筑波大学大学院博士課程

システム情報工学研究科修士論文

安全なコード移動が可能なコード生成言語の型システムの設計と実装

大石 純平

修士（工学）

(コンピュータサイエンス専攻)

指導教員　亀山 幸義

2017年 3月

概要

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ための型システムを構築した．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須藤らの研究等があるが，本研究は，彼らの環境識別子にジョインを追加するという拡張により，shift0/reset0 を持つコード生成言語に対する型システムが構築できることを示した．

目 次

図 目 次

第1章　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．

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[1] 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，型安全性を解決することを目的とする．コントロールオペレータを含む項の計算について分析した結果，スコープの包含関係が逆転することや 2 つのスコープの合流があることから，変数スコープを表す識別子にジョイン (和集合) を追加すればよいという着想を得て，型システムを設計することに成功した．

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 [2]，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 [3] 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および型システムを設計し，静的に安全性を検査できるようにした点に新規性がある．

第2章　背景

2.1　コード生成言語

本節ではコード生成言語について簡潔に説明する．

コード生成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．そのようなコード生成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，実行などを行う機構を備えることが望ましい．そのような言語をコード生成言語という．

本節では，コード生成言語の例として MetaOCaml を用いて，コード生成について紹介する

Bracket `.<e>.` : コード生成

`.<e>.` をコードと呼び，コード化とは， 項 e を Bracket `.< >.` で囲むことにより，e の評価を遅らせることである．Bracket `.< >.` を用いることで，動的にコードを生成する事ができる．

以下で Bracket を用いた MetaOCaml の例を掲載する．

```
# let x = .<1 + 2>.;;
val x : int code = .<1 + 2>.
```

`#` ではじまる行はユーザーの入力であり，そのすぐ下の行は MetaOCaml からの応答である．Bracket を用いることで， 項 `1+2` は，`3` とは評価されず そのまま項 `1+2` を表すコード `.<1 + 2>.` が結果として返されている．`val x : int code = .<1 + 2>.` とは，値 `x` は，`int code` 型であり，その値は `.<1 + 2>.` という意味である．

Escape `.~e` : コードの埋め込み

`.~e` に対する呼び名は特にないが，Escape `.~`はコード`.<e>.` のブラケットを外す演算子である．これにより，コードにコード`.<e>.` の e を埋め込むことができる．

以下で Escape を用いた MetaOCaml の例を掲載する．

```
# let y = .< .~x * .~x >.;;
val y : int code = .<(1 + 2) * (1 + 2)>.
```

ここで, x は .<1 + 2>. に束縛されているとする. その環境で, let y = .< .~x * .~x >. を評価すると, .~x に x の Bracket が外された 1+2 が埋め込まれる. これにより, コード内の計算を特定の部分すすめることができ, コード内にコードを埋め込むことができる.

2.1.1 コード生成の例

次の power 関数は x の n 乗を計算する関数である.

```
# let rec power n x =
    if n = 0 then x
    else x * power (n-1) x;;
val power : int -> int -> int = <fun>

# let power5 = fun x -> power 5 x;;
val power5 : int -> int = <fun>
```

power5 は x の 5 乗を計算する関数である. 例えば, 7 に power5 を適用すると, 7 の 5 乗が計算される. power5 に 7 が渡された時点で, 計算が進み, power 関数が評価され, 5 の 7 乗が求まる.

次にコード生成を用いる例を見る.

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

# let gen_power5 = .<fun x -> .~(gen_power 5 .<x>.)>.;;
val gen_power5 : (int -> int) code =
  .<fun x_1  -> x_1 * (x_1 * (x_1 * (x_1 * (x_1 * 1))))>.
```

gen_power は, x の n 乗を計算する関数のコード生成を行う関数である. この関数は, n に特化したコードを生成する関数である. 特化というのは, コード生成前に n の値による計算をあらかじめしておくということである. それによって, gen_power5 は x の計算を遅らせることができ, gen_power 関数が評価されることによって .<fun x_1 -> x_1 * (x_1 * (x_1 * (x_1 * (x_1 * 1))))>.

というように展開される．このように，コード生成法は事前に計算するところと，計算を遅らせておくところを決めることができるので，効率の良いプログラムが作成できる．

2.2 shift0/reset0

本節では，限定継続を扱うためのコントロールオペレータ shift0/reset0 について説明する．

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．継続は，計算のある時点における残りの計算のことである．限定継続とは，残りの計算すべてではなく，ある時点の計算から，ある時点の計算までのことである．

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[4] 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[4] 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．reset0 は **reset0** (M) というように表し，切り取られる継続を M に限定するという意味となる．shift0 は **shift0** $k \ \rightarrow M$ 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．

$$
\begin{aligned}
&\textbf{reset0}\ (3 + \textbf{shift0}\ k \rightarrow \textbf{let}\ x = 5\ \textbf{in}\ k\ x) \\
&\rightsquigarrow^{*} \textbf{let}\ x = 5\ \textbf{in}\ k\ x \qquad \text{where } k = \textbf{reset0}\ (3 + []) \\
&\rightsquigarrow^{*} \textbf{let}\ x = 5\ \textbf{in reset0}\ (3 + x) \\
&\rightsquigarrow^{*} \textbf{reset0}\ (3 + 5) \\
&\rightsquigarrow^{*} 8
\end{aligned}
$$

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 を x に適用することで，$3+x$ 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．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 コード生成と let 挿入

前節で見たようなコード生成言語と shift0/reset0 を用いて多段階 let 挿入と呼ばれる技法を可能にする．本節では，コード生成において let 挿入可能にするための事柄について述べる．

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，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．

2.3.1 コードコンビネータ方式のプログラム例

ここからは，コードは .< >. でなく <> と表す．

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．次の例は，これらを生成するプログラムである．

$$\underline{\mathbf{int}}\ 3 \rightsquigarrow^* \texttt{<3>}$$

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

$\underline{\mathbf{int}}$ 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 $\underline{@}$ はそれぞれラムダ抽象と関数適用のコードを生成する．

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

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．

let は let 式のコードを生成する．

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

実は，**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．

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

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．

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

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．

2.3.2 コード生成における let 挿入

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，**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．

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

for を入れ子にすると，入れ子の for 式が生成できる．

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

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，

$$
\begin{aligned}
&\textbf{<for}\ i = 3\ \textbf{to}\ 7\ \textbf{do}\\
&\quad \textbf{for}\ j = 1\ \textbf{to}\ 9\ \textbf{do}\\
&\qquad a[i, j] \leftarrow e\textbf{>}
\end{aligned}
$$

というコードの代わりに

$$
\begin{aligned}
&\textbf{<let}\ z = e\ \textbf{in}\\
&\qquad \textbf{for}\ i = 3\ \textbf{to}\ 7\ \textbf{do}\\
&\qquad \textbf{for}\ j = 1\ \textbf{to}\ 9\ \textbf{do}\\
&\qquad\quad a[i, j] \leftarrow z\textbf{>}
\end{aligned}
$$

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．

[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．

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．

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

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．

$$\begin{aligned}
&\textbf{reset0}\ v \rightsquigarrow^* v\\
&\textbf{reset0}\ (E[\textbf{shift0}\ \ k\ \ \rightarrow\ \ \cdots(\textbf{throw}\ k\ v)\cdots])\\
&\rightarrow \textbf{reset0}(\cdots(E[v])\cdots)
\end{aligned}$$

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** が残る点が異なっている．

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 **for** $x = \cdots$ **to** $\cdots$ **do** **for** $y = \cdots$ **to** $\cdots$ **do** [ ] が切り取られ **throw** の部分の k_1 で使われる．結果として，**let** $z = e$ **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，reset0 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，以下のように実現できる．

$$\begin{aligned}
&\textbf{reset0}\ \ (\underline{\textbf{for}}\ x = (\underline{\textbf{int}}\ 3)\ \underline{\textbf{to}}\ (\underline{\textbf{int}}\ 7)\ \underline{\textbf{do}}\\
&\textbf{reset0}\ \ (\underline{\textbf{for}}\ y = (\underline{\textbf{int}}\ 1)\ \underline{\textbf{to}}\ (\underline{\textbf{int}}\ 9)\ \underline{\textbf{do}}\\
&\quad \textbf{shift0}\ k_2\ \rightarrow\ \textbf{shift0}\ k_1\ \rightarrow\ \underline{\textbf{let}}\ z = e\ \underline{\textbf{in}}\\
&\qquad \textbf{throw}\ k_1\ (\textbf{throw}\ k_2\ (\underline{\textbf{aryset}}\ \texttt{<}a\texttt{>}\ (x, y)\ z))))
\end{aligned}$$

[2]精密な意味論は後述する．

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異なる．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．

2.4 Scope extrusion

この節では，Scope extrusion (変数が意図した束縛から抜け出してしまうこと) について例を用いて説明する．前述の例では，e が x や y のとき，Scope extrusion という問題が発生する．

$$
\begin{aligned}
&\textbf{reset0}\ \ (\underline{\textbf{for}}\ x = (\underline{\textbf{int}}\ 3)\ \underline{\textbf{to}}\ (\underline{\textbf{int}}\ 7)\ \underline{\textbf{do}}\\
&\textbf{reset0}\ \ (\underline{\textbf{for}}\ y = (\underline{\textbf{int}}\ 1)\ \underline{\textbf{to}}\ (\underline{\textbf{int}}\ 9)\ \underline{\textbf{do}}\\
&\quad \textbf{shift0}\ k_2\ \rightarrow\ \textbf{shift0}\ k_1\ \rightarrow\ \underline{\textbf{let}}\ z = x\ \underline{\textbf{in}}\\
&\qquad \textbf{throw}\ k_1\ (\textbf{throw}\ k_2\ (\underline{\textbf{aryset}}\ \langle a\rangle\ (x, y)\ z))))
\end{aligned}
$$

このプログラムは，$\underline{\textbf{let}}\ z = x\ \underline{\textbf{in}}$ がトップレベルまで移動するコード生成器である．x は $\underline{\textbf{for}}\ x = (\underline{\textbf{int}}\ 3)\ \underline{\textbf{to}}\ (\underline{\textbf{int}}\ 7)\ \underline{\textbf{do}}$ のところで束縛されているが，x を含んだ式がトップレベルまで移動してしまい，x は未束縛となり，実行時エラーとなる．このように，意図した束縛から変数が抜け出てしまうことを Scope extrusion という．

本研究の目的は，let 挿入などにより，動的にコード生成を行いつつ，Scope extrusion が起きないことを保証することである．

第3章　環境識別子による型システムの構築

プログラム実行時にエラーを起こさないようにしたい場合，プログラムを動的に動かしてエラーを発見するのではなく，静的にプログラムを解析し，意図したプログラムのみ実行できるようにする必要がある．それを実現するひとつの方法として型システムを構築する方法がある．以下の節では，コード生成体系に shift0/reset0 を導入した我々の体系に対して，安全なプログラムに対しては型が付き，それ以外の安全でないプログラムには型が付かないような型システム構築のアイデアを記す．

3.1　先行研究のアイディア

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 Kameyama らの研究 [5] 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．

Taha ら [6] 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 α 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，$\langle \texttt{int} \rangle^{\alpha}$ のように，コード型の一部として使用される．

須藤ら [2] 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．

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 γ_0，内側のスコープ (黄) が γ_1 という環境識別子で表現される．γ_0 で有効なコードレベル変数はなく，γ_1 で有効なコー

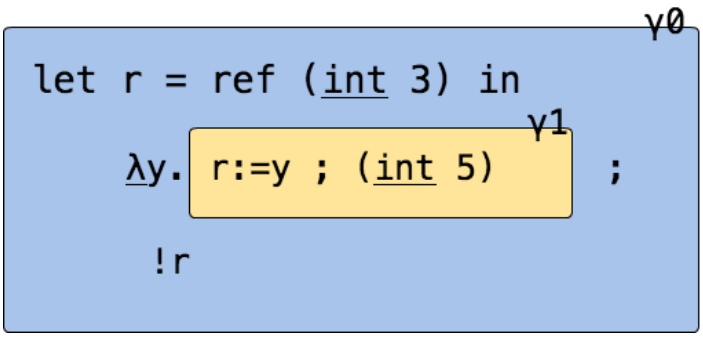

図 3.1: 危険な例

ドレベル変数は y(に対応して生成される変数) である．γ_0 のスコープは γ_1 のスコープを含む．言い換えれば，γ_1 で使える変数の方が γ_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 $\langle \texttt{int} \rangle^{\gamma_0}\texttt{ref}$ 型を持つ．y は γ_1 で使える変数であるが，γ_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．

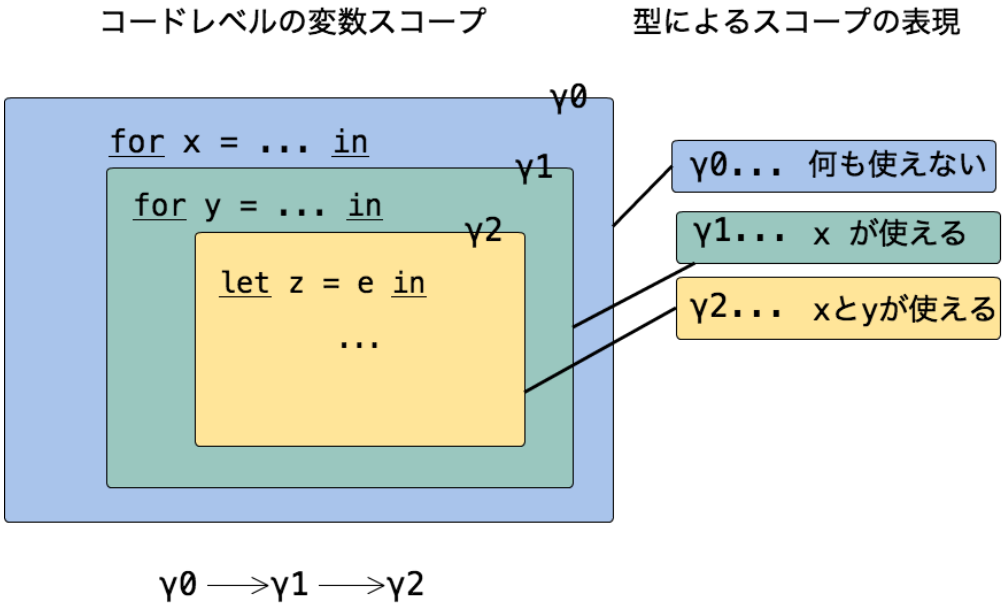

図 3.2: 型によるコードレベル変数のスコープ表現

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 [3]．

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．

このため，前述の 2 重の for ループ生成において，その中間に let 挿入をするプログラムについて考察する．その概形は 3.3 図の上図である．

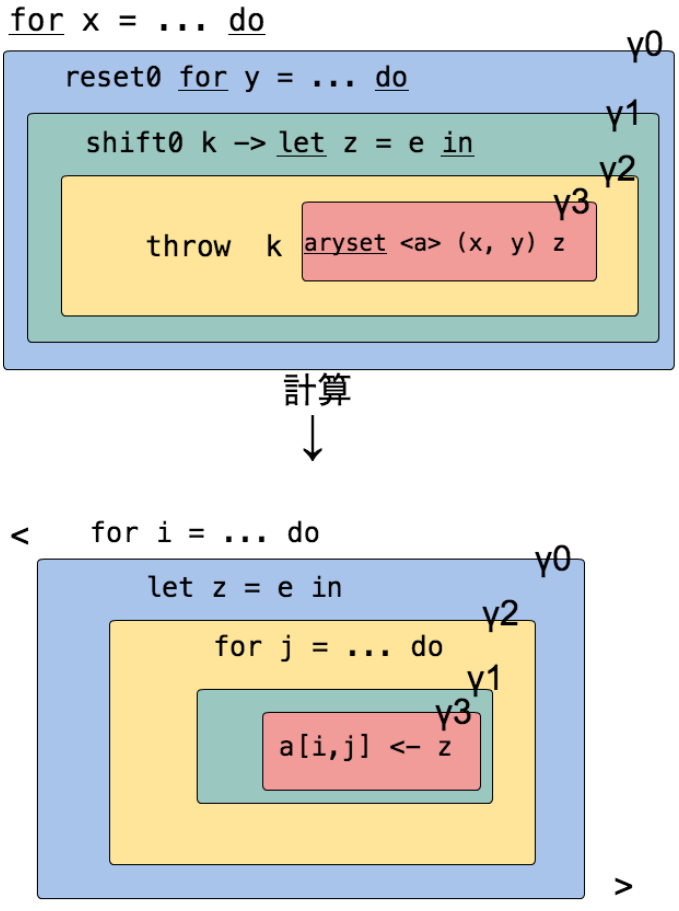

図 3.3: コード生成による let 挿入

3.3 図で，使えるコードレベル変数が異なる場所ごとに，$\gamma_0, \gamma_1, \gamma_2, \gamma_3$ と名付けた．須藤らの体系通りであれば，これらの環境識別子の間の順序は，スコープの包含関係通りであるので，

γ0 ⟶γ1 ⟶γ2 ⟶γ3

図 3.4: 誤った環境識別子の順序

3.4 図のような順序がつくはずである．しかし，計算を進めて得られたコード (3.3 図の下図) を見ると，γ_1(緑色) と γ_2(黄色) の位置関係が入れ代わっている．結果のコードの型が整合する (束縛変数が自由になることはない等) ためには，γ_2 において γ_1 で使える変数を使ってはいけないことがわかる．一方で，赤字で示された γ_3 においては，γ_1 の変数も γ_2 の変数も使って構わない．これらを考慮すると，環境識別子の間の順序は 3.5 図となる．

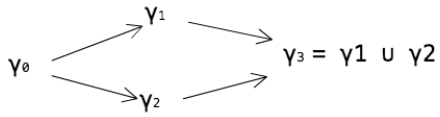

図 3.5: 正しい環境識別子の順序

ここでのポイントは，γ_0 から，2 つの異なる (包含関係のない)γ_1 と γ_2 に別れたあと，再び γ_3 で合流することである．須藤らの体系では，環境識別子のなす半順序集合全体は木の形であったが，本研究の体系では，このように 1 度別れたものが合流することがある．なお，γ_3 で使える変数の集合は γ_1 と γ_2 で使える変数の和集合と一致するので $\gamma_3 = \gamma_1 \cup \gamma_2$ とおくことができる．このように，環境識別子の世界に Join ($\cup$) を導入することにより，コントロールオペレータによる文脈の移動に対応できることになった．

3.3 本研究: 型システムの構築

本研究の基本アイディアは前項で述べたようなシンプルなものであるが，言語体系と型システムの構築にあたってはいくつかの困難があった．ここでは，そのうち，コントロールオペレータが切りとった継続に関する多相性について述べる．

前節の例では，shift0 が切り取る文脈 (継続) は，穴が γ_1 のスコープにあり (その型は $\langle t_1 \rangle^{\gamma_1}$ の型)，文脈全体が γ_0 のスコープのにある (その型は $\langle t_0 \rangle^{\gamma_0}$ の形) となっている．これを throw において使うときは，γ_3 スコープのものを γ_2 スコープに変換している．すなわち，k は $\langle t_1 \rangle^{\gamma_3}$ 型から $\langle t_0 \rangle^{\gamma_2}$ 型への関数であるように振る舞う．

この 2 つのギャップを埋めるのは，継続 (評価文脈) のある種の多相性である．このケース

では，

$$\langle t_1 \rangle^{\gamma_1} \to \langle t_0 \rangle^{\gamma_0}$$

という型で定義された継続変数 k が，任意の $\gamma_2 \geq \gamma_0$ に対して，

$$\langle t_1 \rangle^{\gamma_1 \cup \gamma_2} \to \langle t_0 \rangle^{\gamma_0 \cup \gamma_2}$$

という型を持つことが言えれば良いことがわかる．($\gamma_3 = \gamma_1 \cup \gamma_2$ あること，また，$\gamma_2 \geq \gamma_0$ ならば $\gamma_0 \cup \gamma_2 = \gamma_2$ であることに注意されたい．)

このような多相性 (サブタイプのもとでの多相性) は，継続の研究ではいくつか知られており，今回のケースでも成立すると考えられる[1]．

本研究では，将来的に型推論アルゴリズムを構築することを考慮して，環境識別子に関する一般的な多相性を導入することを避け，継続変数について特別な型を与え，その変数を throw で使うときに，多相性を利用できるようにするという方策をとった．これにより型システムを複雑化させることなく，コントロールオペレータの導入ができ，簡潔な型保存性の証明につながった．

最終的に得られた throw の型付け規則は以下のものであり，継続変数 k には，特別な関数型 $\Rightarrow$ を与えていること (通常の関数型は $\to$ で表現している)，また，γ_1 から γ_0 への変換子として得られた k を，$\gamma_1 \cup \gamma_2$ から $\gamma_2 = \gamma_0 \cup \gamma_2$ への変換子として利用していることがわかる．

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

なお，継続が作用する型 (上記の $\langle t_1 \rangle^{\gamma_1}$ など) は，本研究ではコード型に限定した．その理由は，須藤らの体系の方式で，参照 (ref) が関数型を持つことを許すとコードレベル変数が束縛域を脱出してしまい，本研究のコントロールオペレータを持つ体系でも同様の事態が生じると予想されたからである．なお，コード型のみを扱うことのできるコントロールオペレータであっても，多段階 let 挿入の表現のためには十分である．

[1] この型付けが問題ないことは直感的には明らかであるが，意味論の上で，このような型付け規則が「正しい」ことの証明は，将来課題である．

第4章　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．

前述したように，本研究の言語では，コードコンビネータ (Code Combinator) 方式を使い，コードコンビネータは，$\underline{+}$ や $\underline{\mathbf{if}}$ のように下線を引いて表す．

4.1　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．

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

図 4.1: 対象言語の構文の定義

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．

let $x = e_1$ **in** e_2 および $\underline{\textbf{let}}$ $x = e_1$ $\underline{\textbf{in}}$ e_2 は，それぞれ，$(\lambda x.e_2)e_1$ および $(\underline{\lambda} x.e_2)$ $\underline{@}$ e_1 の省略形である．前述の例でのべた $\underline{\textbf{for}}$ は，コード構築定数とコードレベル適用を用いて導入することとし，(この導入にあたっての型システムの拡張は容易なので) ここでは省略する．

4.2 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．

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

図 4.2: 評価文脈

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，$\underline{\lambda} u.E$ が含まれている．たとえば，$\underline{\lambda} u.\langle u\rangle \underline{+} [\,]$ 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 $e \rightsquigarrow e'$ を図 4.3 のように定義する．

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

図 4.3: $e \rightsquigarrow e'$ の評価関係

計算規則は図 4.4 の通り定義する．

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

図 4.4: 計算規則

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 $e\{k \Leftarrow E\}$ は図 4.5 の通り定義する．

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

図 4.5: 継続への代入

上記以外の e に対する代入の定義は透過的であるとする．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) は図 4.6 のように定義する．

$$\underline{\textbf{int}}\ n \to \texttt{<}n\texttt{>}$$

$$\texttt{<}e_1\texttt{>}\ \underline{@}\ \texttt{<}e_2\texttt{>} \to \texttt{<}e_1\ e_2\texttt{>}$$

$$\texttt{<}e_1\texttt{>}\ \underline{+}\ \texttt{<}e_2\texttt{>} \to \texttt{<}e_1 + e_2\texttt{>}$$

$$\underline{\textbf{if}}\ \texttt{<}e_1\texttt{>}\ \texttt{<}e_2\texttt{>}\ \texttt{<}e_3\texttt{>} \to \texttt{<}\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\texttt{>}$$

図 4.6: コードコンビネータの規則

第5章　型システム

本研究での型システムについて述べる．
基本型 b，環境識別子 (Environment Classifier)γ を以下の通り定義する．

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

図 5.1: 基本型，環境識別子の定義

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．ここで環境識別子として $\cup$ を導入した理由は後述する．
$L ::= \cdot \mid \gamma$ は現在ステージ (レベル 0) と将来ステージ (レベル 1) をまとめて表す記号である．たとえば，$\Gamma \vdash^L e : t \; ; \; \sigma$ は，$L = \cdot$ のとき現在ステージ (レベル 0) の判断で，$L = \gamma$ のとき将来ステージ (レベル 1) の判断となる．なお，現在ステージ (レベル 0) を表す $\cdot$ は省略する事がある．
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 を図 5.2 の通り定義する．

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

図 5.2: (レベル 0 の) 継続の型の定義

σ の ϵ は空の列を表す．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．ただし，本研究の範囲では，answer type modification に対応する必要はないので，エフェクトはシンプルに型の列 (n 個先の reset0 の answer type を $n = 1, \cdots, k$ に対して並べた列) で表現している．この型システムの詳細は，Materzok ら [1] の研究を参照されたい．

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 (レベル 0) にのみあらわれ，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．

$$\Gamma \vdash^{L} e : t;\ \sigma$$
$$\Gamma \models \gamma \geq \gamma$$

図 5.3: 型判断の定義

ここで，型文脈 Γ は次のように定義される．

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

図 5.4: 型文脈の定義

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．

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

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

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

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

図 5.5: $\Gamma \models \gamma \geq \gamma$ の形に対する型導出規則

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

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

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

$$\frac{\Gamma \vdash^{\gamma} e_1 : t_2 \xrightarrow{\sigma} t_1; \ \sigma \quad \Gamma \vdash^{\gamma} e_2 : t_2; \sigma}{\Gamma \vdash^{\gamma} e_1 \ e_2 : t_1; \sigma} \quad \frac{\Gamma \vdash e_1 : t_2 \to t_1; \ \sigma \quad \Gamma \vdash e_2 : t_2; \ \sigma}{\Gamma \vdash e_1 \, e_2 : t; \ \sigma}$$

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

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

図 5.6: (レベル 0，レベル 1 の)$\Gamma \vdash^{L} e : t; \ \sigma$ の単純な形に対する型導出規則

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．

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

図 5.7: コードレベルのラムダ抽象の型導出規則

コントロールオペレータに対する型導出規則である．

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

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

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

図 5.8: コントロールオペレータに対する型導出規則

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．

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

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

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

$$\frac{\Gamma \vdash e : t;\ \langle t' \rangle^{\gamma_1}, \sigma \quad \Gamma \models \gamma_2 \geq \gamma_1}{\Gamma \vdash e : t;\ \langle t' \rangle^{\gamma_2}, \sigma}$$

図 5.9: コード生成に関する Subsumption の型導出規則

shift0 の Answer type の列 (σ) に関する Subsumption に相当する規則がある．

$$\frac{\Gamma \vdash e : t;\ \langle t' \rangle^{\gamma_1}, \sigma \quad \Gamma \models \gamma_2 \geq \gamma_1}{\Gamma \vdash e : t;\ \langle t' \rangle^{\gamma_2}, \sigma}$$

図 5.10: σ-part(Anser type の列) に関する Subsumption の型導出規則

5.1 型付け例

上記の型システムのもとで，いくつかの項の型付けについて述べる．

5.1.1 let 挿入の例

$$\begin{aligned} e = \ & \mathbf{reset0}\ \ \underline{\mathbf{let}}\ x_1 = e_1\ \underline{\mathbf{in}} \\ & \mathbf{reset0}\ \ \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}} \\ & \mathbf{shift0}\ k\ \rightarrow\ \underline{\mathbf{let}}\ y = \Box\ \underline{\mathbf{in}} \\ & \mathbf{throw}\ k\ y \end{aligned}$$

式 e に対して，$\Box = \underline{\mathbf{int}}\ 7$ あるいは $\Box = x_1$ であれば，e は型付け可能である．一方，$\Box = x_2$ であれば，e は型付けできない．

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_3 \vdash y : \langle t \rangle^{\gamma_2 \cup \gamma_1 \cup \gamma_3};\ \cdot} \quad \dfrac{}{\Gamma_3 \models \gamma_1 \cup \gamma_3 \geq \gamma_1}}{\Gamma_3 = \Gamma_2,\ \gamma_3 \geq \gamma_1,\ y : \langle t \rangle^{\gamma_3} \vdash \mathbf{throw}\ k\ (\mathbf{reset0}\ y) : \langle t \rangle^{\gamma_3};\ \langle t \rangle^{\gamma_0}} \quad \dfrac{\vdots}{\Gamma_2 \vdash \Box : \langle t \rangle^{\gamma_1};\ \langle t \rangle^{\gamma_0}}\ (*)}{\Gamma_2 = \Gamma_1,\ k : \langle t \rangle^{\gamma_2} \overset{\langle t \rangle^{\gamma_0}}{\Rightarrow} \langle t \rangle^{\gamma_1} \vdash \underline{\mathbf{let}}\ y = \Box\ \underline{\mathbf{in}}\ \cdots : \langle t \rangle^{\gamma_1};\ \langle t \rangle^{\gamma_0}}}{\Gamma_1 = \gamma_2 \geq \gamma_1,\ x_2 : \langle t \rangle^{\gamma_2},\ \gamma_1 \geq \gamma_0,\ x_1 : \langle t \rangle^{\gamma_1} \vdash \mathbf{shift0}\ k \rightarrow \cdots : \langle t \rangle^{\gamma_2};\ \langle t \rangle^{\gamma_1}, \langle t \rangle^{\gamma_0}}}{\gamma_1 \geq \gamma_0,\ x_1 : \langle t \rangle^{\gamma_1} \vdash \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \cdots : \langle t \rangle^{\gamma_1};\ \langle t \rangle^{\gamma_1}, \langle t \rangle^{\gamma_0}}}{\gamma_1 \geq \gamma_0,\ x_1 : \langle t \rangle^{\gamma_1} \vdash \mathbf{reset0}\ \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \cdots : \langle t \rangle^{\gamma_1};\ \langle t \rangle^{\gamma_0}}}{\vdash \underline{\mathbf{let}}\ x_1 = e_1\ \underline{\mathbf{in}}\ \mathbf{reset0}\ \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \cdots : \langle t \rangle^{\gamma_0};\ \langle t \rangle^{\gamma_0}}}{\vdash e_1 : \langle t \rangle^{\gamma_0};\ \cdot}$$

ここで，$\Box$ が x_1, $\underline{\mathbf{int}}7$, x_2 の場合に $\Gamma_2 \vdash \Box : \langle t \rangle^{\gamma_1};\ \langle t \rangle^{\gamma_0}$ が成り立つかを見ていく．
$(*)$ のところに着目すると，$\Gamma_2 = \gamma_2 \geq \gamma_1,\ x_2 : \langle t \rangle^{\gamma_2},\ \gamma_1 \geq \gamma_0,\ x_1 : \langle t \rangle^{\gamma_1},\ k : \langle t \rangle^{\gamma_2} \overset{\langle t \rangle^{\gamma_0}}{\Rightarrow} \langle t \rangle^{\gamma_1}$ より，

$\square = x_1$ **の時**

$x_1 : \langle t\rangle^{\gamma_1} \vdash x_1 : \langle t\rangle^{\gamma_1}$ が成り立ち，型が付く

$\square = \underline{\mathbf{int}}7$ **の時**

$\underline{\mathbf{int}}7$ は定数であるので，どの Classifier γ_i においても型が付く．

$\square = x_2$ **の時**

$\gamma_2 \geq \gamma_1$，$\gamma_1 \geq \gamma_0$，$x_2 : \langle t\rangle^{\gamma_2}$ より，x_2 のスコープは γ_2 であり，γ_2 スコープのコード変数は，γ_1 スコープでは一般に使用できないので $\gamma_2 \geq \gamma_1$，$\gamma_1 \geq \gamma_0$，$x_2 : \langle t\rangle^{\gamma_2} \vdash x_2 : \langle t\rangle^{\gamma_1}$ は成り立たず，型がつかない

5.1.2　多段階 let 挿入の例

$$
\begin{aligned}
e' = {} & \mathbf{reset0}\ \ \underline{\mathbf{let}}\ x_1 = e_1\ \underline{\mathbf{in}} \\
& \mathbf{reset0}\ \ \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}} \\
& \mathbf{shift0}\ k_2\ \rightarrow\ \mathbf{shift0}\ k_1\ \rightarrow\ \underline{\mathbf{let}}\ y = \square\ \underline{\mathbf{in}} \\
& \mathbf{throw}\ k_1\ (\mathbf{reset0}\ (\mathbf{throw}\ k_2\ y))
\end{aligned}
$$

式 e' に対して，$\square = \underline{\mathbf{int}}7$ であれば e' は型付け可能である．一方，$\square = x_2$ あるいは $\square = x_1$ であれば，e' は型付けできない．

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_3 \vdash y : \langle t\rangle^{\gamma_2\cup\gamma_1\cup\gamma_3};\ \cdot \quad \overline{\Gamma_3 \models \gamma_1\cup\gamma_3 \geq \gamma_0}}{\Gamma_3 \vdash \mathbf{throw}\ k_2\ y : \langle t\rangle^{\gamma_1\cup\gamma_3};\ \langle t\rangle^{\gamma_0}} \quad \overline{\Gamma_3 \models \gamma_1\cup\gamma_3 \geq \gamma_0}}{\Gamma_3 \vdash \mathbf{throw}\ k_2\ y : \langle t\rangle^{\gamma_1\cup\gamma_3};\ \langle t\rangle^{\gamma_1\cup\gamma_3}}\ (\#)}{\Gamma_3 \vdash \mathbf{reset0}(\mathbf{throw}\ k_2\ y) : \langle t\rangle^{\gamma_1\cup\gamma_3};\ \cdot} \quad \overline{\Gamma_3 \models \gamma_3 \geq \gamma_0}}{\Gamma_3 = \Gamma_2,\ \gamma_3 \geq \gamma_0,\ y : \langle t\rangle^{\gamma_3} \vdash \mathbf{throw}\ k_1\ (\mathbf{reset0}(\mathbf{throw}\ k_2\ y)) : \langle t\rangle^{\gamma_3};\ \cdot} \quad \dfrac{\vdots}{\Gamma_2 \vdash \square : \langle t\rangle^{\gamma_0};\ \cdot}\ (*)}{\Gamma_2 = \Gamma_1,\ k_2 : \langle t\rangle^{\gamma_2} \overset{\langle t\rangle^{\gamma_0}}{\Rightarrow} \langle t\rangle^{\gamma_1},\ k_1 : \langle t\rangle^{\gamma_1} \dot{\Rightarrow} \langle t\rangle^{\gamma_0} \vdash \underline{\mathbf{let}}\ y = \square\ \underline{\mathbf{in}}\ \cdots : \langle t\rangle^{\gamma_0};\ \cdot}}{\Gamma_1,\ k_2 : \langle t\rangle^{\gamma_2} \overset{\langle t\rangle^{\gamma_0}}{\Rightarrow} \langle t\rangle^{\gamma_1} \vdash \mathbf{shift0}\ k_1 \rightarrow \cdots : \langle t\rangle^{\gamma_1};\ \langle t\rangle^{\gamma_0}}}{\Gamma_1 = \gamma_2 \geq \gamma_1,\ x_2 : \langle t\rangle^{\gamma_2},\ \gamma_1 \geq \gamma_0,\ x_1 : \langle t\rangle^{\gamma_1} \vdash \mathbf{shift0}\ k_2 \rightarrow \mathbf{shift0}\ k_1 \rightarrow \cdots : \langle t\rangle^{\gamma_2};\ \langle t\rangle^{\gamma_1}, \langle t\rangle^{\gamma_0}}}{\gamma_1 \geq \gamma_0,\ x_1 : \langle t\rangle^{\gamma_1} \vdash \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \cdots : \langle t\rangle^{\gamma_1};\ \langle t\rangle^{\gamma_1}, \langle t\rangle^{\gamma_0}}}{\gamma_1 \geq \gamma_0,\ x_1 : \langle t\rangle^{\gamma_1} \vdash \mathbf{reset0}\ \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \cdots : \langle t\rangle^{\gamma_1};\ \langle t\rangle^{\gamma_0}}}{\vdash \underline{\mathbf{let}}\ x_1 = e_1\ \underline{\mathbf{in}}\ \mathbf{reset0}\ \underline{\mathbf{let}}\ x_2 = e_2\ \underline{\mathbf{in}}\ \cdots : \langle t\rangle^{\gamma_0};\ \langle t\rangle^{\gamma_0}}}{\vdash e : \langle t\rangle^{\gamma_0};\ \cdot}
$$

この型付けで注意するところは，複数回の **throw** を使うときは その間に **reset0** を入れなければいけないところである．**reset0** を入れることで，σ-part のずれを防ぎ，**throw** 規

則を適用できる準備ができる．
(#) のところに着目すると，k_2 の型は $k_2 : \langle t \rangle^{\gamma_2} \overset{\langle t \rangle^{\gamma_0}}{\Rightarrow} \langle t \rangle^{\gamma_1}$ となっているので，**throw** 規則を適用するには，σ-part の subsumption 規則を適用して $\langle t \rangle^{\gamma_1 \cup \gamma_3}$ から $\langle t \rangle^{\gamma_0}$ が導ければ良い．

次に，$\square$ が $x_1, x_2, \underline{\mathbf{int}}7$ の場合に $\Gamma_2 \vdash \square : \langle t \rangle^{\gamma_0};\ \cdot$ が成り立つかを見ていく．
(∗) のところに着目すると，$\Gamma_2 = \gamma_2 \geq \gamma_1,\ x_2 : \langle t \rangle^{\gamma_2},\ \gamma_1 \geq \gamma_0,\ x_1 : \langle t \rangle^{\gamma_1}, k_2 : \langle t \rangle^{\gamma_2} \overset{\langle t \rangle^{\gamma_0}}{\Rightarrow} \langle t \rangle^{\gamma_1},\ k_1 : \langle t \rangle^{\gamma_1} \dot{\Rightarrow} \langle t \rangle^{\gamma_0}$ より，

$\square = x_1$ の時

$\gamma_1 \geq \gamma_0,\ x_1 : \langle t \rangle^{\gamma_1}$ より，x_1 のスコープは γ_1 であり，γ_1 スコープのコード変数は，γ_0 スコープでは一般に使用できないので $\gamma_1 \geq \gamma_0,\ x_1 : \langle t \rangle^{\gamma_1} \vdash x_2 : \langle t \rangle^{\gamma_0}$ は成り立たず，型がつかない

$\square = x_2$ の時

$\gamma_2 \geq \gamma_1,\ \gamma_1 \geq \gamma_0,\ x_2 : \langle t \rangle^{\gamma_2}$ より，x_2 のスコープは γ_2 であり，γ_2 スコープのコード変数は，γ_0 スコープでは一般に使用できないので $\gamma_2 \geq \gamma_1,\ \gamma_1 \geq \gamma_0,\ x_2 : \langle t \rangle^{\gamma_2} \vdash x_2 : \langle t \rangle^{\gamma_0}$ は成り立たず，型がつかない

$\square = \underline{\mathbf{int}}7$ の時

$\underline{\mathbf{int}}7$ は定数であるので，どの Classifier γ_i においても型が付く．

このように，(少なくとも) 上記の例については安全な式と危険な式 (Scope extrusion が起こる式) を正しく選別できていることがわかった．

5.2 型安全性について

本研究の型システムに対する型保存 (Subject Reduction) 定理について述べる．型保存定理は，(証明できれば) 進行 (Progress) 定理とあわせて型システムの健全性を導く定理である．

(型保存性) $\vdash e : t\ ;\ \sigma$ かつ $e \rightsquigarrow e'$ であれば，$\vdash e' : t\ ;\ \sigma$ である．

この定理は reset0-shift0 の計算規則が多相性を持たない場合には容易に証明できるが，多相性については精密な扱いが必要であり，現段階では，型保存定理の証明は進行中である．

第6章　型推論

この章では本体系の言語によって書かれたプログラムの型を推論するための型推論アルゴリズムを述べる．

本研究の型推論アルゴリズムは Γ, L, σ, e が与えられたとき，$\Gamma \vdash^L e : t;\ \sigma$ が成立するような t があるかどうか判定し，その型 t を返すものである．

型推論アルゴリズムは主に以下の 2 ステップから構成する

- 制約生成:与えられた項に対して，型および classifier に関する制約を返す
- 制約解消:その得られた制約を解消し，その制約を満たす代入 Θ を返す

6.1　型システム T_2 の導入

制約生成のための形ステム T_2 を導入する．これは 5 章で与えた型システム T_1 をトップダウンの制約生成に適した形に変形したものである．

T_2 の設計指針は以下のとおりである．

- T_1 と T_2 は「型付けできる」という関係として等価である．
- T_2 は結論側の式のトップレベルの形だけで，適用可能な型付け規則が一意に定まる．
- T_2 は，制約生成をする．

T_2 の設計にあたって解決すべき問題は subsumption 規則である．すなわち，subsumption 規則 は，どのような項に対しても適用ができるので，上で述べた一意性が成立しない．そこで，型付け規則と subsumption 規則 を組み合わせた規則とすることで，問題を解決した．

以下で,「var1」等といった表記は,「型システム T_1 の var1 規則を subsumption 規則と組み合わせた形に改訂し制約を生成する規則」であるということを表す．また，型付け規則の右側に記述した Constr;... は (型付け規則を下から上にむけて使うとき)，Constr 以下の制約が生成される，という意味である．

また，型 t_1, t_2 に対する $\geq$ の記号は以下の意味である．

- $\langle t_1 \rangle^{\gamma_1} \geq \langle t_2 \rangle^{\gamma_2}$ は，「$t_1 = t_2$ かつ $\gamma_1 \geq \gamma_2$
- $\langle t \rangle^{\gamma}$ の形でない t_1, t_2 に対しては，$t_1 = t_2$

図 6.1: 制約生成における $\geq$ の意味の定義

型システム T_2 は $t_1 \geq t_2$ の形のまま，制約として生成する．

(var0)

$$\frac{(x : t') \in \Gamma}{\Gamma \vdash x : t;\ \sigma}\ Constr;\ \Gamma \models t \geq t'$$

(var1)

$$\frac{(u : t)^{\gamma'} \in \Gamma}{\Gamma \vdash^{\gamma} u : t;\ \sigma}\ Constr;\ \Gamma \models \gamma \geq \gamma'$$

(const)

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

(app0)

$$\frac{\Gamma \vdash^{\gamma} e_1 : t_2 \xrightarrow{\sigma} t_1;\ \sigma \quad \Gamma \vdash^{\gamma} e_2 : t_2;\ \sigma}{\Gamma \vdash^{\gamma} e_1\, e_2 : t;\ \sigma}\ Constr;\ \Gamma \models t \geq t_1$$

(app1)

$$\frac{\Gamma \vdash e_1 : t_2 \to t_1;\ \sigma \quad \Gamma \vdash e_2 : t_2;\ \sigma}{\Gamma \vdash e_1\, e_2 : t;\ \sigma}\ Constr;\ \Gamma \models t \geq t_1$$

(lambda0)

$$\frac{\Gamma,\ x : t_1 \vdash e : t_2;\ \sigma'}{\Gamma \vdash \lambda x.e : t;\ \sigma}\ Constr;\ t = t_1 \xrightarrow{\sigma'} t_2,\ \Gamma \models \sigma \geq \sigma'$$

(lambda1)

$$\frac{\Gamma,\ (u : t_1)^{\gamma} \vdash^{\gamma} e : t_2;\ \sigma}{\Gamma \vdash^{\gamma} \lambda u.e : t;\ \sigma}\ Constr;\ t = t_1 \to t_2$$

(if)

$$\frac{\Gamma \vdash^L e_1 : \texttt{Bool};\ \sigma \quad \Gamma \vdash^L e_2 : t;\ \sigma \quad \Gamma \vdash^L e_3 : t;\ \sigma}{\Gamma \vdash^L \textbf{if}\ e_1 \textbf{then}\ e_2 \textbf{else}\ e_3\ :\ t;\ \sigma}\ Constr;\ (none)$$

(code-lambda)

$$\frac{\Gamma, \gamma' \geq \gamma, x : \langle t_1 \rangle^{\gamma'} \vdash e : \langle t_2 \rangle^{\gamma'};\ \sigma}{\Gamma \vdash \underline{\lambda} x.e\ :\ t;\ \sigma}\ Constr;\ \Gamma \models t \geq \langle t_1 \to t_2 \rangle^{\gamma}$$

(code-let)

$$\frac{\Gamma \vdash e_0 : \langle t' \rangle^{\gamma_0}; \sigma \quad \Gamma, \gamma_1 \geq \gamma_0, x : \langle t' \rangle^{\gamma_1} \vdash e_1 : \langle t' \rangle^{\gamma_1}; \sigma}{\Gamma \vdash \underline{\textbf{let}}\ x = e_0\ \underline{\textbf{in}}\ e_1 : t; \sigma}\ Constr;\ \Gamma \models t \geq \langle t' \rangle^{\gamma_0}$$

(reset0)

$$\frac{\Gamma \vdash e : \langle t' \rangle^{\gamma};\ \langle t' \rangle^{\gamma}, \sigma}{\Gamma \vdash \textbf{reset0}\ e\ :\ t;\ \sigma}\ Constr;\ \Gamma \models t \geq \langle t' \rangle^{\gamma}$$

(shift0)

$$\frac{\Gamma,\ k : \langle t_1 \rangle^{\gamma_1} \overset{\sigma}{\Rightarrow} \langle t_0 \rangle^{\gamma_0} \vdash e : \langle t_0 \rangle^{\gamma_0}; \sigma}{\Gamma \vdash \textbf{shift0}\ k \to e : t\ ;\ t_2, \sigma}\ Constr;\ \Gamma \models t \geq \langle t_1 \rangle^{\gamma_1},\ t_2 = \langle t_0 \rangle^{\gamma_0},\ \Gamma \models \gamma_1 \geq \gamma_0$$

(throw0)

$$\frac{\Gamma,\ k : t' \vdash v : \langle t_1 \rangle^{\gamma'}; \sigma}{\Gamma,\ k : t' \vdash \textbf{throw}\ k\ v : t; \sigma}\ Constr;\ \Gamma \models t \geq \langle t_0 \rangle^{\gamma_2},\ (\Gamma,\ k{:}\ t') \models \gamma_2 \geq \gamma_0,\ t' = \langle t_1 \rangle^{\gamma_1} \overset{\sigma}{\Rightarrow} \langle t_0 \rangle^{\gamma_0},\ \Gamma \models \gamma_1 \cup \gamma_2 \geq \gamma'$$

(code)

$$\frac{\Gamma \vdash^{\gamma} e : t_1;\ \sigma}{\Gamma \vdash \texttt{<}e\texttt{>} : t;\ \sigma}\ Constr;\ t \geq \langle t_1 \rangle^{\gamma}$$

この新しい型システム T_2 は T_1 と同じ型付けをあたえる．

6.2 制約生成

制約生成では，与えられた項 e に対して，T_2 を型付け規則を下から上の向きに適用することで，制約を生成する．生成する制約は，それぞれの型付け規則の右側に $Constr;\ ...$ と書いてあるものである．それらの制約を，型付けに従って生成していくことで，制約の集合が生成される

T_2 の型付け規則を適用する時，型付け規則の下の型判断に存在せず，上の型判断，あるいは制約中にのみ存在する型や classifier があるときは，これらを新しい型変数や classifier 変数として生成する．なお，code-lambda 規則での新しい classifier は，classfier 変数ではなく，classifier 定数とする[1]．

制約生成アルゴリズム:

- Γ : 型文脈
- e : 項
- t : 型
- σ : answer type の列
- L : レベル (現在レベル 0, または コードレベル γ)
- C : 制約

入力 $\Gamma,\ e,\ t,\ \sigma,\ L$

出力 C

$\Gamma \vdash^L e : t;\ \sigma$ から始めて，T_2 のおける型付けを下から上に向かって行う．型付けがどこかで失敗するとき，制約生成は失敗する．型付けが成功したとき，生成された制約 C_i の和集合を C としてそれを返す．

6.3 制約の解消

入力を Γ, L, e, t, σ として，前章の制約生成アルゴリズムを走らせ，それが成功して C という制約を生成したとき，

- T_1 で $\Gamma \vdash^L e : t;\ \sigma$ が導出可能ならば，C を満たす解が存在し，
- C を満たす解が存在すれば、ある代入 Θ に対して，$\Theta(\Gamma) \vdash^L \Theta(e) : \Theta(t);\ \Theta(\sigma)$ が T_1 で導出可能である．

という性質が成立する．

つまり，型推論問題を得られた制約を成立するような代入 Θ が存在するかどうかという問題に帰着できる．

制約は以下の文法で与えられたものの有限集合である．

[1]制約生成のでは，変数と定数に違いはないが，制約を解消するとき，cassifier 定数に対する代入はしない．という違いがある

$$\Gamma \models t^0 \geq t^0$$
$$\Gamma \models c \geq c$$
$$\Gamma \models \sigma \geq \sigma$$
$$t^0 = t^0$$
$$t^1 = t^1$$

ただし、ここで t_0, t_1, c, σ は以下の文法で定義される。

$$t^0 ::= \alpha^0 \mid \texttt{Int} \mid \texttt{Bool} \mid t^0 \xrightarrow{\sigma} t^0 \mid \langle t^1 \rangle^c$$
$$t^1 ::= \alpha^1 \mid \texttt{Int} \mid \texttt{Bool} \mid t^1 \to t^1$$
$$c ::= \gamma \mid d \mid c \cup c$$
$$\sigma ::= \sigma_x \mid \epsilon \mid t^0, \sigma$$

図 6.2: 制約の定義

t_0, t_1, c, σ はそれぞれ、レベル 0 型，レベル 1 型，レベル 0 型の列，classifier をあらわす表現 (メタ変数)，answer type の列である．また，α^i はレベル i の型変数，γ は classifier 変数，σ_x は σ 変数である． また，d は，固有変数条件をもつ classifier 変数のことであり，型推論のあいだは，これは実質的に定数として扱われる[2]．

また，Γ は，一般の型文脈であるが，不要な情報を落として以下の形にする．

$$\Delta ::= \emptyset \mid \Delta, (d \geq c) \mid \Delta, (x : t) \mid \Delta, (u : t)^{\Delta}$$

制約の解消とは，制約が与えられたとき，その解となる代入 Θ を求めることである．代入 Θ は、型変数 α^0, α^1 への型の代入と，classifier 変数 γ への classifier の代入とから構成される．

6.3.1 typeinf1: 制約の解消アルゴリズム (前半)

$t^0 = s^0$ と $t^1 = s^1$ の形の制約は，単一化アルゴリズムによって解くことができる．それを解いた結果，α^0，α^1, γ に対する代入が生じる．もしくは「解なし」という結果となる．

$\Delta \models t^0 \geq s^0$ の形の制約は，両方ともが型変数の場合以外は，簡単に解ける．(その結果として、$t^i = s^i$ の型の制約や，$\Delta \models c \geq c$ の形の制約を生む可能性があるが，前者は前と同様に解けばよく，前者を解いている間にあらたに $\Delta \models t^0 \geq s^0$ の形の制約は生じない．)

[2]つまり，classifier 変数 γ に対しては代入するが，d に対しては代入しない

ここまでの段階で残る制約は，以下のものだけである．

- $\Delta \models \alpha^0 \geq \beta^0$
- $\Delta \models c_1 \geq c_2$

ここまでに出てきた代入はすべて，上記の制約に適用済みとする．つまり，$\alpha := \texttt{Int}$ という代入がでてきたら，制約中の α はすべて `Int` にしておく．その結果,「代入における左辺にでてくる型変数や classifier 変数」は，上記の制約には，残っていない．

制約解消アルゴリズム (前半) unify1

- C : 制約
- Θ : 代入

入力 C, Θ

出力 Θ' または,「単一化失敗」ここで，Θ' は Θ に C を解いて得られる代入を追加したもの

1 C の中に $t^0 = s^0$ か $t^1 = s^1$ か $\Delta \models t^0 \geq s^0$ の形の制約がなければ，代入 Θ を返す．

2 C から $t^0 = s^0$ か $t^1 = s^1$ の形の制約を選び，それを $A = B$ とする．$C_1 = C - \{A = B\}$ とする．

2-1 $A = B$ のとき，unify1(C_1, Θ) を呼び出す．

2-2 $A \neq B$ で A が型変数のとき，

2-2-1 型 B に A が現れるなら,「単一化失敗」を返す．

2-2-2 型 B に A が現れないなら，$\Theta_1 = [A := B]$ とし，unify1$(\Theta_1(C_1), \Theta(\Theta_1))$ を呼び出す．

2-3 $A \neq B$ で B が型変数のとき，A と B を入れ替えて，2-2 へ

2-4 $A \neq B$ で $A = A_1 \to A_2$，$B = B_1 \to B_2$ のとき， $C_2 = C_1 \cup \{A_1 = B_1, A_2 = B_2\}$ とし， unify1(C_2, Θ) を呼び出す

2-5 $A \neq B$ で $A = A_1 \stackrel{\sigma_1}{\to} A_2$，$B = B_1 \stackrel{\sigma_2}{\to} B_2$ のとき， $C_2 = C_1 \cup \{A_1 = B_1, A_2 = B_2, \sigma_1 = \sigma_2\}$ とし， unify1(C_2, Θ) を呼び出す

2-6 $A \neq B$ で $A = \langle A_1 \rangle^{c_1}, B = \langle B_1 \rangle^{c_2}$ のとき，$C_2 = C_1 \cup \{A_1 = B_1, c_1 = c_2\}$ とし， unify1(C_2, Θ) を呼び出す

2-7 上記のいずれでもないとき「単一化失敗」を返す.

1-3 C から $\Delta \models t^0 \geq s^0$ の形の制約を選び，それを c とする．$C_1 = C - \{c\}$ とする.

1-3-1 t^0, s^0 のどちらかが型変数でないとき，詳細は省くが，制約 θ が生成される; unify1$(C_1, \Theta(\theta))$ を呼び出す

1-3-2 t^0, s^0 がともに型変数のときつまり，c が $\Delta \models \alpha^0 \geq \beta^0$ のとき，その制約を残す

6.3.2 typeinf2: 制約の解消アルゴリズム (後半)

ここまでの段階で残る制約は，上記で述べたように以下のものだけである.

- $\Delta \models \alpha^0 \geq \beta^0$
- $\Delta \models c_1 \geq c_2$

まず，$\Delta \models c \geq c$ の形の制約の解消について述べる

この形の制約たちを，$\Delta_i \models c_i \geq c_i'$ とすると，それぞれの Δ_i は両立的であるので，$\Delta = \Delta_1 \cup \cdots \cup \Delta_n$ として，$\Delta \models c_i \geq c_i'$ を解けばよい.

(ステップ 1: classifier 変数の除去)

制約中の classifier 変数の 1 つに着目し，それを γ とする．$c_i \geq \gamma$ の形の制約 $(i = 1, 2, \cdots, I)$ と $\gamma \geq c_j'$ の形の制約 $(j = 1, 2, \cdots, J)$ をすべて消去し，以下の制約を，すべての (i, j) に対して追加する.

$$c_i \geq c_j'$$

これにより，classifier 変数は 1 つ減る．したがって，ステップ 1 を繰り返すと，classifier 変数はなくなる.

(ステップ 2: 右辺の $\cup$ の除去)

$c_1 \geq c_2 \cup c_3$ を $c_1 \geq c_2$ と $c_1 \geq c_3$ に変換する.

これにより，不等号の右辺にある $\cup$ の個数が 1 つ減る．したがって，ステップ 2 を繰り返すと，不等号の右辺にある $\cup$ はなくなる.

ステップ 2 の繰り返しがおわると，制約は，$\Delta \models c \geq d$ の形になる.

(ステップ 3: 左辺の分解)

「d は atomic」 とする.「$c_1 \cup c_2 \geq d$ ならば $c_1 \geq d$ または，$c_2 \geq d$」ということである.

これを用いて、$c \geq d$ の左辺を分解することができ,

$$\Delta \models d_1 \geq d_1' \vee \cdots \vee d_n \geq d_n'$$

となる．さらに Δ も $d1 \geq d2$ の形を「かつ」と「または」でつないだ形になる.

$\Delta \models \alpha^0 \geq \beta^0$ の形の制約は残す.

制約解消アルゴリズム (後半) unify2

- C : 制約
- Θ : 代入

入力 C, Θ (Θ には $\Delta \models \alpha^0 \geq \beta^0, \Delta \models c \geq c$ の形の制約のみがある)

出力 C, Θ ($\Delta \models \alpha^0 \geq \beta^0$ の形の制約があればそれも返す)

1 C から任意の制約 c を選び， $C' = C - \{c\}$ とする.

1-1 c が $\Delta \models c \geq c$ の形のとき,

1-1-1: ステップ 1 classifier 変数 γ を選び，$c_i \geq \gamma$ と $\gamma \geq c_j'$ をすべて消去し，制約に $c_i \geq c_j'$ を加える．すべての classifier 変数がなくなるまでステップ 1 を繰り返し，ステップ 2 へ

1-1-2: ステップ 2 $c_1 \geq c_2 \cup c_3$ を $c_1 \geq c_2$ と $c_1 \geq c_3$ に変換する．右辺に $\cup$ がなくなるまで，ステップ 2 を繰り返すことによって，制約は， $c \geq d$ となる．ステップ 3 へ

1-1-3: ステップ 3 $c_1 \cup c_2 \geq d$ を $c_1 \geq d \vee c_2 \geq d$ に変換する．その制約を Θ に加え，unify2(C', Θ) を呼び出す

1-2 解として，C, Θ と $\Delta \models \alpha^0 \geq \beta^0$ の形の制約を返す.

第7章　関連研究

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Kameyamaらの研究[5]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[7]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[2]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[1, 8]．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．

第8章　まとめと今後の課題

本研究では，効率的コード生成に有用な技法である let 挿入を，型安全に実現するための言語と型システムについて述べた．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 [2] などに基づき，多段階の for ループを飛び越えた let 挿入を実現するために，shift0/reset0 を持つコード生成体系を設計した．須藤らの研究で精密化された環境識別子 (Environment Classifier) に join (∪) を導入することで，計算の順序を変更するようなコントロールオペレータ（shift0/reset0）を扱えるようにし，安全に多段階の let 挿入を行えるように型システムを構築した．このような let 挿入が束縛子を越えるケースは，ループにおける不変式の括り出しなどの有用な最適化を含むが，これまでの研究では一般的な let 挿入を安全に実現した体系の提案はなく，我々の知る限り本研究がはじめてである．

今後の課題として，まずあげられるのは，進行 (Progress) の性質および型推論アルゴリズムの開発である．また，理論的には Kiselyov らのグローバルな参照を持つ体系との融合が可能になれば，広い範囲のコード生成技法・最適化技法をカバーできるため極めて有用である．また，既存の MetaOCaml との比較においては，2 レベルのみのコード生成に限定している点や run (生成したコードの実行) や cross-stage persistence (現在ステージの値をコードに埋め込む機能) などに対応していない点が欠点であり，これらの拡張が可能であるかどうかの検討は非常に興味深い将来課題である．

謝辞

本研究に関して，終始ご指導ご鞭撻を頂きました亀山幸義先生に深く感謝いたします．また，研究発表の仕方など有益な助言を下さった海野広志先生に感謝いたします．最後に，研究に関して様々な議論をして下さった薄井千春君に，日頃研究を様々な形でサポートして頂きましたプログラム論理研究室の皆様に感謝いたします．

参考文献

[1] Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. *ICFP 2011*.

[2] 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.

[3] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. Refined environment classifiers - type- and scope-safe code generation with mutable cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, pp. 271–291, 2016.

[4] 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.

[5] 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.

[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.

[7] 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.

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

[9] 杉浦啓介, 亀山幸義. コード実行機能と計算エフェクトを持つ型付きマルチステージ言語. コンピュータ ソフトウェア, Vol. 28, No. 1, pp. 217–229, 2011.

[10] 鈴木輝信, 亀山幸義. 階層化コントロールオペレータに対する型システムの構築. 情報処理学会論文誌プログラミング（PRO）, Vol. 48, No. 10, pp. 138–150, jun 2007.

[11] 須藤悠斗. 計算エフェクトを持ち型安全なコード生成系, 2015. 筑波大学システム情報工学研究科, 修士論文.

[12] R. Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pp. 184–, Washington, DC, USA, 1996. IEEE Computer Society.

[13] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *CoRR*, Vol. abs/cs/0508048, , 2005.

[14] Olivier Danvy. Pragmatics of type-directed partial evaluation. In *Partial Evaluation*, pp. 73–94. Springer, 1996.

[15] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pp. 30–50. Springer, 2004.

[16] Walid Taha. A gentle introduction to multi-stage programming, part ii. In *Generative and Transformational Techniques in Software Engineering II*, pp. 260–290. Springer, 2008.

[17] Oleg Kiselyov. The design and implementation of ber metaocaml. In *International Symposium on Functional and Logic Programming*, pp. 86–102. Springer, 2014.

[18] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, Vol. 46, pp. 127–136. ACM, 2010.