} : \langle t^{1}\rangle^{\gamma}; \sigma}$$

## 7 型付け例

$$\begin{aligned} e_1 = \textbf{reset0}\ \ &\underline{\textbf{let}}\ x_1 = \%3\ \underline{\textbf{in}} \\ &\textbf{reset0}\ \ \underline{\textbf{let}}\ x_2 = \%5\ \underline{\textbf{in}} \\ &\textbf{shift0}\ k\ \to\ \underline{\textbf{let}}\ y = t\ \underline{\textbf{in}} \\ &\textbf{throw}\ k\ (x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y) \end{aligned}$$

If $t = \%7$ or $t = x_1$, then $e_1$ is typable.
If $t = x_2$, then $e_1$ is not typable.

$$\begin{aligned} e_2 = \textbf{reset0}\ \ &\underline{\textbf{let}}\ x_1 = \%3\ \underline{\textbf{in}} \\ &\textbf{reset0}\ \ \underline{\textbf{let}}\ x_2 = \%5\ \underline{\textbf{in}} \\ &\textbf{shift0}\ k_2\ \to\ \textbf{shift0}\ k_1\ \to\ \underline{\textbf{let}}\ y = t\ \underline{\textbf{in}} \\ &\textbf{throw}\ k_1\ (\textbf{throw}\ k_2\ (x_1 \mathbin{\underline{+}} x_2 \mathbin{\underline{+}} y)) \end{aligned}$$

If $t = \%7$, then $e_1$ is typable.
If $t = x_2$ or $t = x_1$, then $e_1$ is not typable.

## 8 型安全性の証明

本研究の型システムに対する型保存 (Subject Reduction) 定理とその証明のポイントを示す。

**定理 8.1 (型保存定理)**
$\vdash e:t\ ;\ \sigma$ かつ $e \rightsquigarrow e'$ であれば，$\vdash e':t\ ;\ \sigma$ である。

通常の型保存定理では、仮定が $\Gamma \vdash e:t\ ;\ \sigma$ となり、項 $e$ が自由変数を持つことを許すのであるが、証明に関する技術的理由により、本稿では、閉じた項のみに対する型保存定理を示す。

**補題 8.1 (不要な仮定の除去)**
$\Gamma_1, \gamma_2 \geq \gamma_1 \vdash e:t_1\ ;\ \sigma$ かつ、$\gamma_2$ が $\Gamma_1, e, t_1, \sigma$ に出現しないなら、$\Gamma_1 \vdash e:t_1\ ;\ \sigma$ である。

**補題 8.2 (値に関する性質)**
$\Gamma_1 \vdash v:t_1\ ;\ \sigma$ ならば，$\Gamma_1 \vdash v:t_1\ ;\ \sigma'$ である。

**補題 8.3 (代入)**
$\Gamma_1, \Gamma_2, x:t_1 \vdash e:t_2\ ;\ \sigma$ かつ $\Gamma_1 \vdash v:t_1\ ;\ \sigma$ ならば，$\Gamma_1, \Gamma_2 \vdash e\{x := v\}:t_2\ ;\ \sigma$

次の補題は、もうちょっとチェックしないといけない

**補題 8.4 (識別子に関する多相性)**
穴の周りに *reset0* を含まない評価文脈 $E$、変数

$x$、そして $\Gamma = (u_1 : t_1)^{\gamma_1}, \cdots, (u_n : t_n)^{\gamma_n}$ かつ $i = 1, \cdots, n$ に対して $\gamma_0 \geq \gamma_i$ であるとする。このとき、$\Gamma, x : \langle t_0 \rangle^{\gamma'} \vdash E[x] : \langle t_1 \rangle^{\gamma_0}$ ; $\sigma$ であれば、$\Gamma \models \gamma \geq \gamma_0$ となる任意の $\gamma$ に対して、$\Gamma, x : \langle t_0 \rangle^{\gamma' \cup \gamma} \vdash E[x] : \langle t_1 \rangle^{\gamma_0}$ ; $\sigma$ である。

## 9 まとめと今後の課題

**謝辞** 本研究は、JSPS 科研費 15K12007 の支援を受けている。

## 参 考 文 献

[1] Kiselyov, O., Kameyama, Y., and Sudo, Y.: Refined Environment Classifiers: Type- and Scope-safe Code Generation with Mutable Cells, *Proceedings of the 14th Asian Symposium on Programming Languages and Systems*, APLAS 2016, 2016.

[2] Taha, W. and Nielsen, M. F.: Environment Classifiers, *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, New York, NY, USA, ACM, 2003, pp. 26–37.

[3] 須藤悠斗, Kiselyov, O., 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.