対称λ計算の基礎理論

阪上紗里　　浅井健一
お茶の水女子大学
sari@pllab.is.ocha.ac.jp
asai@is.ocha.ac.jp

概 要

「プログラムの残りの計算」を表す継続を扱う為の基礎言語体系として、対称 λ 計算 (Symmetric λ-calculus, SLC) が Filinski によって提案されている。SLC においては項と継続が完全に対称な形をしており、項を扱うのと同じように継続を扱うことができる。そのため、項と継続を統一的に議論するのに適していると思われるが、これまで SLC についての研究はほとんどなされていない。ここでは、まず SLC を small step semantics で定式化し直し、型付き言語の基本的な性質である Progress と Preservation を満たすことを証明した。次に、SLC が継続計算を議論・表現するのに適していることを示すため、(1) Felleisen の $\mathcal{C}$ オペレータを含む call-by-value 言語、Λ_C 計算、および (2) call-by-name $\lambda\mu$ 計算 (の変種) が、どちらも自然に SLC に変換できることを示す。近年 call-by-value と call-by-name の双対性が項と継続の対称性と絡めて注目されているが、ここでの結果はそれに対する洞察を与えるものと期待される。

1　はじめに

例外処理など、非局所的な制御を定式化するには「プログラムの残りの計算」を表す継続が使われる。近年、継続を明示的に扱えるようにして、複雑な制御をプログラムの中で記述できるようにした言語体系が多く提案されるようになって来た。

継続を議論するための枠組みには、CPS 変換や評価文脈を使った方法がある。CPS 変換は、継続に対する特殊な枠組を用意することなく継続を表現することができるが、CPS 変換は大域的な変換であるためプログラムの構造が大きく変化してしまう。一方、評価文脈を使えば、プログラムを大きく変換することなく継続についての議論をすることができるが、評価文脈という新しい概念を導入する必要がある。

項と継続を統一的に扱う理論としては、Filinski が既に 1989 年に対称 λ 計算 (Symmetric λ-calculus, SLC) を提案している [5]。SLC においては項と継続が完全に対称（双対）な形をしており、項を扱うのと同じように継続を扱うことができる。言わば評価文脈があらかじめ言語構文の中に組み込まれている格好である。そのため、項と継続を統一的に議論するのに適していると思われるが、これまで SLC についての研究はほとんどなされていない。そのひとつの原因は、項と継続が双対であるという概念自体は明快であるものの、SLC の定式化としては call-by-value 版の表示的意味記述が与えられているのみであり、簡約規則がないなど言語の基本的な挙動がわかりにくく、したがって継続を扱う他のコントロールオペレータに比べて扱いにくかったことなどが考えられる。

そこで、本稿ではまず SLC を small step semantics で定式化し直し、型付き言語の基本的な性質である Progress と Preservation を満たすことを証明する。次に、SLC が継続計算を議論・表現するのに適していることを示すため、(1) Felleisen の $\mathcal{C}$ オペレータを含む call-by-value 言語、Λ_C 計算 [4]、および (2) Parigot の call-by-name $\lambda\mu$ 計算 [6] (の変種) が、どちらも自然に SLC に変換できることを示す。

近年、Wadler の双対計算 [10] が示され、call-by-value と call-by-name の双対性が項と継続の対称性と絡めて注目されているが、ここでの結果はそれに対する洞察を与えるものと期待される。

本稿の構成は以下の通りである。次の節では（非決定的な）SLC を定式化し、型規則の健全性など基本的な性質を証明する。3 節では call-by-value SLC を導入し、4 節で Felleisen の Λ_C 計算が call-by-value SLC に変換できることを示す。5 節では call-by-name SLC を導入し、6 節で Parigot の $\lambda\mu$ 計算が call-by-name SLC に変換できることを示す。関連研究については 7 節で述べ、8 節でまとめる。なお、定理や補題の証明等の詳細については [8] を参照されたい。

2　対称 λ 計算 (SLC)

ここでは、本稿で扱う SLC の構文と簡約規則、型規則とその性質について述べる。SLC は計算を $\langle$継続$|$項$\rangle$ もしくは $\langle$継続$|$関数$|$項$\rangle$ の様に表わして示す。例えば「空の継続のもと、x に引数を受け取り $x+1$ を計算する関数に、3 を渡す」という計算は $\langle \bullet \,|\, x \Rightarrow x+1 \,|\, 3 \rangle$ と表される。これは x に 3 が渡されると項部分が $\langle \bullet \,|\, 4+1 \rangle = \langle \bullet \,|\, 4 \rangle$ と変換され、空の継続に渡された 4 が最終結果に出力されるようになる。

2.1　SLC の構文

λ 計算の構文は項（と値）からなるが、SLC においては項と継続を対称に扱っているため、その構文には項以外に継続が入ってくる。さらに、関数を独立した構文として用意しているところが特徴的である。

$$
\begin{array}{rrcl}
\text{(値)} & v & ::= & x \mid \lceil f \rceil \mid n \\
\text{(項)} & e & ::= & v \mid f \uparrow e \\
\text{(関数)} & f & ::= & x \Rightarrow e \mid y \Leftarrow c \mid \overline{e} \mid \underline{c} \\
\text{(継続)} & c & ::= & k \mid c \downarrow f \\
\text{(値継続)} & k & ::= & y \mid \lfloor f \rfloor \mid \bullet
\end{array}
$$

値 v は、変数 x であるか、関数 f を値として扱うときに使う $\lceil f \rceil$ であるか、整数 n である。項 e は、値 v であるか、関数 f を引数 e で呼び出す関数呼び出し $f \uparrow e$ である。関数呼び出しでは、引数部分は項でなくてはならないが、高階関数を使う場合など関数を引数として渡したいときには $\lceil \cdot \rceil$ を使う。例えば、λ 計算における $(\lambda x.\, x)\,(\lambda x.\, x)$ は $(x \Rightarrow x) \uparrow \lceil x \Rightarrow x \rceil$ と表される。

関数 f のところに現れる $x \Rightarrow e$ は x を受け取り e を返す関数で λ 計算の $\lambda x.\, e$ に相当する。また、$\overline{e}$ は、評価すると値になるような関数を表す。この $\overline{e}$ の挙動については、簡約規則の箇所にて詳しく述べる。

ここまでは通常の λ 計算とほぼ同じである。SLC では、これらに加えて、その双対である継続に関する構文が存在する。継続に関する構文は、項に関する構文の双対を（ほぼ）機械的に作ることで得ることができる。

値継続 k は、（継続を表す）変数 y であるか、関数 f を継続として扱うときに使う $\lfloor f \rfloor$ であるか、初期継続 $\bullet$ である。初期継続については、次の節で詳しく述べる。継続 c は、値継続 k であるか、関数 f を継続 c の前に呼び出す継続 $c \downarrow f$ である。項 $f \uparrow e$ は、項 e を関数 f に渡し、その結果をそのときの継続に渡すものであった。これは「項 e を、関数 f に通し、継続 c に渡す」という計算の流れのうち左側の「項 e を関数 f に通した結果得られる項」に相当する。一方、$c \downarrow f$ はその双対で、継続 c に結果が渡る前に関数 f を実行するような継続である。別の言い方をすると「項 e を、関数 f に通し、継続 c に渡す」という計算の流れのうち右側の「（項 e を受け取ったら）関数 f に通し、継続 c に渡すような継続」に相当する。

関数 f のところに現れる $y \Leftarrow c$ は、現在の継続を y に束縛した上で、現在の継続を c に取り替える関数である。項に関する関数 $x \Rightarrow e$ は「現在の項を x に束縛した上で、現在の項を e に取り替える関数」と思うことができる。$y \Leftarrow c$ は、その継続版で、ちょうど双対の関係にあることがわかる。$\underline{c}$ は、$\overline{e}$ の双対であり、評価すると値継続になるような関数を表す。

2.2 SLC の簡約規則

λ 計算では計算の状態は項のみで表現されるので、簡約規則は項の間の 2 項関係として定義される。一方、SLC の計算の状態は、継続も同時に考えるため、二つ組 $\langle c \,|\, e \rangle$ または三つ組 $\langle c \,|\, f \,|\, e \rangle$ という形で表される。ここで $\langle c \,|\, e \rangle$ は現在、計算しようとしている項が e で、そのときの継続が c であることを示す。(従来、評価文脈 E の中に項 M がある場合、$E[M]$ と記述してきたが、直感的にはこれが $\langle E \,|\, M \rangle$ に相当する。) また、$\langle c \,|\, f \,|\, e \rangle$ は現在、項 e を関数 f に渡そうとしており、そのときの継続が c であることを示している。(この説明は項に着目して書いているが、双対を考えれば継続に着目した説明も可能である。実際 $\langle c \,|\, f \,|\, e \rangle$ は、継続 c を関数 f で変換しようとしており、そのときの項が e であることも示している。) SLC の簡約はこのように定義される計算の状態の間の 2 項関係として定義される。

SLC の簡約規則は次のように与えられる。なお、この段階では簡約規則は非決定的である。call-by-value や call-by-name などの決定的な評価戦略については次節以降で述べる。これらの簡約規則においても、項に着目する規則と、継続に着目する規則の対称性を見ることができる。規則名において $\cdots$ が付いていないものが項に着目した簡約規則で、付いているものは継続に着目した簡約規則である。

$$
\begin{array}{rrcl}
(\mathit{begin}) & e & \rightsquigarrow & \langle \bullet \,|\, e \rangle \\
(\overline{\mathit{pop}}) & \langle c \,|\, f \uparrow e \rangle & \rightsquigarrow & \langle c \,|\, f \,|\, e \rangle \\
(\mathit{push}) & \langle c \,|\, f \,|\, e \rangle & \rightsquigarrow & \langle c \downarrow f \,|\, e \rangle \\
(\mathit{exchange}) & \langle c \,|\, \overline{e'} \,|\, e \rangle & \rightsquigarrow & \langle c \,|\, \lceil f_x \rceil \Rightarrow f_x \uparrow e \,|\, e' \rangle \\
(\beta) & \langle c \,|\, x \Rightarrow e' \,|\, e \rangle & \rightsquigarrow & \langle c \,|\, e'[e/x] \rangle \\
(\overline{\beta}) & \langle c \,|\, y \Leftarrow c' \,|\, e \rangle & \rightsquigarrow & \langle c'[c/y] \,|\, e \rangle \\
(\overline{\mathit{exchange}}) & \langle c \,|\, \underline{c'} \,|\, e \rangle & \rightsquigarrow & \langle c' \,|\, \lfloor f_y \rfloor \Leftarrow c \downarrow f_y \,|\, e \rangle \\
(\overline{\mathit{push}}) & \langle c \,|\, f \,|\, e \rangle & \rightsquigarrow & \langle c \,|\, f \uparrow e \rangle \\
(\mathit{pop}) & \langle c \downarrow f \,|\, e \rangle & \rightsquigarrow & \langle c \,|\, f \,|\, e \rangle \\
(\overline{\mathit{end}}) & \langle \bullet \,|\, v \rangle & \rightsquigarrow & v
\end{array}
$$

(begin) は計算を始める規則である。計算を始めるには、与えられた項 e を空の継続で実行するように、項 e を初期継続 $\bullet$ と組み合わせた $\langle \bullet \,|\, e \rangle$ という形に簡約する。計算の終了を示すのは $(\overline{\mathit{end}})$ である。項 e の実行が終了し $\langle \bullet \,|\, v \rangle$ という形になる、つまり、値 v が得られ、それが初期継続 $\bullet$ に渡されたら、計算結果 v が返ってくる。本稿では、全体の計算結果は常に 1 階の値である整数であると仮定している。(push) と (pop) およびそれらの双対である $(\overline{\mathit{push}})$ と $(\overline{\mathit{pop}})$ は、計算の焦点を変更する規則である。$(\overline{\mathit{pop}})$ は、現在の項が $f \uparrow e$ だったときに f と e に分解して三つ組に変換する。今後、f をさらに実行するか、e をさらに実行するか、あるいは (f が既に $x \Rightarrow e$ または $y \Leftarrow c$ という形だった場合に) 関数呼び出しを実行するかは評価戦略によって異なってくる。(push) は、焦点を e に移す規則である。また、(pop) と $(\overline{\mathit{push}})$ はこれらの継続版である。(β) は、通常の β 簡約の規則である。継続 c が不変の中、関数呼び出しが行われている。$e'[e/x]$ は e' の中の自由な x を e で置き換える (capture-avoiding な) 代入操作である。一方、$(\overline{\beta})$ はその双対で、継続の置き換えを行う規則である。項 e が不変の中、現在の継続 c を y に束縛した上で現在の継続が c' に変更されている。$(\mathit{exchange})$ は、関数部分に出てくる項 $\overline{e'}$ を右端に移動することで e' に焦点を当てる規則である。e' の実行が終了し、$\lceil f \rceil$ の形になったら、その結果を $\lceil f_x \rceil$ に受け取り、もとの引数 e を渡すような関数 $\lceil f_x \rceil \Rightarrow f_x \uparrow e$ を関数の部分に入れている。ここで $\lceil f_x \rceil \Rightarrow \cdots$ はパターンマッチであり、$\lceil f \rceil$ の形の値を受け取り、タグ $\lceil \cdot \rceil$ を外して $\cdots$ を実行する。もし項部分に $\lceil f \rceil$ 以外の値がきたらエラーとなる。$(\overline{\mathit{exchange}})$ は、その継続版である。ここで $\lfloor f_y \rfloor \Leftarrow \cdots$ も同様にパターンマッチであり、$\lfloor f \rfloor$ の形の値継続を受け取り、タグ $\lfloor \cdot \rfloor$ を外して $\cdots$ を実行させる関数を表す。ここで表記されている f_x や f_y は f と同じだが、項側と継続側のどちらから関数を受け取るかをわかりやすく明記するために、項側から関数を受け取る場合にはパターンマッチ部分を $\lceil f_x \rceil$ とし、継続から関数を受け取る場合にはパターンマッチ部分を $\lfloor f_y \rfloor$ と表記している。

通常、簡約関係を定めるときには、簡約規則以外に評価文脈の規則を導入する。例えば、λ 計算であれば β 簡約を $(\lambda x.\, e')\, e \rightsquigarrow e'[e/x]$ のように定義し、この規則をどのような評価文脈の中で用いても良いと定義する。SLC では、もともと継続を明示的に扱っているため、このような評価文脈に相当する規則はなく、すべての簡約規則において継続が明示されている。評価文脈に相当する規則は、焦点を変更する規則に埋め込まれている形になっている。

簡約規則を定義したところでひとつ例を挙げる。Scheme などに存在する call/cc は SLC では次のように表現できる [5]。

$$\mathsf{call/cc} = y \Leftarrow (y \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil \_ \Leftarrow y \rceil))$$

call/cc は、全体としては $y \Leftarrow \cdots$ という形をしていて、これは現在の継続を y に受け取るような関数である。この関数が、継続 c のもとで 1 引数関数（の項表現）$\lceil x \Rightarrow e \rceil$ を渡される（$\langle\, c \,|\, \mathsf{call/cc} \,|\, \lceil x \Rightarrow e \rceil \,\rangle$ を上の簡約規則にしたがって実行する）と、まず現在の継続 c が y に代入され $c \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil \_ \Leftarrow c \rceil)$ に置き換わる。次に、この式に $\lceil x \Rightarrow e \rceil$ が渡されるので、$\lceil f_x \rceil$ には $\lceil x \Rightarrow e \rceil$ がマッチする。よって f_x は $x \Rightarrow e$ になり、これに $\lceil \_ \Leftarrow c \rceil$ が渡されるので、結局 x には $\lceil \_ \Leftarrow c \rceil$ が入ることになる。ここで $\_$ はワイルドカードで、他の場所に現れない変数を示す。この関数（の項表現）$\lceil \_ \Leftarrow c \rceil$ が現在の継続を捨てて c にジャンプする関数である。呼び出されると、現在の継続が捨てられて c に置き換わるのである。以上の簡約の様子を以下に示す。e の中では x という変数で $\lceil \_ \Leftarrow c \rceil$ にアクセスできることがわかる。さらに、継続を捕捉した後も継続部分には c が残っているので、e の中で x を使うことなく評価が終了しても、その値は c に渡されることがわかる。

$$\begin{array}{lcll}
\langle\, c \,|\, \mathsf{call/cc} \,|\, \lceil x \Rightarrow e \rceil \,\rangle & = & \langle\, c \,|\, y \Leftarrow (y \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil _ \Leftarrow y \rceil)) \,|\, \lceil x \Rightarrow e \rceil \,\rangle & (\mathsf{call/cc}\ \text{の定義}) \\
& \rightsquigarrow & \langle\, c \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil _ \Leftarrow c \rceil) \,|\, \lceil x \Rightarrow e \rceil \,\rangle & (\overline{\beta}) \\
& \rightsquigarrow & \langle\, c \,|\, \lceil f_x \rceil \Rightarrow f_x \uparrow \lceil _ \Leftarrow c \rceil \,|\, \lceil x \Rightarrow e \rceil \,\rangle & (\mathit{pop}) \\
& \rightsquigarrow & \langle\, c \,|\, (x \Rightarrow e) \uparrow \lceil _ \Leftarrow c \rceil \,\rangle & (\beta) \\
& \rightsquigarrow & \langle\, c \,|\, x \Rightarrow e \,|\, \lceil _ \Leftarrow c \rceil \,\rangle & (\overline{\mathit{pop}}) \\
& \rightsquigarrow & \langle\, c \,|\, e[\lceil _ \Leftarrow c \rceil / x] \,\rangle & (\beta)
\end{array}$$

このように SLC では継続をふつうの項と同様に計算の対象とすることができる。

2.3 SLC の簡約規則の非決定性

上に示した簡約規則は、評価順序によって実行結果が変化する。例えば $\langle \bullet \downarrow (x \Rightarrow 2) \,|\, (y \Leftarrow \bullet) \uparrow 1 \,\rangle$ を考える。先に $(\overline{\mathit{pop}})$ を実行して $\langle \bullet \downarrow (x \Rightarrow 2) \,|\, y \Leftarrow \bullet \,|\, 1 \,\rangle$ としてから $(\overline{\beta})$ を実行すると $\langle \bullet \,|\, 1 \,\rangle$ となり 1 が返るが、先に (pop) を実行して $\langle \bullet \,|\, x \Rightarrow 2 \,|\, (y \Leftarrow \bullet) \uparrow 1 \,\rangle$ としてから (β) を実行すると $\langle \bullet \,|\, 2 \,\rangle$ となり 2 が返る。

これは SLC 自体が悪いわけではなく、実行の評価順序を定めていないために起こる問題である。実際、普通のプログラミング言語においても実行順序を定めていなければ結果は一通りには定まらない。例えば $(\lambda x.\, 1)\, (3/0)$ の結果は、call-by-value ならエラーであり call-by-name なら 1 になるのと同じことである。

SLC の実行順序については、後の節で具体的に触れる。その前に、以降の節では SLC の型規則とその性質について述べる。

2.4 SLC の型規則

SLC の構文が 3 種類あるのに対応して、SLC における型も 3 種類、存在する。SLC における型は次ページのように定義される。T は言わば「中立の」型である。$T \to T'$ は T から T' への項に関するクロージャの型であり、$T - T'$ はコンテキストの型 [5] である。$T - T'$ は $T \to T'$ の双対で $T \to T' = \neg(\neg T' \to \neg T)$

と定義される。

$$\begin{array}{rrcl} & T & ::= & T \to T' \mid T - T' \mid \mathsf{int} \\ \text{(項の型)} & T_e & ::= & +T \\ \text{(関数の型)} & T_f & ::= & \{^{+T \to +T'}_{\neg T' \to \neg T} \\ \text{(継続の型)} & T_c & ::= & \neg T \end{array}$$

これらの型を使って、項の型は $+$ を、継続の型は $\neg$ をつけて表現される。直感的な意味は、$+T$ は普通の T 型のことで、$\neg T$ は「T 型の値を受け取るような継続の型」を示す。$+$ はなくても構わないのだが、双対の関係を明示するためにつけることにしている。関数の型は $\{^{+T \to +T'}_{\neg T' \to \neg T}$ という形で表される。これで $+T$ 型の項を受け取って $+T'$ 型の項を返す関数の型を表すが、同時にその双対として $\neg T'$ 型の継続を受け取って $\neg T$ 型の継続を返す関数の型も表す。これらふたつの型は、実は見方が違うだけで同じ型を表している。項側から見た場合の $+T$ 型の項を受け取って $+T'$ 型の項を返す関数型をもつ f を考えてみよう。この関数は $+T$ 型の項 e を $+T'$ 型の項 $f \uparrow e$ に変換した上で T' 型の継続 c に結果を渡す。これは、継続側から見れば、T' 型を受け取る継続、つまり $\neg T'$ 型の継続 c を T 型を受け取る継続、つまり $\neg T$ 型の継続 $c \downarrow f$ に変換している。したがって、関数 f は $+T \to +T'$ という型と $\neg T' \to \neg T$ という型を併せ持っていることがわかる。これらふたつの型は、片方からもう片方を簡単に求めることができるので片方のみを書けば十分であるが、双対の関係を明示するために両方書くことにしている。

このような型の定義を使って、SLC の項、関数、継続の型規則は以下のように定義される。

$$\Gamma, x : +T_1 \vdash x : +T_1 \quad (\mathrm{TVar}) \qquad \Gamma, y : \neg T_2 \vdash y : \neg T_2 \quad (\overline{\mathrm{TVar}})$$

$$\frac{\Gamma \vdash f : \{^{+T_1 \to +T_2}_{\neg T_2 \to \neg T_1} \quad \Gamma \vdash e : +T_1}{\Gamma \vdash f \uparrow e : +T_2}\ (\mathrm{TApp}) \qquad \frac{\Gamma \vdash f : \{^{+T_1 \to +T_2}_{\neg T_2 \to \neg T_1} \quad \Gamma \vdash c : \neg T_2}{\Gamma \vdash c \downarrow f : \neg T_1}\ (\overline{\mathrm{TApp}})$$

$$\frac{\Gamma, x : +T_1 \vdash e : +T_2}{\Gamma \vdash x \Rightarrow e : \{^{+T_1 \to +T_2}_{\neg T_2 \to \neg T_1}}\ (\mathrm{TFun}) \qquad \frac{\Gamma, y : \neg T_2 \vdash c : \neg T_1}{\Gamma \vdash y \Leftarrow c : \{^{+T_1 \to +T_2}_{\neg T_2 \to \neg T_1}}\ (\overline{\mathrm{TFun}})$$

$$\frac{\Gamma \vdash f : \{^{+T_1 \to +T_2}_{\neg T_2 \to \neg T_1}}{\Gamma \vdash \lceil f \rceil : +(T_1 \to T_2)}\ (\mathrm{TFunClo}) \qquad \frac{\Gamma \vdash f : \{^{+T_1 \to +T_2}_{\neg T_2 \to \neg T_1}}{\Gamma \vdash \lfloor f \rfloor : \neg(T_1 - T_2)}\ (\overline{\mathrm{TFunClo}})$$

$$\frac{\Gamma \vdash e : +(T_1 \to T_2)}{\Gamma \vdash \overline{e} : \{^{+T_1 \to +T_2}_{\neg T_2 \to \neg T_1}}\ (\mathrm{TCloFun}) \qquad \frac{\Gamma \vdash c : \neg(T_1 - T_2)}{\Gamma \vdash \underline{c} : \{^{+T_1 \to +T_2}_{\neg T_2 \to \neg T_1}}\ (\overline{\mathrm{TCloFun}})$$

$$\Gamma \vdash n : +\mathsf{int} \quad (\mathrm{TInt}) \qquad \Gamma \vdash \bullet : \neg\mathsf{int} \quad (\overline{\mathrm{TInt}})$$

これらの型規則は、いずれも項と継続の双対性を念頭に置いて考えれば自然なものばかりである。特に (TApp) は関数を「項を変換するもの」と考えたときの関数適用の規則で、その双対の $(\overline{\mathrm{TApp}})$ は関数を「継続を変換するもの」と考えたときの関数適用の規則である。また、(TFunClo) 等の規則は、関数が項や継続としてとらえ直せることを示す規則である。最後の $(\overline{\mathrm{TInt}})$ の規則は、初期継続は整数を受け取る継続であるという規則である。

上記のような型規則を用いて、計算の状態に対する以下のような型規則を定義する。

$$\frac{\vdash c : \neg A \quad \vdash e : +A}{\vdash \langle\, c \,|\, e \,\rangle : +\mathsf{int}}\ (\mathrm{TProg1}) \qquad \frac{\vdash c : \neg B \quad \vdash f : \{^{+A \to +B}_{\neg B \to \neg A} \quad \vdash e : +A}{\vdash \langle\, c \,|\, f \,|\, e \,\rangle : +\mathsf{int}}\ (\mathrm{TProg2})$$

これらの規則は、状態の中の項、関数、継続が正しい型を持っていたら、全体の計算結果は整数になることを示している。計算結果が整数になるのは、初期継続が整数を受け取ると仮定しているためである。

このような型規則を定義すると、次のような Progress と Preservation の定理が（計算の非決定性や非合流性があるままで）成り立つ。

定理 2.1 (Progress) 1. $\langle\, c \,|\, f \,|\, e \,\rangle : +\mathsf{int}$ ならば、ある c', f', e' が存在して $\langle\, c \,|\, f \,|\, e \,\rangle \rightsquigarrow \langle\, c' \,|\, f' \,|\, e' \,\rangle$ または $\langle\, c \,|\, f \,|\, e \,\rangle \rightsquigarrow \langle\, c' \,|\, e' \,\rangle$ である。

2. $\langle c \,|\, e \rangle : +\mathsf{int}$ ならば、それは $\langle \bullet \,|\, v' \rangle$ という形であるか、ある c', f', e' が存在して $\langle c \,|\, e \rangle \rightsquigarrow \langle c' \,|\, f' \,|\, e' \rangle$ である。

定理 2.2 (Preservation) 1. $\langle c \,|\, f \,|\, e \rangle : +\mathsf{int}$ であるとき、$\langle c \,|\, f \,|\, e \rangle \rightsquigarrow \langle c' \,|\, f' \,|\, e' \rangle$ ならば $\langle c' \,|\, f' \,|\, e' \rangle : +\mathsf{int}$ であり、$\langle c \,|\, f \,|\, e \rangle \rightsquigarrow \langle c' \,|\, e' \rangle$ ならば $\langle c' \,|\, e' \rangle : +\mathsf{int}$ である。

2. $\langle c \,|\, e \rangle : +\mathsf{int}$ であるとき、$\langle c \,|\, e \rangle \rightsquigarrow \langle c' \,|\, f' \,|\, e' \rangle$ ならば $\langle c' \,|\, f' \,|\, e' \rangle : +\mathsf{int}$ である。

これらの定理は、いずれも簡約規則に関する簡単な場合分けで示すことができる。特に Progress は λ 計算の場合とは違い、型規則による帰納法を使わずに示すことができる。

以上が評価戦略を指定しない一般の SLC である。以下の節では、この SLC に制限を加える形で評価戦略を規定していく。

3 Call-by-value SLC

この節では、2 節で示した SLC の call-by-value 版を示す。

3.1 call-by-value SLC の構文

call-by-value SLC の構文を以下に示す。2 節に示した SLC の構文と基本的に同一だが、値に $[(\lfloor f_y \rfloor \Leftarrow c \downarrow f_y) \uparrow v]$ が加わり、値継続に $[c \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow v)]$ が加わっている点のみが異なっている。$[(\lfloor f_y \rfloor \Leftarrow c \downarrow f_y) \uparrow v]$ は、継続 c と値 v をパッケージ化したもので、Filinski がコンテキスト [5] と呼んでいるものである。直感的には $(\lfloor f_y \rfloor \Leftarrow c \downarrow f_y) \uparrow v$ と同じものだが、call-by-value の評価戦略に従わせるために一時的にこの関数適用を凍結するために使われる。値継続のコンテキストである $[c \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow v)]$ は call-by-value の戦略規則においては無くても問題はない。しかし値と値継続との対称性を保つ為に、両方表記することにする。後に説明する call-by-name SLC においては、この値のコンテキストと値継続のコンテキストの必要性が丁度逆になっている。コンテキストについて詳しくは次節で簡約規則とともに説明する。

$$\begin{array}{rrcl} \text{(値)} & v & ::= & x \mid \lceil f \rceil \mid n \mid [(\lfloor f_y \rfloor \Leftarrow c \downarrow f_y) \uparrow v] \\ \text{(値継続)} & k & ::= & y \mid \lfloor f \rfloor \mid \bullet \mid [c \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow v)] \end{array}$$

これらの構文に対する型規則は簡略にとらわれることがないので非決定性 SLC と同じ型規則を持つが、値と値継続にコンテキストが加わったので、それらに対する型規則 $(\mathrm{TContext_v})$ と $(\overline{\mathrm{TContext_v}})$ を新しく以下に与える。

$$\frac{\Gamma \vdash c : \neg B \quad \Gamma \vdash v : +A}{\Gamma \vdash [(\lfloor f_y \rfloor \Leftarrow c \downarrow f_y) \uparrow v] : +(A - B)}\ (\mathrm{TContext_v})$$

$$\frac{\Gamma \vdash c : \neg B \quad \Gamma \vdash v : +A}{\Gamma \vdash [c \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow v)] : \neg(A \to B)}\ (\overline{\mathrm{TContext_v}})$$

これらはコンテキスト自体と、その [ ] でくくられた内容はどちらも同じ型を持つので、今迄に示した他の型規則から導くことができる。

3.2 call-by-value SLC の簡約規則

call-by-value SLC の簡約規則を以下に示す。基本的には、2 節に示した SLC の簡約規則から非決定性を取り除いて、call-by-value にしたがって評価されるように変更したものである。規則名に $_{\mathrm{v}}$ がついているものは、2 節の簡約規則から変更されている。

$$
\begin{array}{rrcl}
(\mathit{begin}) & e & \rightsquigarrow & \langle \bullet \,|\, e \rangle \\
(\overline{\mathit{pop}}) & \langle c \,|\, f \uparrow e \rangle & \rightsquigarrow & \langle c \,|\, f \,|\, e \rangle \\
(\mathit{push}_{\mathrm{v}}) & \langle c \,|\, f \,|\, f' \uparrow e \rangle & \rightsquigarrow & \langle c \downarrow f \,|\, f' \uparrow e \rangle \\
(\mathit{context}_{\mathrm{v}}) & \langle [c \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow v)] \,|\, \lceil f \rceil \rangle & \rightsquigarrow & \langle c \,|\, f \,|\, v \rangle \\
(\mathit{exchange}_{\mathrm{v}}) & \langle c \,|\, \overline{e} \,|\, v \rangle & \rightsquigarrow & \langle [c \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow v)] \,|\, e \rangle \\
(\beta_{\mathrm{v}}) & \langle c \,|\, x \Rightarrow e \,|\, v \rangle & \rightsquigarrow & \langle c \,|\, e[v/x] \rangle \\
(\overline{\beta}_{\mathrm{v}}) & \langle c \,|\, y \Leftarrow c' \,|\, v \rangle & \rightsquigarrow & \langle c'[c/y] \,|\, v \rangle \\
(\overline{\mathit{exchange}}_{\mathrm{v}}) & \langle c \,|\, \underline{c'} \,|\, v \rangle & \rightsquigarrow & \langle c' \,|\, [(\lfloor f_y \rfloor \Leftarrow c \downarrow f_y) \uparrow v] \rangle \\
(\overline{\mathit{context}}_{\mathrm{v}}) & \langle \lfloor f \rfloor \,|\, [(\lfloor f_y \rfloor \Leftarrow c \downarrow f_y) \uparrow v] \rangle & \rightsquigarrow & \langle c \,|\, f \,|\, v \rangle \\
(\overline{\mathit{push}}) & \langle c \,|\, f \,|\, e \rangle & \rightsquigarrow & \langle c \,|\, f \uparrow e \rangle \\
(\mathit{pop}_{\mathrm{v}}) & \langle c \downarrow f \,|\, v \rangle & \rightsquigarrow & \langle c \,|\, f \,|\, v \rangle \\
(\overline{\mathit{end}}) & \langle \bullet \,|\, v \rangle & \rightsquigarrow & v
\end{array}
$$

call-by-value の評価戦略を端的に表している規則は (β_{v}) である。(β) とは違って、この規則は引数が値 v になっていなくては使うことができなくなっている。引数部分を値にするため、まず $(\overline{\mathit{pop}})$ を使って関数呼び出しを三つ組に分解する。三つ組の項部分がさらに別の関数呼び出しだった場合には、$(\mathit{push}_{\mathrm{v}})$ を使って関数 f を継続側に押しやり、$(\overline{\mathit{pop}})$ を使ってさらに実行を続けていく。$(\overline{\mathit{pop}})$ により項部分が値になったら、その際の関数部分の形によって実行が進む。関数部分がまだ関数の形（$x \Rightarrow e$ または $y \Leftarrow c$ の形）になっていない場合は、$(\mathit{exchange}_{\mathrm{v}})$ を使って関数部分を (項側で) 実行する。その関数部分を実行し $\lceil \cdot \rceil$ に評価されると、$(\mathit{context}_{\mathrm{v}})$ によって評価された関数を再び関数部分に戻し、三つ組にして実行を続ける。このようにして引数部分が値になり、関数部分も関数の形になって初めて β 簡約が行われる。$(\mathit{pop}_{\mathrm{v}})$ は、β 簡約の結果などで二つ組の項部分がすでに値になっていたときに、関数部分を継続から引き戻すのに使う。なお、ここに示した簡約規則は right-to-left の簡約規則、つまり関数部分より引数を先に評価する規則である。引数より関数部分を先に評価する left-to-right の簡約規則は、$(\mathit{push}_{\mathrm{v}})$ と $(\mathit{exchange}_{\mathrm{v}})$ および $(\overline{\mathit{exchange}}_{\mathrm{v}})$ の規則を以下の規則で置き換えれば良い。

$$
\begin{array}{rrcl}
(\mathit{push}'_{\mathrm{v}}) & \langle c \,|\, f_v \,|\, f' \uparrow e \rangle & \rightsquigarrow & \langle c \downarrow f_v \,|\, f' \uparrow e \rangle \\
(\mathit{exchange}'_{\mathrm{v}}) & \langle c \,|\, \overline{e'} \,|\, e \rangle & \rightsquigarrow & \langle [c \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow e)] \,|\, e' \rangle \\
(\overline{\mathit{exchange}'}_{\mathrm{v}}) & \langle c \,|\, \underline{c'} \,|\, e \rangle & \rightsquigarrow & \langle c' \,|\, [(\lfloor f_y \rfloor \Leftarrow c \downarrow f_y) \uparrow e] \rangle
\end{array}
$$

ここで f_v は $x \Rightarrow e$ または $y \Leftarrow c$ の形のことである。もとの right-to-left の規則では項が v になったときのみ $(\mathit{exchange}_{\mathrm{v}})$ が適用可能であったが、left-to-right の規則では $(\mathit{exchange}'_{\mathrm{v}})$ はいつでも適用可能なのに対し、$(\mathit{push}'_{\mathrm{v}})$ は関数部分が完全に評価されたときのみ適用可能となっている。この簡約規則の挙動の違いにより、 left-to-right の規則ではコンテキストの定義を変更し、中に項 e を含むようにする必要がある。$(\overline{\mathit{exchange}'}_{\mathrm{v}})$ は $(\mathit{exchange}'_{\mathrm{v}})$ の双対である。

call-by-value の評価戦略では項が先に実行されるため、項についての関数 $x \Rightarrow e$ を β 簡約するときには必ず引数部分は値になっている。しかし、その双対である $y \Leftarrow c'$ を β 簡約するときには、その引数である継続は値継続になっているとは限らない。したがって、$(\overline{\beta}_{\mathrm{v}})$ は（項部分は値になっているが）本質的には $(\overline{\beta})$ と同じである。同様のことが $(\overline{\mathit{exchange}}_{\mathrm{v}})$ についても言える。項部分を実行してみたが、関数部分が $\underline{c'}$ という形をしていたら、結局、この部分を実行せざるを得ない。よって、継続 c を一時的に項の方へ押しやり、c' の実行を開始している。ここで注目すべきことは、規則の右辺が非決定性 SLC の簡約規則である $(\overline{\mathit{exchange}})$ のように $\langle c' \,|\, \lfloor f_y \rfloor \Leftarrow c \downarrow f_y \,|\, v \rangle$ とはなっておらず、$\langle c' \,|\, [(\lfloor f_y \rfloor \Leftarrow c \downarrow f_y) \uparrow v] \rangle$ となっていることである。これは $\langle c' \,|\, \lfloor f_y \rfloor \Leftarrow c \downarrow f_y \,|\, v \rangle$ としてしまうと c' がまだ $\lfloor f \rfloor$ という形をしていない場合、継続部分を評価しなくてはいけないが、call-by-value の評価戦略では二つ組の項部分が値になって初めて継続部分に焦点があたる。そこで c や v を一時的に忘れて継続部分を先に評価するために関数部分

を項側におしやり、さらに値としてパッケージ化しておく。これによって継続部分に焦点があたり、c' が評価される。c' が $\lfloor f \rfloor$ という形に評価されたときに初めて $(\overline{context}_{\mathrm{v}})$ を使って $\langle\, c \,|\, f \,|\, v \,\rangle$ に戻すようになる。なお、$[(\lfloor f_y \rfloor \Leftarrow c \downarrow f_y) \uparrow v]$ は Filinski がコンテキストと呼んでいるものに対応する [5]。この双対の $(exchange_{\mathrm{v}})$ は実は call-by-value においてはコンテキストにする必要はないが、$(\overline{exchange}_{\mathrm{v}})$ の双対として表すためにコンテキストに変換する。

以上が call-by-value の評価規則である。ここには $(\overline{push})$ が現れていないが、call-by-value では常に項の部分が先に計算されるので、項の部分を保留して継続部分を評価にいく $(\overline{push})$ は不要である。

コンテキストが導入されているために、call-by-value SLC は 2 節の SLC の枠組を逸脱しているように思うかもしれないが、そうではない。コンテキストは、単に評価順序を規定するためのみに導入されており、call-by-value SLC の簡約は、すべて 2 節の SLC の簡約の一部になっている。

call-by-value の簡約規則についても、2 節の簡約規則とほぼ同様にして Progress と Preservation の性質が成り立つことを示すことができる。さらに、call-by-value の簡約規則の定義を調べると以下の命題が成り立つことが簡単にわかる。

命題 3.1 call-by-value SLC の簡約は一意的である。

評価戦略を規定すると、簡約の一意性が言える。簡約の一意性が示されると、状態の等価性を定義することができる。

定義 3.2 ふたつの状態 s_1 と s_2 （いずれも $\langle\, c \,|\, e \,\rangle$ または $\langle\, c \,|\, f \,|\, e \,\rangle$ という形）は、同じ値に簡約されるときに等価であるといい $s_1 \approx s_2$ と書く。

4 Λ_C 計算の埋め込み

call-by-value SLC は call-by-value の言語に継続を扱う命令が入った体系を議論・表現するのに適している。ここでは、その具体例として Felleisen による Λ_C 計算 [4] を call-by-value SLC に変換することができることを示す。

4.1 Λ_C 計算の構文

Felleisen の Λ_C 計算の構文を次に示す。Λ_C 計算とは、λ 計算に継続を扱うコントロールオペレータ $\mathcal{C}$ を追加したものである。ここでは call-by-value SLC の評価戦略に合わせて right-to-left のΛ_C 計算を扱うが、left-to-right でも同様の変換が可能である。$\mathcal{C}$ の意味については、次の節で簡約規則とともに述べる。

$$\begin{array}{llllll} \text{(値)} & V & ::= & x \mid \lambda x.\, M \mid \mathcal{C} & \text{(評価文脈)} \quad E & ::= \quad [\,] \mid E[F] \\ \text{(項)} & M & ::= & V \mid M\, M' & \text{(フレーム)} \quad F & ::= \quad M\,[\,] \mid [\,]\, V \end{array}$$

$E[F]$ とは評価文脈の中に、評価戦略を規定するフレームを伴うものである。フレームが $M\,[\,]$ のように引数部分が評価できる評価文脈になっている場合はまず引数部分を先に実行し、$[\,]\,V$ のように引数部分が値になってから関数部分が評価される挙動を示す。

4.2 Λ_C 計算の簡約規則

Λ_C 計算の簡約規則は次のふたつからなる。

$$\begin{array}{rcl} E[(\lambda x.\, M)\, V] & \rightsquigarrow & E[M[V/x]] \\ E[\mathcal{C}\, V] & \rightsquigarrow & V\,(\lambda x.\, \mathcal{A}\,(E[x])) \end{array}$$

最初の簡約規則は通常の call-by-value の β 簡約で、ふたつ目の簡約規則が $\mathcal{C}$ の動作を示す規則である。$\mathcal{C}$ に値 V （典型的には $\lambda k.\, M$ の形）が渡されると、次のふたつのことが行われる。まず第一に、そのときの評価文脈である $E[\ ]$ が $\lambda x.\, \mathcal{A}\,(E[x])$ という普通の関数の形に変換され V に渡される。V が $\lambda k.\, M$ という形をしていれば、以後、M の中で k という変数を通してその文脈を自由に使うことができるようになる。この k に渡される関数は、$\mathcal{C}\,V$ が実行されたときの継続を表す。実際、$\lambda x.\, \mathcal{A}\,(E[x])$ に引数を渡すと、それは $E[\ ]$ という文脈の中で実行される。$E[\ ]$ を囲んでいる $\mathcal{A}$ は Abort のことである。これは、そのときの文脈を捨てるコントロールオペレータであり、$\mathcal{A}\,M \stackrel{\text{def}}{=} \mathcal{C}\,(\lambda\_.\, M)$ と定義される。

$\mathcal{C}$ に値 V を渡したときに行われるもうひとつのことは、そのときの評価文脈の廃棄である。$E[\mathcal{C}\,V]$ の評価結果を見ると、全体が $E[\ ]$ では囲まれていない。これは V の中で文脈を明示的に使用しなかった場合、この文脈は捨てられることを意味している。これは、文脈を使用しなくても返って来た値が文脈に渡される call/cc とは対照的である。

4.3 call-by-value SLC への変換

ここでは、Λ_C 計算から call-by-value SLC への変換規則を示す。与えられた Λ_C 計算の項 M は、まず次の規則で call-by-value SLC に変換される。

$$T[\![M]\!] \quad = \quad \langle\, \bullet \,|\, T_e[\![M]\!] \,\rangle$$

つまり、M は $T_e[\![\cdot]\!]$ によって変換された結果を初期継続 $\bullet$ のもとで実行するような状態に変換される。$T_e[\![\cdot]\!]$ は、Λ_C 計算の項を call-by-value SLC の項に変換する関数である。これは、Λ_C 計算の項を call-by-value SLC の関数に変換する $T_f[\![\cdot]\!]$ とともに以下のように定義される。

$$\begin{array}{rcl@{\qquad}rcl}
T_e[\![x]\!] & = & x & T_f[\![x]\!] & = & \overline{x} \\
T_e[\![\lambda x.\, M]\!] & = & \lceil x \Rightarrow T_e[\![M]\!] \rceil & T_f[\![\lambda x.\, M]\!] & = & x \Rightarrow T_e[\![M]\!] \\
T_e[\![M\ N]\!] & = & T_f[\![M]\!] \uparrow T_e[\![N]\!] & T_f[\![M\ N]\!] & = & \overline{T_f[\![M]\!] \uparrow T_e[\![N]\!]} \\
T_e[\![\mathcal{C}]\!] & = & \lceil y \Leftarrow [\bullet \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil _ \Leftarrow y \rceil)] \rceil & T_f[\![\mathcal{C}]\!] & = & y \Leftarrow [\bullet \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil _ \Leftarrow y \rceil)]
\end{array}$$

変数 x、λ 抽象 $\lambda x.\, M$ については、対応する call-by-value SLC の項に変換しているだけである。関数呼び出し $M\ N$ についてもほぼそのままであり、関数部分 M を関数用の $T_f[\![\cdot]\!]$ で変換したものと、項部分 N を項用の $T_e[\![\cdot]\!]$ で変換したものを適用させた形になる。次の $T_e[\![\mathcal{C}]\!]$ が $\mathcal{C}$ の挙動を定義している規則である。$\mathcal{C}$ は「そのときの継続 y を値継続 $[\bullet \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil \_ \Leftarrow y \rceil)]$ で置き換える関数」として実現される。y を受け取った際に置き換わる継続は、値継続のコンテキストではなく $\bullet \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil \_ \Leftarrow y \rceil)$ としても call-by-value では問題はない。しかし、 3.1 節に示した構文にあわせるために $[\ ]$ でくくり、コンテキストにする。置き換わる継続値のコンテキストの内容を見ると、基本的には初期継続 $\bullet$ であるので現在の継続がリセットされていることがわかる。しかし、初期継続に値を渡す前に $(\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil \_ \Leftarrow y \rceil)$ が実行されるようになっている。この関数は、引数 $\lceil f_x \rceil$ を受け取ったら、そのタグ $\lceil \cdot \rceil$ をはずし、その関数 f_x に $\lceil \_ \Leftarrow y \rceil$ つまり「$\mathcal{C}$ が実行された時点での継続の関数表現」、より正確には「その時点での継続を捨てて、y に置き換えるような関数」を渡す。

$T_e[\![\mathcal{C}]\!]$ の定義は call/cc の定義にとても似ていることがわかる。実際、call/cc の定義は (値継続のコンテキストは call-by-value においては無くても構わないのでそのまま外すことができると思うと) $T_e[\![\mathcal{C}]\!]$ の $\bullet$ の部分が y になっているだけである。これは、call/cc がそのときの継続を複製した上で引数に渡すのに対し、$\mathcal{C}$ はそのときの継続を取り除いて引数に渡しているためである。SLC による定義は両者の特徴をよく表していることがわかる。

$T_e[\![M\ N]\!]$ の定義の中で使われている $T_f[\![M]\!]$ は M を関数として変換するために、$T_e[\![M]\!]$ の変換において $\lceil \cdot \rceil$ のつく λ 抽象 $\lambda x.\, M$ および $\mathcal{C}$ の変換は、$T_e[\![\cdot]\!]$ の $\lceil \cdot \rceil$ を外した形となり、それ以外のタグが付かないものに関しては $\bar{\cdot}$ を付加した形というだけで、ほぼ $T_e[\![M]\!]$ と同じ変換内容になっている。

以上で Λ_C の項 M を call-by-value SLC に変換することができた。しかし、この変換が簡約規則を保存することを示そうと思うと、単に M を $\langle \bullet \,|\, T_e[\![M]\!] \rangle$ の形に変換するだけでなく、任意の評価文脈 $E[\ ]$ の中にある項 M を上手に call-by-value SLC の状態に変換する必要が出てくる。そこで、上の $T_e[\![\cdot]\!]$ と $T_f[\![\cdot]\!]$ に加えて、次の $T_c[\![\cdot]\!]$ を（$T_F[\![\cdot]\!]$ を使って）定義する。

$$
\begin{array}{rcl}
T_c[\![[\ ]]\!] & = & \bullet \\
T_c[\![E[F]]\!] & = & (F = M\ [\ ] \text{ の場合})\ T_c[\![E]\!] \downarrow T_F[\![F]\!] \\
 & & (F = [\ ]\ V \text{ の場合})\ [T_c[\![E]\!] \downarrow (T_F[\![F]\!])]
\end{array}
\qquad
\begin{array}{rcl}
T_F[\![M\ [\ ]]\!] & = & T_f[\![M]\!] \\
T_F[\![[\ ]\ V]\!] & = & \lceil f_x \rceil \Rightarrow (f_x \uparrow T_e[\![V]\!])
\end{array}
$$

これらは、評価文脈とフレームをそれぞれ call-by-value SLC の継続と関数に変換する規則である。空の評価文脈は初期継続 $\bullet$ に変換され、フレームの列は初期継続の前に実行する関数の列になる。フレームは対応する関数に変換されるが、right-to-left の評価戦略を用いているため $[\ ]\ V$ の場合は V を後で適用するような関数に変換される。この場合変換の全体は $[\ ]$ でくくられコンテキストとなる。call-by-value においてはコンテキストにしなくても良いが、call-by-value SLC の構文に合わせるためにこのようにしている。後に出てくる $\lambda\mu$ 計算から call-by-name SLC への変換では、常にコンテキストを伴った変換が必要となってくる。

このように定義をすると、次の重要な補題を評価文脈 E の構造に関する帰納法により示すことができる。

補題 4.1 $\langle \bullet \,|\, T_e[\![E[M]]\!] \rangle \approx \langle T_c[\![E]\!] \,|\, T_e[\![M]\!] \rangle$

この補題は、評価文脈 $E[\ ]$ と項 M が SLC の世界では継続と項にきれいに分離できることを示している。この補題を使うと、これまでの $E[(\lambda x.\, M)\, V] \rightsquigarrow E[M[V/x]]$ といった簡約規則は、評価文脈 $E[\ ]$ を継続部分に分離することで従来と同じく $(\lambda x.\, M)\, V \rightsquigarrow M[V/x]$ の部分に着目した議論をすることができるとともに、$E[\mathcal{C}\, V] \rightsquigarrow V\, (\lambda x.\, \mathcal{A}\, (E[x]))$ のようにコンテキストを巻き込んだ規則についても議論ができるようになる。

なお、上の補題は M が値 V の場合にはより強い $\langle \bullet \,|\, T_e[\![E[V]]\!] \rangle \rightsquigarrow^* \langle T_c[\![E]\!] \,|\, T_e[\![V]\!] \rangle$ という関係が成り立つことを証明できる。しかし、M が値でないときには途中で逆の簡約をしなければならないことがあるため、この強い関係は値以外の M については一般には成り立たない。

上の補題を使うと、$T[\![\cdot]\!]$ による変換で Λ_C の簡約規則の等価性が保存されていることを示すことができる。

定理 4.2 Λ_C 計算の式 M について $M \rightsquigarrow N$ が成り立ち、かつ $\vdash T[\![M]\!] : +\mathsf{int}$ なら、$T[\![M]\!] \approx T[\![N]\!]$ が成り立つ。

証明の概略　$M \rightsquigarrow N$ についての場合分けを行う。$E[(\lambda x.\, M)\, V] \rightsquigarrow E[M[V/x]]$ の場合は、ほぼ従来通りの証明を行うことができる。補題 4.1 により評価文脈 $E[\ ]$ は継続部分に分離できるので、$\langle T_c[\![E]\!] \,|\, T_e[\![(\lambda x.\, M)\, V]\!] \rangle$ と $\langle T_c[\![E]\!] \,|\, T_e[\![M[V/x]]\!] \rangle$ が同じ値になることを言えば良い。それには $T_e[\![(\lambda x.\, M)\, V]\!]$ の定義を展開して、簡約をした上で、次の代入補題を使う。

補題 4.3 $T_e[\![M]\!][T_e[\![V]\!]/x] = T_e[\![M[V/x]]\!]$

この代入補題自身は、M の構造に関する帰納法より証明できる。

一方、$E[\mathcal{C}\, V] \rightsquigarrow V\, (\lambda x.\, \mathcal{A}\, (E[x]))$ の場合は、多少、複雑になる。まず、簡約前の項と後の項をそれぞれ $T[\![\cdot]\!]$ で変換し、補題 4.1 を使いながら簡約する。すると、両者、似たような格好の状態になるが、一部、変数に代入される値が異なった格好になる。したがって、この格好の異なったふたつの値があらゆる文脈の中で等しく振る舞うことを示さなくてはならない。

通常の λ 計算であれば β 等価性があるので、簡単にそのような議論を行うことができるが、SLC では等価性の概念は定義されておらず、あるのは簡約規則だけである。ふたつの値があらゆる文脈の中で等し

く振る舞うことを示すため、我々は住井らの方法 [9] にならって SLC に対する論理関係を型に関する帰納法で定義し、論理関係の基本定理を証明した。紙面の都合上、詳細には述べないが、それを使うと、そこから文脈等価性を導くことができ、ふたつの値の等価性を証明することができる。証明の詳細は [8] にて示している。 □

上の定理は Λ_C の世界で簡約をしても変換した call-by-value SLC の世界で型がついていたら、$\approx$ である関係が崩れないことを示している。この定理から次の系を導くことができる。

系 4.4 Λ_C 計算の式 M について $M \rightsquigarrow^* V$ が成り立つなら、$T[\![M]\!] \rightsquigarrow^* T[\![V]\!]$ が成り立つ。

これらの結果によって Λ_C における計算をある程度、call-by-value SLC で模倣できていることを示していると考えられるが、模倣関係として典型的に思い浮かべるのは以下のようなより強い関係である。

Λ_C 計算の式 M について $M \rightsquigarrow N$ が成り立つなら、$T[\![M]\!] \rightsquigarrow^* T[\![N]\!]$ が成り立つ。

しかし、この関係は現在の定式化では成り立たない。$T[\![M]\!]$ を実行しても $T[\![N]\!]$ そのものにはならず、$T[\![N]\!]$ を何段階か実行したあとのものにしかならないのである。これは Plotkin が CPS 変換の模倣性を示すときの状況 [7] と似ている。CPS 変換前の簡約は CPS 変換後の簡約にちょうどは対応しておらず、administrative redex を上手に簡約した後の項にしか対応しないのである。この問題は変換を工夫することによってきれいな形で Danvy と Filinski が解決している [2]。同様の方法が我々の場合にも使えるかどうかは今後の課題である。

本稿で示した変換は Λ_C 計算の等価性を粗くモデルしている可能性は残っている。つまり Λ_C 計算では異なる二つの項が同じ SLC の項に変換されている可能性はある。これを排除するためには変換の単射性を示す必要がある。単射を示すには、$T[\![M]\!] \approx T[\![N]\!]$ となるような Λ_C 計算の式 M, N があるならば、β 簡約と $\mathcal{C}$ の挙動を表す簡約規則において $M = N$ であることを証明すればよい。これも今後の課題である。

5 Call-by-name SLC

この節では、2 節で示した SLC の call-by-name 版を示す。ここに示す体系は 3 節で示した体系の値と継続を丁度逆にした格好になっている。

5.1 call-by-name SLC の構文

call-by-name SLC の構文を以下に示す。3.1 節では値と値継続にコンテキストが加わっていたが、call-by-name においても値と値継続にコンテキストが加わり、それぞれ $[k \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow e)]$ と $[k \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow e)]$ となる。call-by-name におけるコンテキストは、値継続 k と (値 v ではなく) 項 e をパッケージ化したものである。その用途は call-by-value の場合と同じである。

$$\begin{array}{rrcl} \text{(値)} & v & ::= & x \mid \lceil f \rceil \mid n \mid [(\lfloor f_y \rfloor \Leftarrow k \downarrow f_y) \uparrow e] \\ \text{(値継続)} & k & ::= & y \mid \lfloor f \rfloor \mid \bullet \mid [k \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow e)] \end{array}$$

5.2 call-by-name SLC の簡約規則

call-by-name SLC の簡約規則を次ページに示す。call-by-value の場合と同様に call-by-name に特有の規則には $_{\mathrm{n}}$ をつけてある。これらの規則は (begin) と $(\overline{\mathit{end}})$ を除けば、いずれも call-by-value の規則の項と継続を機械的に入れ換えることで得ることができる。例えば、$(\overline{\mathit{pop}}_{\mathrm{n}})$ は継続が値継続 k に評価されたら項から関数を呼び戻す規則だが、これは call-by-value の $(\mathit{pop}_{\mathrm{v}})$ において項が値 v に評価されたら継続か

ら関数を呼び戻しているのに対応する。call-by-name のそれ以降の規則もすべて順に call-by-value の規則の下から順に対応していることがわかる。そして call-by-name では常に継続部分が先に計算されるので、継続部分を保留して項部分を評価にいく ($push$) は不要である。これも call-by-value において ($\overline{push}$) が使われなかったこととの双対となっていることがわかる。

$$
\begin{array}{rrcl}
(begin) & e & \rightsquigarrow & \langle \bullet \,|\, e \rangle \\
(\overline{pop}_{\mathrm{n}}) & \langle k \,|\, f \uparrow e \rangle & \rightsquigarrow & \langle k \,|\, f \,|\, e \rangle \\
(push) & \langle c \,|\, f \,|\, e \rangle & \rightsquigarrow & \langle c \downarrow f \,|\, e \rangle \\
(context_{\mathrm{n}}) & \langle [k \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow e)] \,|\, \lceil f \rceil \rangle & \rightsquigarrow & \langle k \,|\, f \,|\, e \rangle \\
(exchange_{\mathrm{n}}) & \langle k \,|\, \overline{e'} \,|\, e \rangle & \rightsquigarrow & \langle [k \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow e)] \,|\, e' \rangle \\
(\beta_{\mathrm{n}}) & \langle k \,|\, x \Rightarrow e' \,|\, e \rangle & \rightsquigarrow & \langle k \,|\, e'[e/x] \rangle \\
(\overline{\beta}_{\mathrm{n}}) & \langle k \,|\, y \Leftarrow c \,|\, e \rangle & \rightsquigarrow & \langle c[k/y] \,|\, e \rangle \\
(\overline{exchange}_{\mathrm{n}}) & \langle k \,|\, \underline{c} \,|\, e \rangle & \rightsquigarrow & \langle c \,|\, [(\lfloor f_y \rfloor \Leftarrow k \downarrow f_y) \uparrow e] \rangle \\
(\overline{context}_{\mathrm{n}}) & \langle \lfloor f \rfloor \,|\, [(\lfloor f_y \rfloor \Leftarrow k \downarrow f_y) \uparrow e] \rangle & \rightsquigarrow & \langle k \,|\, f \,|\, e \rangle \\
(\overline{push}_{\mathrm{n}}) & \langle c \downarrow f' \,|\, f \,|\, e \rangle & \rightsquigarrow & \langle c \downarrow f' \,|\, f \uparrow e \rangle \\
(pop) & \langle c \downarrow f \,|\, e \rangle & \rightsquigarrow & \langle c \,|\, f \,|\, e \rangle \\
(\overline{end}) & \langle \bullet \,|\, v \rangle & \rightsquigarrow & v
\end{array}
$$

項と継続を入れ換えると call-by-value の簡約規則が call-by-name の簡約規則になることは、次のように理解できる。call-by-name では、必要な最小限の計算のみを行って答えを返す。この「答えの必要性」「答えを受け取るもの」とはまさに継続に他ならない。つまり、初期継続に始まり、答えを受け取るために必要なものを eager に求めていくことが必要最小限の計算のみを行うことに対応しているのである。

call-by-value と call-by-name の簡約規則が双対の関係にあることがわかると、いろいろと面白いことがわかってくる。例えば、call-by-value では right-to-left と left-to-right の評価戦略が考えられたが、今まで call-by-name では（引数は実行しないので）評価戦略は「関数部分（のみ）を先に実行する」というものしかなかった。しかし、双対の関係を調べてみると、call-by-name の世界でも継続の関数（$y \Leftarrow c$）の実行についてはふたつの評価戦略が考えられることがわかる。

また、call-by-value の関数 $x \Rightarrow e$ の双対は $y \Leftarrow c$ なので、call-by-name における関数 $x \Rightarrow e$ とは双対の関係にはなっていないこともわかる。このことは、Wadler の双対計算において、and と or はきれいに対応しているが関数については必ずしもきれいに対応がとれていないことと関係していると思われる。

call-by-value と call-by-name の双対性については、これまでもいろいろなところで言及されて来たが、ほとんどのものは古典論理との関係によるもので必ずしも普通の λ 計算を素直に中に含む形では示されてこなかった。本稿は、λ 計算を自然に含む SLC の枠組みの中で双対性を厳密に示している点で意義があると考えられる。

call-by-name の簡約規則についても Progress と Preservation の性質が成り立つことを示すことができる。また、call-by-value の場合と同様に簡約の一意性も簡単に示すことができる。

命題 5.1 call-by-name SLC の簡約は一意的である。

6 $\lambda\mu$ 計算の埋め込み

ここでは、Call-by-name SLC への埋め込みの例として、Parigot の $\lambda\mu$ 計算 [6] (の変種) を考える。

6.1 $\lambda\mu$ 計算の構文

$\lambda\mu$ 計算の構文を次に示す。$\lambda\mu$ 計算とは、call-by-name の λ 計算に μ 変数 α、μ 抽象 $\mu\alpha.M$、そして

名前付き項 $[\alpha]\,M$ を追加したものである。これらの意味については、次の節で簡約規則とともに述べる。

$$\begin{array}{rrcl} \text{(値)} & V & ::= & \lambda x.\,M \\ \text{(項)} & M & ::= & x \mid M\,M' \mid V \mid \alpha \mid \mu\alpha.\,M \mid [\alpha]\,M \\ \text{(評価文脈)} & E & ::= & [\,] \mid E[F] \\ \text{(フレーム)} & F & ::= & [\,]\,M \end{array}$$

call-by-name の $E[F]$ の評価文脈中に現れるフレームは一種類のみである。フレーム $[\,]\,M$ は関数部分から評価されることを表している。

6.2 $\lambda\mu$ 計算の簡約規則

本稿で考える $\lambda\mu$ 計算の簡約規則は次の 4 つからなる。

$$\begin{array}{rcll} E[(\lambda x.\,M)\,N] & \rightsquigarrow & E[M[N/x]] & \\ E[(\mu\alpha.\,M)\,N] & \rightsquigarrow & E[\mu\alpha'.\,M[\lambda x.\,\alpha'\,(x\,N)/\alpha]] & (x, \alpha' : \text{fresh}) \\ E[\mu\alpha.\,[\alpha]\,M] & \rightsquigarrow & E[M] & (\alpha \text{ は } M \text{ に現れない}) \\ \mu\alpha.\,M & \rightsquigarrow & M[\lambda x.\,\mathcal{A}\,x/\alpha] & \end{array}$$

最初の簡約規則は通常の call-by-name の β 簡約である。次の簡約規則が μ 抽象他の動作を示す規則である。μ 抽象 $\mu\alpha.\,M$ は、直感的にはそのときの継続を取って来て α に束縛し、M を実行する。しかし、$\lambda\mu$ 計算では普通、簡約規則を局所的なものにするため、そのときの継続全体を一度に取ってくるのではなく直近の継続を順にひとつずつ取ってくる形で定式化される。μ 抽象が $(\mu\alpha.\,M)\,N$ のように関数呼び出しの関数部分に現れたとしよう。その時点での $\mu\alpha.\,M$ の直近の継続は $[\,]\,N$ であり、さらにその後 $(\mu\alpha.\,M)\,N$ の継続が続くはずである。$(\mu\alpha.\,M)\,N$ の継続を α' とすると、結局、$\mu\alpha.\,M$ の継続は全体として $\alpha'\,([\,]\,N)$ となる。これを、通常の関数にした $\lambda x.\,\alpha'\,(x\,N)$ を α に束縛すればよいので、簡約結果は $\mu\alpha'.\,M[\lambda x.\,\alpha'\,(x\,N)/\alpha]$ となる。全体を $\mu\alpha'.$ でくくることで、$(\mu\alpha.\,M)\,N$ の継続（$= E[\,]$）を α' に取って来ている。

普通の $\lambda\mu$ 計算では、上に示したような通常の代入ではなく $[\alpha]\,M$ を $\alpha'\,(M\,N)$ に直接、置き換えるような特殊な代入が使われる。これは $\lambda\mu$ 計算のいろいろな理論的な性質を示すにあたっての技術的な問題を解決するためだが、本稿では上のような通常の代入で十分である。したがって、本稿では名前付き項 $[\alpha]\,M$ を通常の関数呼び出し $\alpha\,M$ と同一視し、通常の代入を使うことにする。

3 つ目の規則は、η 簡約に相当する規則である。現在の継続を取得したが、その継続にそのまま M を渡すだけであれば、初めから単に M と書くのと同じであることを示している。

Parigot の μ 抽象の動作は以上の通りであるが、これだけだとプログラミング言語としては使いにくい。というのは、上の簡約規則だと μ 抽象が（うまく 3 つ目の規則で消えない限り）簡約結果に残るので、一度 μ 抽象が実行されるとその μ 抽象は外側に波及していき、いずれトップレベルに到達する。その時点で、簡約できる規則がなくなり μ 抽象自体が計算結果となってしまう。しかし、μ 抽象の意図がそのときの継続を取って来て α に束縛することだと解釈すると、μ 抽象がトップレベルに現れたらそのときの継続（＝計算を終了する）を α に束縛してさらに実行を続けて欲しいと考えられる。このプログラム上の簡約を行っているのが最後の簡約規則である。この規則は μ 抽象 $\mu\alpha.\,M$ がトップレベルに現れたら α を「実行されたら計算を終了する」ような継続に束縛して μ 抽象の中身 M の実行を続けるようになっている。

6.3 call-by-name SLC への埋め込み

ここでは、$\lambda\mu$ 計算から call-by-value SLC への変換規則を示す。call-by-value の場合と同様に、与えられた $\lambda\mu$ 計算の項 M は、まず次の規則で call-by-name SLC に変換される。

$$T[\![M]\!] \;=\; \langle\, \bullet \,|\, T_e[\![M]\!] \,\rangle$$

$T_e[\![\cdot]\!]$ とそこで使われる $T_f[\![\cdot]\!]$、および後に使う $T_c[\![\cdot]\!]$ とそこで使われる $T_F[\![\cdot]\!]$ の定義は以下の通りである。

$$
\begin{array}{rcl}
T_e[\![x]\!] & = & x \\
T_e[\![\lambda x.\, M]\!] & = & \lceil x \Rightarrow T_e[\![M]\!] \rceil \\
T_e[\![M\, N]\!] & = & T_f[\![M]\!] \uparrow T_e[\![N]\!] \\
T_e[\![\alpha]\!] & = & \alpha \\
T_e[\![\mu\alpha.\, M]\!] & = & (y \Leftarrow [\bullet \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil \_ \Leftarrow y \rceil)]) \uparrow (\lceil \alpha \Rightarrow T_e[\![M]\!] \rceil) \\
T_e[\![[\alpha]\, M]\!] & = & \overline{\alpha} \uparrow T_e[\![M]\!] \\
 & & \\
T_f[\![x]\!] & = & \overline{x} \\
T_f[\![\lambda x.\, M]\!] & = & x \Rightarrow T_e[\![M]\!] \\
T_f[\![M\, N]\!] & = & \overline{T_f[\![M]\!] \uparrow T_e[\![N]\!]} \\
T_f[\![\alpha]\!] & = & \overline{\alpha} \\
T_f[\![\mu\alpha.\, M]\!] & = & \overline{(y \Leftarrow [\bullet \downarrow (\lceil f_x \rceil \Rightarrow f_x \uparrow \lceil \_ \Leftarrow y \rceil)]) \uparrow (\lceil \alpha \Rightarrow T_e[\![M]\!] \rceil)} \\
T_f[\![[\alpha]\, M]\!] & = & \overline{\overline{\alpha} \uparrow T_e[\![M]\!]} \\
 & & \\
T_c[\![[\,]]\!] & = & \bullet \\
T_c[\![E[F]]\!] & = & [T_c[\![E]\!] \downarrow (T_F[\![F]\!])] \\
 & & \\
T_F[\![[\,]\, M]\!] & = & \lceil f_x \rceil \Rightarrow (f_x \uparrow T_e[\![M]\!])
\end{array}
$$

μ 変数 α は、SLC では普通の項の変数と解釈される。μ 抽象 $\mu\alpha.\, M$ の変換結果は複雑な形をしているが、これは Λ_C 計算のところで示した $\mathcal{C}$ の定義を使うと $T_e[\![\mathcal{C}\,(\lambda\alpha.\, M)]\!]$ と書き表すことができる。つまり、μ 抽象は call-by-name で解釈する $\mathcal{C}$ と同じ動作となるのである。名前付き項 $[\alpha]\, M$ は単に普通の関数呼び出しと同じように変換される。$T_f[\![\cdot]\!]$ の定義は call-by-value のときと同じように対応する SLC の関数への変換規則であり $T_e[\![\cdot]\!]$ とほぼ同じである。$T_c[\![E[F]]\!]$ は少し注意が必要である。評価順序を制御するために常に $[\ldots]$ でパッケージ化する必要がある。

以上の定義を使うと call-by-value の場合と同様にして、評価文脈 $E\,[\,]$ の構造に関する帰納法により、以下の補題を示すことができる。

補題 6.1 $\langle \bullet \,|\, T_e[\![E[M]]\!] \rangle \approx \langle T_c[\![E]\!] \,|\, T_e[\![M]\!] \rangle$

さらに、この補題を使って次の定理を示すことができる。

定理 6.2 $\lambda\mu$ 計算の式 M について $M \rightsquigarrow N$ が成り立つなら、$T[\![M]\!] \approx T[\![N]\!]$ が成り立つ。

call-by-value の場合と同様、この定理の証明には次の代入補題と論理関係を使った議論が必要である。

補題 6.3 $T_e[\![M]\!][T_e[\![N]\!]/x] = T_e[\![M[N/x]]\!]$

7 関連研究

SLC は 1989 年に Filinski [5] によって提案された。Filinski は圏論に基づいて SLC を導入し、その call-by-value 版の表示的意味記述を示している。本稿で示している SLC はそれに基づいて作られており、特に本稿の Call-by-value SLC は Filinski のものと同一であると考えているが、現在のところ厳密な対応関係は示されていない。

項と継続、そして call-by-value と call-by-name の双対性については、Curien と Herbelin が $\overline{\lambda}\mu\tilde{\mu}$ 計算の中で述べている [1]。この仕事は、さらに Wadler の双対計算 [10] へと発展している。しかし、これらの仕事で使われている関数呼び出しは古典論理に関する考察から導かれたもので、通常の関数呼び出しとは異なる形をしている。一方、SLC には自然な形で λ 計算が埋め込まれている。

Felleisen の $\mathcal{C}$ オペレータ（を call-by-name で解釈したもの）と$\lambda\mu$ 計算が同等であることは de Groote によって示されている [3]。本稿は、同様の結果を両者を SLC に埋め込むことで得たものとなっている。

8 まとめと今後の課題

本稿では、Filinski による対称 λ 計算 SLC を small step semantics で定式化し直し、型に関する基本的な性質を証明するとともに、Felleisen の Λ_C 計算、Parigot の $\lambda\mu$ 計算がそれぞれ call-by-value、call-by-name の SLC に埋め込めることを示した。SLC が提案されたのはかなり前であるが、SLC についての研究は始まったばかりという感じである。本稿は SLC の最も基礎的な部分を定式化し、ふたつの継続を扱う体系を埋め込むことで、SLC がいろいろな継続計算に関する議論を行う際の土台となることを示している。

今後の方向性としては、いろいろなものが考えられる。まず第一に考えられるのは、Wadler の双対計算との関係の解明である。Filinski の提案した SLC では直積と直和が既に入っているので、そこまで合わせて考えれば容易に双対計算との関係がわかってくると期待される。また、それを通して SLC と論理学との関係を追求するのも興味深いテーマである。関連して、Wadler は双対計算と $\lambda\mu$ 計算の間に対応関係があることを示している [11] が、それに対する洞察が得られることも期待される。

本稿で示した埋め込みはもとの言語の型を考慮に入れていないが、Λ_C 計算にも $\lambda\mu$ 計算にもそれぞれ型システムがある。本稿で考えた埋め込みが型的にどうなっているのかを解明するのも面白い。さらに大きなテーマとして、本稿では非限定継続のみを扱って来たが、shift/reset などの限定継続がどのように組み込めるかも検討してみたい。

謝辞 筑波大学の亀山幸義先生には、$\lambda\mu$ 計算を始めとして多くのことについて教えて頂きました。ここに記して感謝いたします。本研究は科研費 (18500005) の助成を受けたものである。

参考文献

[1] Curien, P.-L., and H. Herbelin “The Duality of Computation,” *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP’00)*, pp. 233–243 (September 2000).

[2] Danvy, O., and A. Filinski “Representing Control, a Study of the CPS Transformation,” *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. 361–391 (December 1992).

[3] de Groote, P. “On the Relation between the $\lambda\mu$-Calculus and the Syntactic Theory of Sequential Control,” In F. Pfenning, editor, *Logic Programming and Automated Reasoning (LNCS 822)*, pp. 31–43 (July 1994).

[4] Felleisen, M., and R. Hieb “The Revised Report on the Syntactic Theories of Sequential Control and State,” *Theoretical Computer Science*, Vol. 103, No. 2, pp. 235–271 (September 1992).

[5] Filinski, A. “Declarative Continuations and Categorical Duality,” Master’s thesis, DIKU Report 89/11, University of Copenhagen (August 1989).

[6] Parigot, M. “$\lambda\mu$-calculus: An Algorithmic Interpretation of Classical Natural Deduction,” In A. Voronkov, editor, *Logic Programming and Automated Reasoning (LNCS 624)*, pp. 190–201 (July 1992).

[7] Plotkin, G. D. “Call-by-name, call-by-value, and the λ-calculus,” *Theoretical Computer Science*, Vol. 1, No. 2, pp. 125–159 (December 1975).

[8] 阪上紗里「型付き対称ラムダ計算の基礎理論」修士論文, お茶の水女子大学 (2008 年 1 月).

[9] Sumii, E., and B. C. Pierce “The Cryptographic λ-Calculus: Syntax, Semantics, Type System and Logical Relations,” 第 3 回プログラミングおよびプログラミング言語ワークショップ (PPL 2001), pp. 97–108 (March 2001).

[10] Wadler, P. “Call-by-value is dual to call-by-name,” *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP’03)*, pp. 189–201 (August 2003).

[11] Wadler, P. “Call-by-value is dual to call-by-name, Reloaded,” In J. Giesl, editor, *Term Rewriting and Applications (LNCS 3467)*, pp. 185–203 (April 2005).