

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 Sudo 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

| | | |
|-------|----------------------|----|
| 第 1 章 | はじめに | 1 |
| 第 2 章 | 背景 | 3 |
| 2.1 | コード生成言語 | 3 |
| 2.1.1 | コード生成の例 | 4 |
| 2.2 | shift0/reset0 | 4 |
| 2.3 | コード生成と let 挿入 | 5 |
| 2.3.1 | コードコンビネータ方式のプログラム例 | 6 |
| 2.3.2 | コード生成における let 挿入 | 7 |
| 第 3 章 | 環境識別子による型システムの構築 | 10 |
| 3.1 | 先行研究のアイデア | 10 |
| 3.2 | 本研究: 環境識別子の拡張 | 11 |
| 3.3 | 本研究: 型システムの構築 | 13 |
| 第 4 章 | 対象言語: 構文と意味論 | 15 |
| 4.1 | 構文の定義 | 15 |
| 4.2 | 操作的意味論 | 16 |
| 第 5 章 | 型システム | 18 |
| 5.1 | 型付け例 | 23 |
| 5.2 | 型安全性について | 23 |
| 第 6 章 | 型推論 | 24 |
| 6.1 | 制約生成 | 24 |
| 6.2 | 型システム T_2 の導入 | 24 |
| 第 7 章 | 制約の解消 | 28 |
| 7.1 | 制約の解消アルゴリズム (前半) | 29 |
| 7.2 | 制約の解消アルゴリズム (後半) | 29 |
| 7.3 | 制約の解消アルゴリズム (後半のおまけ) | 30 |

| | |
|-------------------------|-----------|
| 第 8 章 型安全性の証明 | 31 |
| 8.1 型保存 | 31 |
| 8.2 進行 | 31 |
| 第 9 章 関連研究 | 32 |
| 第 10 章 まとめと今後の課題 | 33 |
| 謝辞 | 34 |
| 参考文献 | 35 |

図 目 次

| | | |
|-----|---|----|
| 3.1 | 危険な例 | 10 |
| 3.2 | 型によるコードレベル変数のスコープ表現 | 11 |
| 3.3 | コード生成による let 挿入 | 12 |
| 3.4 | 誤った環境識別子の順序 | 12 |
| 3.5 | 正しい環境識別子の順序 | 12 |
| 4.1 | 対象言語の構文の定義 | 15 |
| 4.2 | 評価文脈 | 16 |
| 4.3 | $e \rightsquigarrow e'$ の評価関係 | 16 |
| 4.4 | 計算規則 | 17 |
| 4.5 | 継続への代入 | 17 |
| 4.6 | コードコンビネータの規則 | 17 |
| 5.1 | 基本型, 環境識別子の定義 | 18 |
| 5.2 | (レベル 0 の) 継続の型の定義 | 18 |
| 5.3 | 型判断の定義 | 19 |
| 5.4 | 型文脈の定義 | 19 |
| 5.5 | $\Gamma \models \gamma \geq \gamma$ の形に対する型導出規則 | 20 |
| 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 | 21 |
| 5.7 | コードレベルのラムダ抽象の型導出規則 | 21 |
| 5.8 | コントロールオペレータに対する型導出規則 | 22 |
| 5.9 | コード生成に関する Subsumption の型導出規則 | 22 |

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を設計した点に新規性がある．

第2章 背景

oishi said: ここでは、コード生成 + s0/r0 で let 挿入を用いたコード生成ができるよ。その安全性は、型システムで保証するよ。既存の型システムを参考に s0/r0 を用いる let 挿入が安全に行えるようにしたやつだよ。みたいなことを書く

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、実行などが行える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 $\langle e \rangle$. : コード生成 $\langle e \rangle$. をコードと呼び、コード化とは、項 e を Bracket $\langle \rangle$. で囲むことにより、 e の評価を遅らせることである。Bracket $\langle \rangle$. 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 =  $\langle 1 + 2 \rangle$ .;;  
val x : int code =  $\langle 1 + 2 \rangle$ .
```

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 $\langle 1 + 2 \rangle$. が結果として返されている。val x : int code = $\langle 1 + 2 \rangle$. とは、値 x は、int code 型であり、その値は $\langle 1 + 2 \rangle$. という意味である。

Escape $\sim e$: コードの埋め込み $\sim e$ に対する呼び名は特にはないが、Escape \sim はコード $\langle e \rangle$. 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 $\langle e \rangle$. 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, $\text{let } y = .<.\tilde{x} * .\tilde{x}>.$ を評価すると, \tilde{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

[oishi said: power 関数の例でいいのだろうか](#)

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. 継続とは, 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, と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. また, `shift0/reset0` は `shift/reset` よりも単純な CPS 変換で意味が与えられていて, 純粋なラムダ式で表せるために形式的に扱いやすいという利点がある. `reset0` は **reset0** (M) というように表し, 継続の範囲を M に限定するという意味となる. `shift0` は **shift0** $k \rightarrow M$ 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.

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

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` に `x` を適用させることで、`3 + x` が得られる。すると、`let x = 5 in k x` は `let x = 5 in 3 + x` に評価される。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、先行研究にならな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

例えば、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

まず、(完成した) コードは、 $\langle 3 \rangle$ や $\langle 3 + 5 \rangle$ 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。

$$\begin{aligned} (\underline{\text{int}}\ 3) &\rightsquigarrow^* \langle 3 \rangle \\ (\underline{\text{int}}\ 3) \underline{+} (\underline{\text{int}}\ 5) &\rightsquigarrow^* \langle 3 + 5 \rangle \end{aligned}$$

$\underline{\text{int}}$ 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。

$$\begin{aligned} \underline{\lambda}x.x \underline{+} (\underline{\text{int}}\ 3) &\rightsquigarrow^* \langle \lambda u.u + 3 \rangle \\ (\underline{\lambda}x.x \underline{+} (\underline{\text{int}}\ 3)) \underline{@} (\underline{\text{int}}\ 5) &\rightsquigarrow^* \langle (\lambda u.u + 3)\ 5 \rangle \end{aligned}$$

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 $\langle u \rangle$ (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。

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

$$\begin{aligned} \underline{\text{let}}\ x = (\underline{\text{int}}\ 3)\ \underline{\text{in}}\ x \underline{+} (\underline{\text{int}}\ 7) \\ \rightsquigarrow^* \langle \text{let } u = 3 \text{ in } u + 7 \rangle \end{aligned}$$

実は、 $\underline{\text{let}}$ 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。

$$\begin{aligned} (\underline{\lambda}x.x \underline{+} (\underline{\text{int}}\ 7)) \underline{@} (\underline{\text{int}}\ 3) \\ \rightsquigarrow^* \langle \text{let } u = 3 \text{ in } u + 7 \rangle \end{aligned}$$

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。

$$\underline{\lambda}y.\underline{\text{let}}\ x = y\ \underline{\text{in}}\ \underline{\lambda}y'.x \underline{+} y \rightsquigarrow^* \langle \lambda u.\lambda u'.u + u' \rangle$$

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。

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

for を入れ子にすると、入れ子の for 式が生成できる。

```
for  $x = (\text{int } 3)$  to  $(\text{int } 7)$  do  
  for  $y = (\text{int } 1)$  to  $(\text{int } 9)$  do  
    aryset  $\langle a \rangle\ (x, y)\ (\text{int } 0)$   
 $\rightsquigarrow^*$  <for  $i = 3$  to  $7$  do  
  for  $j = 1$  to  $9$  do  
     $a[i, j] \leftarrow 0$ >
```

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、

```
<for  $i = 3$  to  $7$  do  
  for  $j = 1$  to  $9$  do  
     $a[i, j] \leftarrow e$ >
```

というコードの代わりに

```
<let  $z = e$  in  
  for  $i = 3$  to  $7$  do  
    for  $j = 1$  to  $9$  do  
       $a[i, j] \leftarrow z$ >
```

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。

```

reset0 (for  $x = (\text{int } 3)$  to  $(\text{int } 7)$  do
  (for  $y = (\text{int } 1)$  to  $(\text{int } 9)$  do
    shift0  $k_1 \rightarrow \text{let } z = e \text{ in}$ 
    throw  $k_1$  (aryset  $\langle a \rangle (x, y) z$ )))

```

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。

$$\begin{aligned}
 &\text{reset0 } v \rightarrow v \\
 &\text{reset0 } (E[\text{shift0 } k \rightarrow \dots (\text{throw } k v) \dots]) \\
 &\quad \rightarrow \dots (\text{reset0}(E[v])) \dots
 \end{aligned}$$

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 $x = \dots$ to \dots do for $y = \dots$ 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。

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。

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。

```

reset0 (for  $x = (\underline{\text{int}}\ 3)$  to (int 7) do
reset0 (for  $y = (\underline{\text{int}}\ 1)$  to (int 9) do
  shift0  $k_2 \rightarrow$  shift0  $k_1 \rightarrow$  let  $z = e$  in
    throw  $k_1$  (throw  $k_2$  (aryset  $\langle a \rangle (x, y) z$ ))))

```

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 $\langle 7 \rangle$ 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。

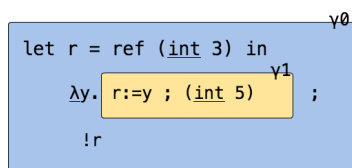
第3章 環境識別子による型システムの構築

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年の亀山らの研究 [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、 $\langle \text{int} \rangle^\alpha$ 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。



```
let r = ref (int 3) in
  λy. r := y ; (int 5) ;
!r
```

図 3.1: 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、 $\langle u \rangle$ 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 $\langle u \rangle$ 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 $\langle \text{int} \rangle^{\gamma_0} \text{ref}$ 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。

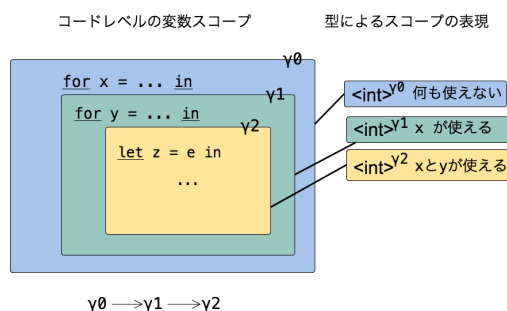


図 3.2: 型によるコードレベル変数のスコープ表現

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 (U)` 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。その概形は下記の 1 つ目の図である。

上の方の図で、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、

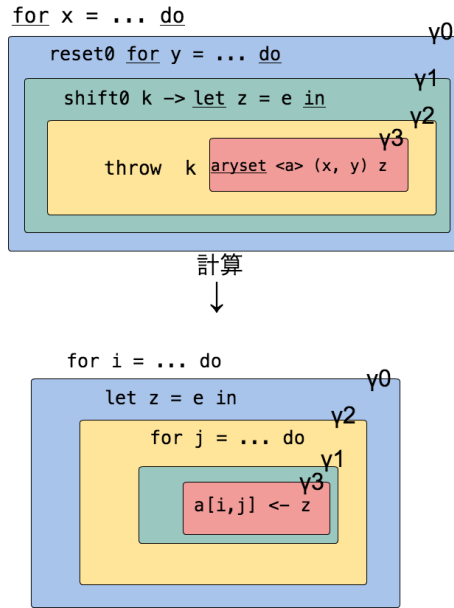


図 3.3: コード生成による let 挿入

$$\gamma_0 \longrightarrow \gamma_1 \longrightarrow \gamma_2 \longrightarrow \gamma_3$$

図 3.4: 誤った環境識別子の順序

という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を見ると（上記の下図はコードの中身を示している）， 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 で使える変数を使つ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 の変数も使つて構わない．これらを考慮すると，環境識別子の間の順序は以下の図となる．

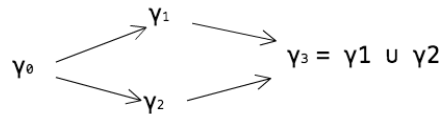


図 3.5: 正しい環境識別子の順序

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、

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

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、

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

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 (通常の間数型は \rightarrow 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。

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

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。

なお，継続が作用する型 (上記の $\langle t_1 \rangle^{\eta_1}$ 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．

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、`+` や `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。

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

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。

$\text{let } x = e_1 \text{ in } e_2$ および $\text{let } x = e_1 \text{ in } e_2$ は、それぞれ、 $(\lambda x.e_2)e_1$ $(\lambda x.e_2) @ e_1$ の省略形である。前述の例でのべた **for** は、コード構築定数とコードレベル適用 \cap 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。

$$E ::= [] \mid E e^0 \mid v E \\ \mid \text{if } E \text{ then } e^0 \text{ else } e^0 \mid \text{reset0 } E \mid \underline{\lambda}u.E$$

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、 $\underline{\lambda}u.u \pm []$ 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 $r \rightarrow l$ により、評価関係 $e \rightsquigarrow e'$ を次のように定義する。

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

図 4.3: $e \rightsquigarrow e'$ の評価関係

計算規則は以下の通り定義する。

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。

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

$$\begin{aligned}
& (\lambda x.e) v \rightarrow e\{x := v\} \\
& \text{if } true \text{ then } e_1 \text{ else } e_2 \rightarrow e_1 \\
& \text{if } else \text{ then } e_1 \text{ else } e_2 \rightarrow e_2 \\
& \underline{\lambda}x.e \rightarrow \underline{\lambda}u.(e\{x := \langle u \rangle\}) \\
& \underline{\lambda}u.\langle e \rangle \rightarrow \langle \lambda u.e \rangle \\
& \text{reset0 } v \rightarrow v \\
& \text{reset0}(E[\text{shift0 } k \rightarrow e]) \rightarrow e\{k \Leftarrow E\}
\end{aligned}$$

図 4.4: 計算規則

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

図 4.5: 継続への代入

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。

$$\begin{aligned}
& \underline{\text{int}} n \rightarrow \langle n \rangle \\
& \langle e_1 \rangle \underline{@} \langle e_2 \rangle \rightarrow \langle e_1 e_2 \rangle \\
& \langle e_1 \rangle \underline{+} \langle e_2 \rangle \rightarrow \langle e_1 + e_2 \rangle \\
& \underline{\text{if}} \langle e_1 \rangle \langle e_2 \rangle \langle e_3 \rangle \rightarrow \langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rangle
\end{aligned}$$

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.

$$\begin{aligned} b &::= \text{int} \mid \text{bool} \\ \gamma &::= \gamma_x \mid \gamma \cup \gamma \end{aligned}$$

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.

$L ::= \mid \gamma$ 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, $L = \mid$ 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.

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

図 5.2: (レベル 0 の) 継続の型の定義

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 $n = 1, \dots, k$ 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のみにあられ、生成されるコードの中にはあられないため、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、通常関数とは区別しており、そのため、継続の型も通常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。

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

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。

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

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

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

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

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.

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

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

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

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

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

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.

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

oishi said:

$\underline{\lambda}$. は必要あるかどうか

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

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.

$$\begin{array}{c}
\frac{\Gamma \vdash e : \langle t \rangle^\gamma ; \langle t \rangle^\gamma, \sigma}{\Gamma \vdash \mathbf{reset0} \ e : \langle t \rangle^\gamma ; \sigma} \\
\\
\frac{\Gamma, k : \langle t_1 \rangle^{\gamma_1} \xrightarrow{\sigma} \langle t_0 \rangle^{\gamma_0} \vdash e : \langle t_0 \rangle^{\gamma_0}; \sigma \quad \Gamma \models \gamma_1 \geq \gamma_0}{\Gamma \vdash \mathbf{shift0} \ k \rightarrow e : \langle t_1 \rangle^{\gamma_1} ; \langle t_0 \rangle^{\gamma_0}, \sigma} \\
\\
\frac{\Gamma, k : \langle t_1 \rangle^{\gamma_1} \xrightarrow{\sigma} \langle t_0 \rangle^{\gamma_0} \vdash v : \langle t_1 \rangle^{\gamma_1 \cup \gamma_2}; \sigma \quad \Gamma \models \gamma_2 \geq \gamma_0}{\Gamma, k : \langle t_1 \rangle^{\gamma_1} \xrightarrow{\sigma} \langle t_0 \rangle^{\gamma_0} \vdash \mathbf{throw} \ k \ v : \langle t_0 \rangle^{\gamma_2}; \sigma}
\end{array}$$

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.

oishi said: \geq の左右の γ は逆? 一般に項の内部に入るにしたがって, 使えるコードレベル変数は増えていくので, 逆のような気がする. $\gamma_1 \geq \gamma_0$ の意味は γ_0 より γ_1 のほうが使えるコードレベル変数は多いという意味である

$$\begin{array}{c}
\frac{\Gamma \vdash e : \langle t \rangle^{\gamma_1}; \sigma \quad \Gamma \models \gamma_2 \geq \gamma_1}{\Gamma \vdash e : \langle t \rangle^{\gamma_2}; \sigma} \\
\\
\frac{\Gamma \vdash^{\gamma_1} e : t ; \sigma \quad \Gamma \models \gamma_2 \geq \gamma_1}{\Gamma \vdash^{\gamma_2} e : t ; \sigma} \\
\\
\frac{\Gamma \vdash^\gamma e : t^1; \sigma}{\Gamma \vdash \langle e \rangle : \langle t^1 \rangle^\gamma; \sigma} \\
\\
\frac{\Gamma \vdash e : t; \langle t' \rangle^{\gamma_1}, \sigma \quad \Gamma \models \gamma_2 \geq \gamma_1}{\Gamma \vdash e : t; \langle t' \rangle^{\gamma_2}, \sigma}
\end{array}$$

図 5.9: コード生成に関する 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.

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

この式 e_1 に対して、もし、 $t = \%7$ あるいは $t = x_1$ であれば、 e_1 は型付け可能である. 一方、 $t = x_2$ であれば、 e_1 は型付けできない.

$$\begin{aligned} e_2 = & \text{reset0 } \underline{\text{let}} \ x_1 = \%3 \ \underline{\text{in}} \\ & \text{reset0 } \underline{\text{let}} \ x_2 = \%5 \ \underline{\text{in}} \\ & \text{shift0 } k_2 \rightarrow \text{shift0 } k_1 \rightarrow \underline{\text{let}} \ y = t \ \underline{\text{in}} \\ & \text{throw } k_1 \ (\text{throw } k_2 \ (x_1 \pm x_2 \pm y)) \end{aligned}$$

この式 e_2 に対して、もし $t = \%7$ であれば e_1 は型付け可能である. 一方、 $t = x_2$ あるいは $t = x_1$ であれば、 e_1 は型付けできない.

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を正しく峻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

概要: 以下の2ステップから構成

- 制約生成:与えられた項にたいして、(型およびクラシファイアに関する) 制約を返す。
- 制約を解く。

6.1 制約生成

これは、もの型システム (T_1 とする) を「トップダウンでの制約生成向け型システム (T_2 とする)」に変形することであたえる。

T_2 の設計指針:

- 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、term-oriented である。(結論側の式のトップレベルの形だけで、どの型付けルールを適用可能か、一意的にわかる。)
- T_2 は、制約生成をする。(結論側の式の要素は変数として、「それがこういう形でなければいけない」という条件は、制約の形で「生成」する。)

以上をどう満たすか? ポイントは、subsumption rule の適用タイミング (なるべく subsumption rule を適用するのを避けたい) である。

6.2 型システム T_2 の導入

subsumption rule が出現する場所を限定することができる。特に、ルールと、その直後に subsumption がつかわれる場合を考えてみよう。以下で、「var1」等といった表記は、「もともとある var1 ルールを subsumption 規則と組み合わせた形に改訂したもの」である。また、横棒の右に書いてある Constr;... は (ルール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が、とりあえず、(以下の意味にしたがつて分解はせずに) $t_1 \geq t_2$ の形のまま、制約として生成する。

- $\langle t_1 \rangle^{\gamma_1} \geq \langle t_2 \rangle^{\gamma_2}$ 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、 $t_1 = t_2$ 。

(型推論のプロセスの最中では、 t_1, t_2 はメタ型変数。治筈キ、謹此その場合、どちらの形かは決定できないので、 $t_1 \geq t_2$ を上記の意味にしたがって、「ほどく」ことはできない。なので、 $t_1 \geq t_2$ という形のまま制約として生成する。)

(亀山メモ: ただ、もしかすると、「レベル0の型変数」と「レベル1の型変数」を最初からわけておく方法もあるかもしれない。そうすると、上記はとける?)

oishi said: 以下のルールにおいて、Constraint の $\Gamma \models t \geq t'$ の $\Gamma \models$ はルールを適用した瞬間の Γ の情報は制約解消において使わないから必要ない?

一貫性をもたせるのなら、 $t = t'$ の形の Constraint も $\Gamma \models t = t'$ とする必要がある?

(var0)

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

(var1)

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

(const)

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

(app0)

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

(app1)

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

(lambda0)

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

(lambda1)

oishi said: コードの中で、shift0/reset0 は使わないので、 σ は ϵ ではなく、 σ は ϵ としておいてもよい

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

(if)

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

(code-lambda)

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

(reset0)

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

(shift0)

$$\frac{\Gamma, k : \langle t_1 \rangle^{\gamma_1} \xrightarrow{\sigma} \langle t_0 \rangle^{\gamma_0} \vdash e : \langle t_0 \rangle^{\gamma_0}; \sigma}{\Gamma \vdash \text{shift0 } k \rightarrow e : t; t_2, \sigma} \text{Constr}; \Gamma \models t \geq \langle t_1 \rangle^{\gamma_1}, t_2 = \langle t_0 \rangle^{\gamma_0}, \Gamma \models \gamma_1 \geq \gamma_0$$

(throw0)

oishi said: throw0 ルールにのみ σ part の subsumption ルールを適用すればおk ?

変更前

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

変更後

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

(code)

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

この新しい型システム T_2 は (hopefully) T_1 と同じ型付けをあたえる。

型推論では、与えられた項 e に対して、 T_2 を「下から上」の向きに適用して、Constraint を生成する。(もちろん、途中で、つまってしまったら、型推論は失敗する。

下から上に行くときに、「横棒の下には存在しないで、横棒の上、あるいは、Constraint の中にのみ存在する」型や classifier があるときは、これらを型変数や classifier 変数として生成する。

なお、code-lambda 規則での新しい classifier は、classifier 変数ではなく、classifier 定数とする。(この段階では、変数も定数も差はないが、制約を解消するとき、classifier 定数に対する代入はしない、という違いがある。)

第7章 制約の解消

制約生成が成功したとき (つまり、途中で「つかかかったり」しないとき)、以下の性質が成立するはずである。

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、

- $\Gamma \vdash^L e : t; \sigma$ 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、 $\Gamma \theta \vdash^L e \theta : t \theta; \sigma \theta$ が導出可能である。

という性質が成立することができる。(これが成立すれば、もともとの型推論問題を、型制約の解消問題に帰着できたことになる。)

というわけで、制約の解消をはじめよう。制約は以下の文法で与えられたものの有限集合である。

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

ただし、ここで t_0, t_1, c, σ は以下の文法で定義される。

$$\begin{aligned} t^0 &::= t_x^0 \mid \text{Int} \mid \text{Bool} \mid t^0 \xrightarrow{c} t^0 \mid \langle t^1 \rangle^c \\ t^1 &::= t_x^1 \mid \text{Int} \mid \text{Bool} \mid t^1 \rightarrow t^1 \\ c &::= \gamma \mid d \mid c \cup c \\ \sigma &::= \epsilon \mid t^0, \sigma \end{aligned}$$

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)、である。また、 t_x^i はそれぞれのレベルの型変数、 γ は classifier 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 (つまり、 d に対しては代入しない。classifier 変数 γ に対しては代入する。)

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してよい。

$$\Delta ::= \dots \mid \Delta, d \geq c$$

(左辺は、固有変数なので、classifier 定数である。右辺は一般の classifier 式がなんでも来る可能性がある。)

制約の解消とは、制約 (上記の形の有限集合) が与えられたとき、その「解」となる代入をととめることである。代入 S 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。この代入は「最も一般的」であるべきである。(定義の詳細はいまは省略)

7.1 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、普通の型推論で解ける。それを解いた結果、 $\alpha^0, \alpha^1, \gamma$ に対する代入が生じる (か、あるいは、「解なし」という結果になる)。

$\Delta \models t^0 \geq s^0$ 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。)

ここまでの段階で (代入がいくつか出たほか)、残る制約は、以下のものだけである。

- $\Delta \models \alpha^0 \geq \beta^0$
- $\Delta \models c \geq c$

ここまでにでてきた代入はすべて、上記の制約たちに適用済みとする。」(つまり、 $\alpha := \text{Int}$ という代入がでてきたら、上の式にある 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、上記の制約たちには、のこっていない。)

7.2 制約の解消アルゴリズム (後半)

$\Delta \models c \geq c$ の形の制約を解こう。($\Delta \models \alpha^0 \geq \beta^0$ については次の section で考えるので、ここでは無視する。)

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 compatible であるはずなので (ここはあとでチェックが必要) $\Delta = \Delta_1 \cup \dots \cup \Delta_n$ という風に全部を合体させた上で、 $\Delta \models c_i \geq c'_i$ を解けばよい。

(ステップ 1: classifier 変数の除去) 使われている classifier 変数の 1 つに着目する。(どれでもよい。) それを γ とする。 $c_i \geq \gamma$ の形の制約 ($i = 1, 2, \dots, I$) とを全部あつめる。 $\gamma \geq c'_j$ の形の制約 ($j = 1, 2, \dots, J$) と、これらを消去して、かわりに、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。(制約は一般には増えるかもしれない。) ステップ 1 を繰返すと、classifier 変数はなくなる。

(ステップ 2) $c_1 \cup c_2 \geq c_3$ を $c_1 \geq c_3$ と $c_2 \geq c_3$ に変換する。

これにより、不等号の左辺にある \cup の個数が 1 つ減る。ステップ 2 を繰返すと、不等号の左辺にある \cup はなくなる。

ステップ 2 の繰返しがおわると、制約は、 $\Delta \models d \geq c$ の形になる。

[oishi said](#): ステップ 3 は変更する必要あり

(ステップ 3) ここで「 d は atomic」 という仮定をおく。これについてはあとで吟味する。これは、「 $d \geq c_1 \cup c_2$ ならば $d \geq c_1$ または、 $d \geq c_2$ 」 という内容である。これらの逆向きは、いつでも成立するので、結局、ここの「ならば」は「同値」といってもよい。

これをもちいて、 $d \geq c$ の右辺も分解できて、

$$\Delta \models d_1 \geq d'_1 \vee \dots \vee d_n \geq d'_n$$

となり、さらに Δ も $d_1 \geq d_2$ の形を「かつ」と「または」でつないだ形になるはずである。

これは decidable なので、「解があるかどうか」も decidable である。

7.3 制約の解消アルゴリズム (後半のおまけ)

実は、まだ、 $\Delta \models \alpha^0 \geq \beta^0$ という制約がのこっていた。

これを最後まで残した (解かなかった) のは、以下の 2 つの可能性があるからである。これはどうするか？

- $\alpha^0 = \beta^0$
- $\alpha^0 = \langle t \rangle^{\gamma_1}, \beta^0 = \langle t \rangle^{\gamma_2}, \Delta \models \gamma_1 \geq \gamma_2$

第8章 型安全性の証明

8.1 型保存

8.2 進行

第9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 年の亀山らの研究 [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、研究対象となっていなかったが 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。

[oishi said: let-insertion の参考論文](#) : [Olivier Danvy 最初って言ってるやつ](#), [MetaOcaml : Gentle to introduction part1, part2](#) Taha, [varmeta Flops](#) : [oleg, lightweight-module scala](#) : [romps staging](#) のやつを参考文献に加える

第10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 (\cup) 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

ありがとうありがとう

参考文献

- [1] Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. *SIGPLAN Not.*, Vol. 46, No. 9, pp. 81–93, September 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, Yuki Yoshi Kameyama, and Yuto Sudo. Refined environment classifiers: Type- and scope-safe code generation with mutable cells. In *Proceedings of the 14th Asian Symposium on Programming Languages and Systems*, APLAS 2016, to appear.
- [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] Yuki Yoshi 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, *Programming Languages and Systems*, Vol. 7705 of *Lecture Notes in Computer Science*, pp. 296–311. Springer Berlin Heidelberg, 2012.