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．示したとするかどうか？

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?[]、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?[] 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　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

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.

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 ?[] 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 ?[] 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，reset0 は **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．shift0 は **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 ?[] 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．言語の定義，意味論については後ほど紹介する．ここでは，多段階 let 挿入がどのように行われるか雰囲気を掴んでもらいたい．ここを後のフォーマルな，言語・意味論の定義にならって修正する

$$
\begin{aligned}
e = \ & \mathbf{reset0}\ \ \underline{\mathbf{let}}\ x_1 = \underline{\mathbf{int}}3\ \underline{\mathbf{in}} \\
& \quad \mathbf{reset0}\ \ \underline{\mathbf{let}}\ x_2 = \underline{\mathbf{int}}5\ \underline{\mathbf{in}} \\
& \quad \mathbf{shift0}\ k_2\ \rightarrow\ \mathbf{shift0}\ k_1\ \rightarrow\ \underline{\mathbf{let}}\ y = t\ \underline{\mathbf{in}} \\
& \quad \mathbf{throw}\ k_1\ (\mathbf{throw}\ k_2\ (x_1 \underline{+} x_2 \underline{+} y))
\end{aligned}
$$

まず，**reset0** によって，切り取られた継続 $\underline{\mathbf{let}}\ x_2 = \underline{\mathbf{int}}5\ \underline{\mathbf{in}}$ 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 $\underline{\mathbf{let}}\ x_2 = \underline{\mathbf{int}}3\ \underline{\mathbf{in}}$ 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，

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

$\underline{\mathbf{let}}\ y = t\ \underline{\mathbf{in}}$ 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，

$$
\begin{aligned}
e \rightsquigarrow^* \ & \underline{\mathbf{let}}\ y = t\ \underline{\mathbf{in}} \\
& \underline{\mathbf{let}}\ x_1 = \underline{\mathbf{int}}3\ \underline{\mathbf{in}} \\
& \underline{\mathbf{let}}\ x_2 = \underline{\mathbf{int}}5\ \underline{\mathbf{in}} \\
& (x_1 \underline{+} x_2 \underline{+} y)
\end{aligned}
$$

となり，$\underline{\mathbf{let}}\ y = t\ \underline{\mathbf{in}}$ が 二重の $\underline{\mathbf{let}}$ 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 ?[] 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 ?[] は，書き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 ?[] ?[]．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 ?[] 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?[]?[]を拡張することによって，shift0/reset0 の型安全性を保証する．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ったのだが， 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．そこで，環境識別子の結合を意味する $\cup$ を導入することにより上記の問題を解決することにした．

6 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．

6.1 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．

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

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，α 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．

6.2 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．

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

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．

$$(\textbf{throw}\ k\ v)\{k \Leftarrow E\} \equiv \textbf{reset0}(E[v])$$
$$(\textbf{throw}\ k'\ v)\{k \Leftarrow E\} \equiv \textbf{throw}\ k'\ (v\{k \Leftarrow E\})$$
$$\text{ただし}\ k \neq k'$$

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 δ 規則に相当) は以下のように定義する．

$$\underline{\textbf{int}}\ n \rightarrow \texttt{<}n\texttt{>}$$
$$\texttt{<}e_1\texttt{>}\ \underline{@}\ \texttt{<}e_2\texttt{>} \rightarrow \texttt{<}e_1\ e_2\texttt{>}$$
$$\texttt{<}e_1\texttt{>}\ \underline{+}\ \texttt{<}e_2\texttt{>} \rightarrow \texttt{<}e_1 + e_2\texttt{>}$$
$$\underline{\textbf{if}}\ \texttt{<}e_1\texttt{>}\ \texttt{<}e_2\texttt{>}\ \texttt{<}e_3\texttt{>} \rightarrow \texttt{<}\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\texttt{>}$$

計算の例を以下に示す．

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

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

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

7 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)γ を以下の通り定義する．

$$b ::= \texttt{int} \mid \texttt{bool}$$
$$\gamma ::= \gamma_x \mid \gamma \cup \gamma$$

γ の定義における γ_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 γ_x を γ と表記する．また，$L ::= \mid \gamma$ 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 σ, (レベル 0 の) 継続の型 κ を次の通り定義する．

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

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 σ 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 σ で表現している．

本稿の範囲では，コントロールオペレータは現在ステージのみにあらわれ，生成されるコードの中にはあらわないため，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 Γ 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．

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

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

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

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

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．

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

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

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

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

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

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．

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

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

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

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．

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

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

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

8 型付け例

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

If $t = \underline{\mathbf{int}}7$ or $t = x_1$, then e_1 is typable.

If $t = x_2$, then e_1 is not typable.

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

If $t = \underline{\mathbf{int}}7$, then e_1 is typable.

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

9 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 9.1 (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 9.1 (不要な仮定の除去)
$\Gamma_1, \gamma_2 \geq \gamma_1 \vdash e : t_1 \,;\, \sigma$ かつ、γ_2 が Γ_1, e, t_1, σ に出現しないなら、$\Gamma_1 \vdash e : t_1 \,;\, \sigma$ である。

補題 9.2 (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 9.3 (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 9.4 (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 γ 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。

10 関連研究と結論

謝辞 本研究は、JSPS 科研費 15K12007 の支援を受けている。

11 まとめと今後の課題

途中までになったところを今後やりたいこととする？そうすると，研究結果としてはまとまっていないので，タイトルを “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の試み” に変更するかも

謝辞ほげほげ