擬似引用を持つ型付計算体系 λq

山本 和樹　岡本 暁広　五十嵐 淳　佐藤 雅彦
京都大学 大学院情報学研究科
{yamamo,aokamoto,igarashi,masahiko}@kuis.kyoto-u.ac.jp

概 要

擬似引用とは超変数を含む引用表現であり，超変数に別の引用表現を与えて評価することで，新たな引用表現を得ることができるものである．Lisp や Scheme は，擬似引用の機構を導入した代表的なプログラミング言語である．擬似引用中の超変数の評価は，ほぼ文字列の埋め込みに相当するため，擬似引用を用いて文脈計算における変数の動的束縛を表現することも可能である．本論文では，このような擬似引用・超変数・評価の機構を持つ型付計算体系 λq を提案し，subject reduction や合流性などの基本的な性質が成立することを証明する．

1 はじめに

擬似引用と超変数　一般に，ある言語 L の表現を引用符で囲むことにより別の言語 M の表現に埋め込むことが出来る．例えば，

「dog」は犬であり,「cat」は猫である．

とすることにより，言語 M の世界で言語 L を扱うことが出来る．この時，言語 M をメタ言語といい，言語 L を対象言語という．この例の場合，メタ言語 M は日本語であり対象言語 L は英語である．

擬似引用 (quasi-quotation) とは，論理学者 Quine によって導入された引用表現の拡張で，引用符の中に，超変数 (metavariable) と呼ばれる対象言語の文字列を動く変数を含んでいるような引用表現である．Quine の記法では，擬似引用には引用符 ⌜⌝ を用いる．例えば，

$$\ulcorner \alpha \text{ is a pen.} \urcorner$$

という表現は超変数 α を含む擬似引用である．超変数 α に適当な値を与えると，通常の引用が得られる．例えば α を「This」とすると，

「This is a pen.」

となる．このように，指示する対象言語の文字列で超変数を置き換えることを評価 (evaluation) という．

超変数の評価は，ほぼ文字列の埋め込みに相当する操作であるため，対象言語に変数束縛の機構が備わっている時，超変数に与えられた対象言語表現中の変数が束縛されうることに注意が必要である．例えば，

$$\ulcorner \forall x.(x+1)^2 = x^2 + \alpha + 1 \urcorner$$

という擬似引用において，超変数 α に「$2x$」を与えると

$$\text{「}\forall x.(x+1)^2 = x^2 + 2x + 1\text{」}$$

という引用が得られるが，ここで，埋め込まれた「$2x$」中の「x」が「$\forall x.$」に束縛されている．

擬似引用の概念はいくつかのプログラミング言語にも導入されており，代表的なものとして Lisp [16] や Scheme [7] が挙げられる．Scheme では，「'」が引用，「`」が擬似引用，「,」が擬似引用中の評価する部分を示す．例えば，

```
(define x 'one)
```

という Scheme プログラムは，超変数 x に引用された記号 one を与えていると考えることができる．ここで

```
`((lambda (y z) y) ,x two)
```

というプログラムを実行すると，

```
((lambda (y z) y) one two)
```

という結果が得られる．これは擬似引用「`((lambda (y z) y) x two)`」の x を評価して，引用「`((lambda (y z) y) one two)`」が得られたものと考えられる．また，上で述べた変数束縛のような状況も発生する．例えば，

```
(define x '(+ 1 y))
`(lambda (y) (* 2 ,x))
```

というプログラムからは

```
(lambda (y) (* 2 (+ 1 y)))
```

という結果が得られる．

研究目的と本論文の貢献　本研究の目的は，擬似引用，超変数の評価，評価による変数の動的束縛といった概念を統一的に取り扱うことができる型理論・計算体系を構築することにある．そのような計算体系を logical framework [5, 13] に応用することで，通常非形式的になりがちな超変数の表現を体系内部の概念として，より形式的に扱うことができるものと考えている．

そのために，本論文では単純型付 λ 計算を擬似引用・評価の機構で拡張した計算体系 λq を導入し，その基本的な性質のいくつかについて考察する．具体的には，λq の型付け規則・代入や簡約規則の形式的な定義を行い，簡約の前後で型付けが保存されるという性質 (subject reduction) や，合流性 (Church-Rosser property) を証明する．

本論文の構成　まず，第 2 節において計算体系 λq の概要を述べ，第 3 節で型付け規則・代入や簡約規則などを形式的に定義する．次に第 4 節で，λq の基本的な性質として subject reduction および合流性について述べ，その証明の概略を示す．第 5 節で，関連研究について述べ，最後に，第 6 節で，今後の課題を述べるとともに本研究のまとめを行う．

2　λq の概要

λq は，単純型付 λ 計算を擬似引用と評価の機構で拡張した型付計算体系である．また，メタ言語だけでなく引用される対象言語自体も λq であるため，メタメタレベル・対象対象レベルなど, 有限任意の段階の言語階層を表現することができる．

具体的には，擬似引用，評価を意味するような項として，それぞれ$\ulcorner a \urcorner$，$[\![a]\!]$という形のものを導入する．例えば，恒等関数に超変数 x を引数として与えるようなプログラムは，

$$\ulcorner \lambda y[y]([\![x]\!]) \urcorner$$

と表現できる．(ここで $[\cdots]$ は関数本体を，$(\cdots)$ は関数適用の引数を示している．) さらに，超変数 x へ two を与えるような項は，関数を用いることで

$$\lambda x[\ulcorner \lambda y[y]([\![x]\!]) \urcorner](\ulcorner two \urcorner)$$

と表現でき[1]，以下のように簡約される．

$$\begin{aligned} \lambda x[\ulcorner \lambda y[y]([\![x]\!]) \urcorner](\ulcorner two \urcorner) &\longrightarrow [x := \ulcorner two \urcorner](\ulcorner \lambda y[y]([\![x]\!]) \urcorner) \\ &\equiv \ulcorner \lambda y[y]([\![\ulcorner two \urcorner]\!]) \urcorner \\ &\longrightarrow \ulcorner \lambda y[y](two) \urcorner \\ &\longrightarrow \ulcorner two \urcorner \end{aligned}$$

この例からわかるように，一般には擬似引用された項が評価されると，

$$[\![\ulcorner a \urcorner]\!] \longrightarrow a$$

という簡約がおこり，引用が外れることになる．

引用の型　引用表現を考える時には，常にメタ言語・対象言語の区別に注意する必要がある．例えば，擬似引用

$$\ulcorner \alpha \text{ is a pen.} \urcorner$$

の超変数 α にメタ言語の表現「これ」を与えることは意味がない．同様に，

$$\ulcorner \lambda x[x] \urcorner(1)$$

という表現は，引用された対象レベルの関数に，メタレベルの関数適用を行っている不自然なものであると考えられる．そこで λq では，引用の型 $\Box A$ というものを導入し，レベルの混同が起らないようにしている．具体的には，$\ulcorner \lambda x[x] \urcorner$ という項には，「対象レベルでは関数である」ということを示す $\Box(A \Rightarrow A)$ という引用型が与えられるため，この項を引数に適用することは許されず，上の表現は型付け出来ないと判断される．

レベルと代入　このように，λq では項のどの部分がどのレベルの表現なのかを常に意識する必要がある．レベルの概念は，簡約および代入を定義する際に特に重要になる．例えば，

```
((lambda (x) `(f x ,x)) `y)
```

のような Scheme プログラムを実行すると

```
(f x y)
```

[1]擬似引用は引用を一般化したものなので，超変数を含まない場合にも同じ引用符を用いる．

が得られるように，$\lambda x[\ulcorner f(x)(\llbracket x \rrbracket)\urcorner](\ulcorner y\urcorner)$ は

$$\begin{aligned}\lambda x[\ulcorner f(x)(\llbracket x \rrbracket)\urcorner](\ulcorner y\urcorner) &\longrightarrow [x := \ulcorner y\urcorner]\ulcorner f(x)(\llbracket x \rrbracket)\urcorner \\ &\equiv \ulcorner f(x)(\llbracket \ulcorner y\urcorner \rrbracket)\urcorner \\ &\longrightarrow \ulcorner f(x)(y)\urcorner\end{aligned}$$

というように簡約される．ここで，x が代入の対象にならない場合があるのは，関数本体中に出現する二つの x の出現が違うレベルにあるためと考えることができる．つまり，ふたつめの ($\llbracket\ \rrbracket$ 内の) x のみが超変数であり代入の対象になるのである．後の節で見るように，代入の定義は厳密には対象となる変数の名前だけではなくレベルを取り入れたものとなる．

前節で超変数への代入が対象レベルでの変数束縛を発生させることに触れた．このようなことを表現したい一方で，代入による超変数同士の衝突は通常の λ 計算と同じく避けなければならない．そのため，λ 抽象式に対して超変数への代入を行う時には，その抽象式のレベルによって α 変換による束縛変数の名前替えを行うかどうかを決定する．例えば，

$$[z := f(x)(\ulcorner x\urcorner)](\lambda x[\ulcorner \lambda x[\llbracket z \rrbracket]\urcorner])$$

のような項を考えると，項 $\lambda x[\ulcorner \lambda x[\llbracket z \rrbracket]\urcorner]$ の外側の λx はメタレベルの抽象であり，代入される項 $f(x)(\ulcorner x\urcorner)$ の自由な超変数 (左側の x) と衝突しているので，ここでは束縛変数の名前替えを行い

$$[z := f(x)(\ulcorner x\urcorner)](\lambda y[\ulcorner \lambda x[\llbracket z \rrbracket]\urcorner])$$

とし，

$$\lambda y[[z := f(x)(\ulcorner x\urcorner)](\ulcorner \lambda x[\llbracket z \rrbracket]\urcorner)]$$

と等しいと考えるのが自然である．しかし，内側の λx は対象レベルの抽象であるので，代入をそのまま進め，

$$\lambda y[\ulcorner \lambda x[\llbracket f(x)(\ulcorner x\urcorner) \rrbracket]\urcorner]$$

とし，対象レベルでの変数束縛が発生するように代入を定義することになる．

subject reduction と合流性について　変数の動的束縛のある体系では，subject reduction，合流性などの基本的な性質を成立させるために，様々な工夫がなされている [6, 15]．λq においても，素朴な型付け規則，簡約規則ではこれらの性質は成り立たない．

例えば，$\lambda x[\ulcorner \lambda y[\llbracket x \rrbracket](1)\urcorner](\llbracket y \rrbracket)$ という項を考える．この項について，素朴に $\lambda x[\cdots](\llbracket y \rrbracket)$ と $\lambda y[\llbracket x \rrbracket](1)$ を簡約の対象と考えると，それぞれの簡約列は

$$\begin{aligned}\lambda x[\lceil \lambda y[\llbracket x \rrbracket](1)\rceil](\ulcorner y\urcorner) &\longrightarrow [x := \ulcorner y\urcorner](\ulcorner \lambda y[\llbracket x \rrbracket](1)\urcorner) \\ &\equiv \ulcorner \lambda y[\llbracket \ulcorner y\urcorner \rrbracket](1)\urcorner \\ &\longrightarrow \ulcorner \lambda y[y](1)\urcorner \\ &\longrightarrow \ulcorner 1\urcorner\end{aligned}$$

と

$$\begin{aligned}\lambda x[\ulcorner \lambda y[\llbracket x \rrbracket](1)\urcorner](\ulcorner y\urcorner) &\longrightarrow \lambda x[\ulcorner [y := 1](\llbracket x \rrbracket)\urcorner](\ulcorner y\urcorner) \\ &\equiv \lambda x[\ulcorner \llbracket x \rrbracket\urcorner](\ulcorner y\urcorner) \\ &\longrightarrow [x := \ulcorner y\urcorner](\ulcorner \llbracket x \rrbracket\urcorner) \\ &\equiv \ulcorner \llbracket \ulcorner y\urcorner \rrbracket\urcorner \longrightarrow \ulcorner y\urcorner\end{aligned}$$

となり，合流性が満たされない．これは，後者の簡約では，超変数の値が決まらないうちに対象レベルでの計算を行ったために起きた問題であると考えられる．λq では，簡約の定義の中で，メタレベルの変数が残っている項については，対象レベルでの計算が行えないという制限を加えることで，この問題を回避した．この例では，$\lambda y[[\![x]\!](1)$ は超変数 x を含むため，簡約されない．

次に，$\lambda X[\ulcorner \lambda x[x + [\![X]\!]]\urcorner]$ という項を考える．この項を項 $\ulcorner x(y)\urcorner$ に適用すると，

$$\begin{aligned}\lambda X[\ulcorner \lambda x[x + [\![X]\!]]\urcorner](\ulcorner x(y)\urcorner) &\longrightarrow \ulcorner \lambda x[x + [\![\ulcorner x(y)\urcorner]\!]]\urcorner \\ &\longrightarrow \ulcorner \lambda x[x + x(y)]\urcorner\end{aligned}$$

となり，この結果の項 $\ulcorner \lambda x[x + x(y)]\urcorner$ には型付けを行うことができない．この例では，同じ名前の変数が違う型になっているために起きた問題であると考えられる．このような場合，単純型付 λ 計算では束縛変数の名前換えを行って問題を回避するが，λq では動的束縛を考えるので，単純な束縛変数の名前換えができない．本研究では超変数を考慮した α 同値の概念を与え，変数名から型が一意に決まるような体系を定義することによって，この問題を回避した．

以上のことを念頭に，次節で λq の形式的定義を与えていく．

3 λq の定義

本節では，まず型や型環境に相当する仮定集合列 (hypothesis set sequence) を導入し，続いて項を構成するための型付け規則を定義する．そして，前節で述べたようなレベルを考慮した代入の定義を与え，簡約規則を定義する．

3.1 型，変数，仮定集合列

3.1.1 定義 [型]: 型の集合は，

$$A, B ::= K \mid B \Rightarrow A \mid \Box A$$

で定義する．

ここで K は基底型，$B \Rightarrow A$ は関数型を表す．$\Box A$ は引用の型と呼ばれるもので，型 A の項を引用したものの型を表している．また，$\Rightarrow$ は右結合し，$\Box$ は $\Rightarrow$ より結合が強いとする．例えば $\Box A \Rightarrow A \Rightarrow A$ は $(\Box A) \Rightarrow (A \Rightarrow A)$ という意味である．

変数，定数について以下のような仮定をする．

- 各型 A に対して， 変数の可算無限集合 $\mathbf{V}_A$ がある．
- 各型 A に対して， 定数の可算無限集合 $\mathbf{C}_A$ がある．
- A と B が異なるのなら $\mathbf{V}_A$ と $\mathbf{V}_B$ は共通部分を持たない．

つまり，変数を見ると型がわかるものであると仮定する．今後変数を表すには，x, y, z といった記号を用いることにする．また $x \in \mathbf{V}_A$ なる x を x^A と書き，変数は仮定 (hypothesis) と呼ぶこともある．

次に，通常の型付 λ 計算における型環境に相当する仮定集合列を導入する．前節で述べたように，項中の変数のレベルを区別する必要があるため，レベル毎の変数の集合の列として定義している．

3.1.2 定義: 仮定集合 (hypothesis set) とは，仮定の有限集合

$$\{x_1^{A_1}, ..., x_m^{A_m}\}$$

である．仮定集合列 Φ は次の形をしている．

$$\Phi = ..., \{x_{n1}^{A_{n1}}, ..., x_{nm_n}^{A_{nm_n}}\}_n, ..., \{x_{01}^{A_{01}}, ..., x_{0m_0}^{A_{0m_0}}\}_0, ..., \{x_{-n1}^{A_{-n1}}, ..., x_{-nm_{-n}}^{A_{-nm_{-n}}}\}_{-n}, ...$$

ただし $\{x_{n1}^{A_{n1}}, ..., x_{nm_n}^{A_{nm_n}}\}$ は有限個の n 以外に対しては空である．また，Φ に対して，$\Phi(n) = \{x_{n1}^{A_{n1}}, ..., x_{nm_n}^{A_{nm_n}}\}$ と定義する．

$\Phi(0)$ の変数は通常の意味での変数であり，$\Phi(1)$ の変数はメタレベルの変数である．$\Phi(n)$ 中の変数をレベル n の変数とことがある．また, 仮定集合列の中の空の仮定集合は省略することがある．

次に仮定集合列を扱う記法を定義しておく．

3.1.3 定義:

- $\Phi \cup \Psi$ は，任意の整数 n に対し $(\Phi \cup \Psi)(n) = \Phi(n) \cup \Psi(n)$ であるような仮定集合列である．
- $\Phi - \{x^A\}_n$ は，$\Psi(m) = \Phi(m)\ (m \neq n), \Psi(n) = \Phi(n) - \{x^A\}$ であるような仮定集合列 Ψ である．

3.1.4 例: 仮定集合列の例としては次のようなものがある．

$$\{z^{int}\}_1, \{x^A, y^B\}_0, \{w^{bool}, y^B\}_{-1}$$

この例では，変数 z はレベル 1 の変数であり，変数 x はレベル 0 の変数である．一つの変数が複数のレベルに存在することもある．例えば，変数 y はレベル 0 の変数であると同時にレベル -1 の変数でもある．

3.2 擬項

擬項は以下の BNF で表される．

$$a, b ::= x \mid c \mid \lambda x[a] \mid b(a) \mid \ulcorner a \urcorner \mid [\![a]\!]$$

ただし, x は変数, c は定数を表す.

3.3 型付け規則と項

λq の項は型付け規則を用いて型付け判断を導出することで構成される．型付け判断は,

$$\Phi \vdash_n a : A$$

の形をしており，Φ は仮定集合列，n は整数，a は擬項，A は型である．判断が以下の型付け規則で導出可能の時， 擬項 a は仮定集合列 Φ において， 型 A のレベル n の項であると言う．以下項を表すのに，a, b といった記号を用いる．

$$\frac{x^A \in \Phi(n)}{\Phi \vdash_n x : A} \quad \text{(T-Var)}$$

$$\frac{c \in \mathbf{C}_A}{\Phi \vdash_n c : A} \quad \text{(T-Const)}$$

$$\frac{\Phi \vdash_n b : B}{\Phi - \{x^A\}_n \vdash_n \lambda x^A[b] : A \Rightarrow B} \quad \text{(T-Abs)}$$

$$\frac{\Phi \vdash_n b : A \Rightarrow B \qquad \Psi \vdash_n a : A}{\Phi \cup \Psi \vdash_n b(a) : B} \quad \text{(T-App)}$$

$$\frac{\Phi \vdash_{n-1} a : A}{\Phi \vdash_n \ulcorner a \urcorner : \Box A} \quad \text{(T-Quote)}$$

$$\frac{\Phi \vdash_{n+1} a : \Box A}{\Phi \vdash_n [\![a]\!] : A} \quad \text{(T-Eval)}$$

$\Phi \vdash_0 a : A$ を省略して，$\Phi \vdash a : A$ と書くことがある．また，以下基準となる項からのレベルの差に着目することがある．これをランクという．例えば，項 $[\![x]\!](y)$ において，これがレベル n 項とすると変数 x はレベル $n+1$ の変数であるが，このことを x はランク 1 の変数であるという．

規則 T-Abs を用いる時には束縛変数 x と関数本体を表す項 b のレベルが同じでなければならない． 同様に, 規則 T-App を用いる時にも各項のレベルが同じでなければならない．規則 T-Quote は，項 a が $n-1$ レベルの型 A の項であるならば，それを引用した項 $\ulcorner a \urcorner$ は，レベル n で A の引用型（$\Box A$) の項であることを示している．逆に，規則 T-Eval は，項 a がレベル $n+1$ で A の引用型の時，それを評価した項 $[\![a]\!]$ は，レベル n で型 A の項であることを示している．

3.3.1 例: 以下に導出の例を与える．この例は，レベル 1 の項とレベル -1 の項が，レベル 0 で関数適用が行われることを示す例である．

$$\dfrac{\dfrac{\dfrac{X^{\Box(\Box int \Rightarrow int)} \in \{X^{\Box(\Box int \Rightarrow int)}\}}{\{X^{\Box(\Box int \Rightarrow int)}\}_1 \vdash_1 X : \Box(\Box int \Rightarrow int)}\ \text{(T-Var)}}{\{X^{\Box(\Box int \Rightarrow int)}\}_1 \vdash_0 [\![X]\!] : \Box int \Rightarrow int}\ \text{(T-Eval)} \qquad \dfrac{\dfrac{x^{int} \in \{x^{int}\}}{\{x^{int}\}_{-1} \vdash_{-1} x : int}\ \text{(T-Var)}}{\{x^{int}\}_{-1} \vdash_0 \ulcorner x \urcorner : \Box int}\ \text{(T-Quote)}}{\{X^{\Box(\Box int \Rightarrow int)}\}_1, \{x^{int}\}_{-1} \vdash_0 [\![X]\!](\ulcorner x \urcorner) : int}\ \text{(T-App)}$$

3.4 代入

前節でみたように，項中に現れる変数は名前が同じでもレベルが違う変数であるかもしれないため，代入操作は，変数のレベルを考慮する必要がある．具体的には，項 a 中のランク k の変数 x に b を代入することを $[x :=_k b](a)$ と記述する．例えば，$\ulcorner f([\![x]\!])(x) \urcorner$ という項で左側の x はランク 0 の変数であり，右側のものはランク -1 であるので，

$$[x :=_0 \ulcorner y \urcorner](\ulcorner f([\![x]\!])(x) \urcorner) \equiv \ulcorner f([\![\ulcorner y \urcorner]\!])(x) \urcorner$$

である．また，合流性を満すために，簡約はメタレベルの変数が (自由であるか束縛されているに関わらず) 存在する場合には起こらない，という条件を課すことになる．そのため，代入操作も対応する場合を排除して考える．つまり，$[x :=_0 y(z)](x([\![w]\!]))$ や，$[x :=_0 y(z)](x([\![\lambda w[w](\ulcorner z \urcorner)]\!]))$ のような代入は w がランク 1 の変数であり，代入の対象となる変数 x のランク (ここでは 0) より高いため定義されない．

以下では，代入操作を定義するために，項中の変数出現集合，自由変数集合を求める操作，α 同値などの定義を与える．

変数出現集合と自由変数集合

$V_k(a)$ は，a が Φ においてレベル n の項の時 a に出現する レベル $n+k$ の変数出現集合，すなわちランク k の変数出現集合を表す．また，$FV_k(a)$ は，a が Φ においてレベル n の項の時 a に出現する レベル $n+k$ の自由変数集合，すなわちランク k の自由変数集合を表す．$V_k(a)$, $FV_k(a)$ を以下のように定義する．

$$
\begin{array}{rcl@{\qquad}rcl}
V_k(x) & = & \begin{cases} \{x\} & (k=0) \\ \emptyset & (k \neq 0) \end{cases} & FV_k(x) & = & \begin{cases} \{x\} & (k=0) \\ \emptyset & (k \neq 0) \end{cases} \\
V_k(\lambda x[a]) & = & \begin{cases} V_0(a) \cup \{x\} & (k=0) \\ V_k(a) & (k \neq 0) \end{cases} & FV_k(\lambda x[a]) & = & \begin{cases} FV_0(a) - \{x\} & (k=0) \\ FV_k(a) & (k \neq 0) \end{cases} \\
V_k(b(a)) & = & V_k(b) \cup V_k(a) & FV_k(b(a)) & = & FV_k(b) \cup FV_k(a) \\
V_k(\ulcorner a \urcorner) & = & V_{k+1}(a) & FV_k(\ulcorner a \urcorner) & = & FV_{k+1}(a) \\
V_k([\![a]\!]) & = & V_{k-1}(a) & FV_k([\![a]\!]) & = & FV_{k-1}(a)
\end{array}
$$

$V_k(x)$ においては，変数 x はランクが 0 なので，$k = 0$ の時のみ変数出現集合に含まれる．$V_k(\ulcorner a \urcorner), V_k([\![a]\!])$ においては，それぞれ引用された項の変数，あるいは評価される項の変数を順次探してゆくので，引用された項では上に，評価された項では下へそれぞれレベルが変動している．$FV_k(a)$ についても同様である．ただし，$V_0(\lambda x[a])$ においては x を変数出現に含めている一方，$FV_0(\lambda x[a])$ においては x を自由変数から除いているところに注意されたい．

3.4.1 例:

$$
\begin{array}{l}
V_0(x(\ulcorner y \urcorner)) = V_0(x) \cup V_0(\ulcorner y \urcorner) = \{x\} \cup V_1(y) = \{x\} \\
FV_0(x(\ulcorner y \urcorner)) = FV_0(x) \cup FV_0(\ulcorner y \urcorner) = \{x\} \cup FV_1(y) = \{x\} \\
V_{-1}(x(\ulcorner y \urcorner)) = V_{-1}(x) \cup V_{-1}(\ulcorner y \urcorner) = V_0(y) = \{y\} \\
FV_{-1}(x(\ulcorner y \urcorner)) = FV_{-1}(x) \cup FV_{-1}(\ulcorner y \urcorner) = FV_0(y) = \{y\} \\
V_0(\lambda x[x(y)]) = V_0(x(y)) \cup \{x\} = V_0(x) \cup V_0(y) \cup \{x\} = \{x, y\} \cup \{x\} = \{x, y\} \\
FV_0(\lambda x[x(y)]) = FV_0(x(y)) - \{x\} = (FV_0(x) \cup FV_0(y)) - \{x\} = \{x, y\} - \{x\} = \{y\}
\end{array}
$$

$x(\ulcorner y \urcorner)$ の例は，変数 x はランク 0 の変数であるが，変数 y はランク -1 の変数であることを示している．このように，本体系では常にレベルを意識して変数を取り扱う．また $\lambda x[x(y)]$ の例で，自由変数と変数出現では変数束縛に関する扱いが違うことがわかる．

置き換え

次に，自由変数を別の変数で置き換える操作を定義する．この操作は α 同値を定義するのに用いられる．置き換えは $[y \leftarrow_l z](a)$ と記述され，直感的には項 a のランク l の変数 y を変数 z で置

き換えるという意味である．置き換え操作も代入と同様に，ランクの高い変数が現れている項に対しては定義されない．

3.4.2 定義: 任意の $k > l$ に対し $V_k(a) = \emptyset$ かつ $z \notin V_l(a)$ である時，$[y \leftarrow_l z](a)$ を以下のように定義する．

$$
\begin{aligned}
[y \leftarrow_l z](y) &\equiv \begin{cases} y & (l > 0) \\ z & (l = 0) \\ \text{未定義} & (l < 0) \end{cases} \\
[y \leftarrow_l z](x) &\equiv x \qquad (x \not\equiv y) \\
[y \leftarrow_l z](\lambda x[b]) &\equiv \begin{cases} \lambda x[[y \leftarrow_l z](b)] & (l > 0) \\ \lambda w[[y \leftarrow_0 z]([x \leftarrow_0 w](b))] & (l = 0) \\ \text{未定義} & (l < 0) \end{cases} \\
[y \leftarrow_l z](b(c)) &\equiv [y \leftarrow_l z](b)([y \leftarrow_l z](c)) \\
[y \leftarrow_l z](\ulcorner b \urcorner) &\equiv \ulcorner [y \leftarrow_{l+1} z](b) \urcorner \\
[y \leftarrow_l z](\llbracket b \rrbracket) &\equiv \llbracket [y \leftarrow_{l-1} z](b) \rrbracket
\end{aligned}
$$

$[y \leftarrow_0 z](\lambda x[b])$ においては，$w \not\equiv y$ かつ $w \not\equiv z$ かつ $w \notin FV_0(b)$ となる w を選ぶ．

置き換えもレベルを意識して行われる．したがって，$[y \leftarrow_l z](y)$ の置き換えにおいてランク l が 0 でない時には，レベルが異なるので置き換えが行われない．また，以後 $[y \leftarrow_0 z]$ を省略して $[y \leftarrow z]$ と書くことにする．

次の補題で置き換えの定義が妥当であることを示す．

3.4.3 補題:

$$\forall k > l, V_k(a) = \emptyset \text{ ならば } [y \leftarrow_l z](a) \text{ は定義されている．}$$

証明: 項 a の構成に関する帰納法による． □

α 同値

α 同値とは，直感的に述べると，束縛変数の名前だけが異なり，他はすべて同じである項に成り立つ同値関係のことである．ただし，本体系では文脈計算を考えるので超変数を含む場合はこの限りではない．例えば，$\lambda x[[y]]$ は x よりもランクの高い変数 y をその λ 抽象本体に含むので，$\lambda z[[y]]$ と α 同値ではないと考える．

3.4.4 定義: 二項関係 $a \equiv_\alpha b$，すなわち 2 つの項 a, b が α 同値であることを

$$[x \leftarrow z](a) \equiv_\alpha [x' \leftarrow z](a') \text{ ならば } \lambda x[a] \equiv_\alpha \lambda x'[a'] \text{ ただし，} z \notin FV_0(a), z \notin FV_0(a')$$

を含む最小の合同関係であると定義する．

ここで，置き換え $[x \leftarrow z](a)$ が定義できる時のみ α 同値が定義される．つまり，項 $\lambda x[a]$ が内部に超変数の出現を含む時，これと α 同値関係にあるのは $\lambda x[a]$ のみである．以降，α 同値な項は同じ項であると考えることにする．

代入

以上の準備をふまえて，代入操作を定義する．代入は $[x :=_k a](b)$ のように表され，項 b 中のランク k の変数 x に a を代入した項を意味する．

3.4.5 定義: 代入 $[x :=_k a](b)$ は

$$\text{任意の } l > 0 \text{ に対し } V_l(a) = \emptyset \text{ かつ任意の } l > k \text{ に対し } V_l(b) = \emptyset$$

が成り立つ時のみ，以下のように定義する．

$$\begin{aligned}
[x :=_k a](x) &\equiv \begin{cases} x & (k > 0) \\ a & (k = 0) \\ \text{未定義} & (k < 0) \end{cases} \\
[x :=_k a](y) &\equiv y \qquad (x \not\equiv y) \\
[x :=_k a](\lambda y[b]) &\equiv \begin{cases} \lambda y[[x :=_k a](b)] & (k > 0) \\ \lambda y[[x :=_0 a](b))] & (k = 0) \quad \text{ただし } x \not\equiv y, y \notin FV_0(a) \\ \text{未定義} & (k < 0) \end{cases} \\
[x :=_k a](c(d)) &\equiv [x :=_k a](c)([x :=_k a](d)) \\
[x :=_k a](\ulcorner c \urcorner) &\equiv \ulcorner [x :=_{k+1} a](c) \urcorner \\
[x :=_k a](\llbracket c \rrbracket) &\equiv \llbracket [x :=_{k-1} a](c) \rrbracket
\end{aligned}$$

$[x :=_0 a](\lambda y[b])$ の定義においては，一般性を失わず，常に付帯条件が満たされていると仮定できる．これは，代入の定義される条件から α 同値の概念を用いて束縛変数の名前を変えられることがわかるからである．代入の定義の妥当性も置き換えと同様に証明することができる．また，以後 $[x :=_0 a]$ を省略して，$[x := a]$ と書くことにする．

3.4.6 例:

$$\begin{aligned}
[x :=_1 3](f(x)(\llbracket x \rrbracket)) &\equiv [x :=_1 3](f)([x :=_1 3](x))([x :=_1 3](\llbracket x \rrbracket)) \\
&\equiv f(x)(\llbracket [x :=_0 3](x) \rrbracket) \\
&\equiv f(x)(\llbracket 3 \rrbracket)
\end{aligned}$$

$$\begin{aligned}
[y := x(\ulcorner z \urcorner)](\lambda x[\ulcorner \lambda z[\llbracket y \rrbracket] \urcorner]) &\equiv \lambda w[[y := x(\ulcorner z \urcorner)](\ulcorner \lambda z[\llbracket y \rrbracket] \urcorner)] \\
&\equiv \lambda w[\ulcorner [y :=_1 x(\ulcorner z \urcorner)](\lambda z[\llbracket y \rrbracket]) \urcorner] \\
&\equiv \lambda w[\ulcorner \lambda z[\llbracket [y := x(\ulcorner z \urcorner)](y) \rrbracket] \urcorner] \\
&\equiv \lambda w[\ulcorner \lambda z[\llbracket x(\ulcorner z \urcorner) \rrbracket] \urcorner]
\end{aligned}$$

最初の例では，項 $f(x)(\llbracket x \rrbracket)$ の左の x はランク 0 の自由変数であるが，右の x はランク 1 の自由変数である．この例で行いたい代入はランク 1 の変数への代入なので，左の x には代入が行われずに右の x にだけ代入が行われる．次の例では，束縛変数 x については名前換えが起きているが，束縛変数 z については名前換えが起きていない．束縛変数 x についてはレベルが等しいため α 同値の概念を用いて名前を換えをしているが，束縛変数 z に関しては超変数に対しての代入になっているため，変数 z の動的束縛が起きているのである．このように，本体系では変数や項のレベルを定義し，それを利用することによって代入が行われるように定義されている．

3.5 簡約規則

$a \longrightarrow b$ という二項関係を以下で定義する．$a \longrightarrow b$ は項 a が簡約されて項 b になる事を表す．

$$\lambda x[b](a) \longrightarrow [x := a](b) \qquad \text{(R-Beta)}$$

$$[\![\ulcorner a \urcorner]\!] \longrightarrow a \qquad \text{(R-Eval)}$$

$$\frac{b \longrightarrow c}{\lambda x[b] \longrightarrow \lambda x[c]} \qquad \frac{b \longrightarrow c}{b(a) \longrightarrow c(a)} \qquad \frac{a \longrightarrow c}{b(a) \longrightarrow b(c)}$$

$$\frac{a \longrightarrow c}{\ulcorner a \urcorner \longrightarrow \ulcorner c \urcorner} \qquad \frac{a \longrightarrow c}{[\![a]\!] \longrightarrow [\![c]\!]}$$

ただし，規則 R-Beta は，$\forall m > 0.V_m(a) = V_m(b) = \emptyset$ の時のみ適用可能であるとする．これは超変数が出現する場合は簡約を行わないという考えに基づく．規則 R-Eval は引用された項を評価することによって元の項に戻ることを示している．以後，a から 0 回以上簡約を繰り返して b が得られる時，$a \xrightarrow{*} b$ と書く．

3.5.1 例:

$\lambda x[\ulcorner f([\![x]\!])(x) \urcorner](\ulcorner y \urcorner) \longrightarrow [x := \ulcorner y \urcorner](\ulcorner f([\![x]\!])(x) \urcorner) \equiv \ulcorner f([\![\ulcorner y \urcorner]\!])(x) \urcorner \longrightarrow \ulcorner f(y)(x) \urcorner$
$\lambda x[y([\![\ulcorner z \urcorner]\!])](w) \longrightarrow \lambda x[y([\![z]\!])](w)$
$\lambda x[y([\![z]\!])](w) \longrightarrow [x := w](y([\![z]\!]))$ とは簡約出来ない．

項 $\lambda x[\ulcorner f([\![x]\!])(x) \urcorner](\ulcorner y \urcorner)$ の簡約例では，左の変数 x と右の変数 x はそれぞれレベルが異なる．簡約の結果はランクが 0 の変数 x，つまり左の変数 x にだけ代入が行われることを示している．

項 $\lambda x[y([\![\ulcorner z \urcorner]\!])](w)$ 簡約例では，$[\![\ulcorner z \urcorner]\!]$ は変数 z が引用されその後に評価が行われる事を示している．したがって，$[\![\ulcorner z \urcorner]\!]$ は簡約されて z となる．(規則 R-Eval)

また，項 $\lambda x[y([\![z]\!])](w)$ は，一見規則 R-Beta により簡約が行えそうである．しかし，この項中の変数 z はランク 1 の変数，すなわち超変数である．したがって規則 R-Beta による簡約は行うことができないのである．

4 λq の性質

以下では，本研究で証明した性質である subject reduction，合流性について述べる．

4.1 Subject Reduction

subject reduction とは，簡約の前後で型付けが変わらないことを意味する性質である．

4.1.1 定理 [subject reduction]: $\Phi \vdash_n a : A$ かつ $a \longrightarrow b$ ならば，$\Phi \vdash_n b : A$

証明の準備として，項に代入が行われてもその項の型が変わらないことを以下の補題で示す．

4.1.2 補題: $\forall l > 0, V_l(a) = \emptyset$ かつ $\forall l > k, V_l(b) = \emptyset$ の時，

$$\Phi \vdash_{n-k} b : B \ , \ \Psi \vdash_n a : A \text{ ならば } (\Phi - \{x^A\}_n) \cup \Psi \vdash_{n-k} [x :=_k a](b) : B$$

証明: 証明は項 b の構成に関する帰納法による． □

定理 4.1.1 の証明: $a \longrightarrow b$ の構成に関する帰納法で証明できる．ここでは，$a \equiv \lambda x[c](d)$, $b \equiv [x := d](c)$ の場合と $a \equiv [\![\ulcorner c \urcorner]\!], b \equiv c$ の場合についてのみ述べる．

項 $\lambda x[c](d)$ の導出木は $\Phi \equiv \Phi' \cup \Phi'', \Phi' \equiv \Psi - \{x^B\}_n$ として

$$\dfrac{\dfrac{\vdots}{\Psi \vdash_n c : A}}{\Phi' \vdash_n \lambda x[c] : B \Rightarrow A} \qquad \dfrac{\vdots}{\Phi'' \vdash_n d : B}$$
$$\Phi \vdash_n \lambda x[c](d) : A$$

の形をしている．したがって，補題 4.1.2 より $(\Psi - \{x^B\}_n) \cup \Phi'' \vdash_n [x := d](c) : A$ が成立する．

項 $[\![\ulcorner c \urcorner]\!]$ の導出木は

$$\dfrac{\dfrac{\dfrac{\vdots}{\Phi \vdash_n c : A}}{\Phi \vdash_{n+1} \ulcorner c \urcorner : \Box A}}{\Phi \vdash_n [\![\ulcorner c \urcorner]\!] : A}$$

の形をしている．したがって，$\Phi \vdash_n c : A$ が成立する． □

4.2 合流性

次に合流性の証明に取りかかる．合流性とは，一つの項から別の簡約列によって得られた二つの項が，それぞれさらに簡約を繰り返すことによって同じ項へと簡約することができる (合流できる) ということを意味する性質である．

4.2.1 定理 [Church-Rosser]: $a \xrightarrow{*} b_i \ (i = 1, 2)$ ならば，ある c が存在して $b_i \xrightarrow{*} c \ (i = 1, 2)$

証明は高橋 [19] の並行簡約の技法を参考にした．並行簡約とは，直感的には項の簡約できる部分をいくつか同時に簡約するような簡約である．この手法では，一つの項が一回の並行簡約で別々の項になった時，一回の並行簡約で合流するという性質 (diamond property) を利用する．a が b に並行簡約されることを，$a \Longrightarrow b$ のように書く．

4.2.2 定義: 関係 $a \Longrightarrow b$ を以下の規則で定義する．

$$\dfrac{\lambda x[b] \Longrightarrow \lambda x[b'] \qquad a \Longrightarrow a'}{\lambda x[b](a) \Longrightarrow [x := a'](b')} \quad \text{(P-Beta)}$$

$$\dfrac{a \Longrightarrow a'}{[\![\ulcorner a \urcorner]\!] \Longrightarrow a'} \quad \text{(P-Eval)}$$

$$x \Longrightarrow x \qquad\qquad c \Longrightarrow c$$

$$\dfrac{b \Longrightarrow b'}{\lambda x[b] \Longrightarrow \lambda x[b']} \qquad \dfrac{b \Longrightarrow b' \qquad a \Longrightarrow a'}{b(a) \Longrightarrow b'(a')} \qquad \dfrac{a \Longrightarrow a'}{\ulcorner a \urcorner \Longrightarrow \ulcorner a' \urcorner} \qquad \dfrac{a \Longrightarrow a'}{[\![a]\!] \Longrightarrow [\![a']\!]}$$

ただし，規則 P-Beta は，$\forall m > 0.V_m(a) = V_m(b) = \emptyset$ の時のみ適用可能であるとする．以後，a から 0 回以上並行簡約を繰り返して b が得られる時，$a \stackrel{*}{\Longrightarrow} b$ と書く．

以下で証明を述べる．まず最初に，簡約の繰り返しと並行簡約の繰り返しが全く同じものであることを証明する．

4.2.3 補題: $a \stackrel{*}{\longrightarrow} b$ と $a \stackrel{*}{\Longrightarrow} b$ は同値である．

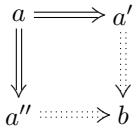
証明: $a \longrightarrow b$ ならば $a \Longrightarrow b$ は $a \longrightarrow b$ の構成に関する帰納法で証明できる．逆に $a \Longrightarrow b$ ならば $a \stackrel{*}{\longrightarrow} b$ は $a \Longrightarrow b$ の構成に関する帰納法で証明できる．以上により補題 4.2.3 が示された． □

次に，並行簡約の diamond property を証明するために必要な補題を示す．補題 4.2.4 は，項中の独立した部分項は並行簡約により同時に簡約できることを示す．

4.2.4 補題: $a_i \Longrightarrow b_i (i = 1, 2)$, 任意の $m > 0$ に対して $V_m(a_2) = V_m(b_2) = V_{m+k}(a_1) = V_{m+k}(b_1) = \emptyset$ の時，$[x :=_k a_2](a_1) \Longrightarrow [x :=_k b_2](b_1)$

証明: 項 a_1 の構成に関する帰納法で証明できる． □

4.2.5 補題: $\Longrightarrow$ は diamond property を満たす．



つまり，任意の a に対して，$a \Longrightarrow a'$ かつ $a \Longrightarrow a''$ ならばある b が存在し，$a' \Longrightarrow b$ かつ $a'' \Longrightarrow b$ である．

証明: 項 a の構成に関する帰納法で証明できる．以下では，$a \equiv \lambda x[d](e)$ かつ d, e 中に超変数が出現しない場合を示す．

帰納法の仮定より，$d \Longrightarrow d_1$ かつ $d \Longrightarrow d_2$ ならば f が存在し，$d_1 \Longrightarrow f$ かつ $d_2 \Longrightarrow f$ であり，$e \Longrightarrow e_1$ かつ $e \Longrightarrow e_2$ ならば g が存在し，$e_1 \Longrightarrow g$ かつ $e_2 \Longrightarrow g$ である．

したがって，

- $a' \equiv \lambda x[d'](e')$ の時: $a' \Longrightarrow [x := g](f) \equiv b$
- $a' \equiv [x := e'](d')$ の時: 補題 4.2.4 より，$a' \Longrightarrow [x := g](f) \equiv b$

a'' についても同様である． □

定理 4.2.1 の証明: 補題 4.2.5 から $a \stackrel{*}{\Longrightarrow} b_i$ $(i = 1, 2)$ ならば，ある c が存在し，$b_i \stackrel{*}{\Longrightarrow} c$ $(i = 1, 2)$ が言える．補題 4.2.3 より 4.2.1 の定理が成立する． □

5 関連研究

擬似引用の概念は，1955 年にアメリカの論理学者 Quine によって提案 [11] されている．λq における擬似引用の記法 $\ulcorner \cdots \urcorner$ は彼の記法にしたがったものである．

λq と文脈計算　本論文で提案する λq は佐藤・桜井・亀山 [20] による $\lambda\mathbf{q}$ を，文脈 [15, 18, 12, 8, 1, 9] を使った計算を表現できるように拡張したものと考えることができる．文脈とは，大まかに言って，穴 [ ] をもつ λ 項であり，穴に別の項を埋め込むことにより，文脈から新たな項を得ることができる．例えば，C を $\lambda x[x + [\,]]$ という文脈とし，C 中の穴に $x * y$ という項を埋め込むと，$\lambda x[x + x * y]$ という項が得られる．この時，埋め込みは代入操作とは違い，文脈中の λx による，埋め込まれた項中の自由変数 x の束縛が発生する．λq では，文脈は穴を超変数とした λ 抽象項として，埋め込みは関数適用で表現することができる．例えば，上の C は $\lambda X^{\Box int}[\ulcorner\lambda x^{int}[x + [\![X]\!]]\urcorner]$，穴への $x * y$ の埋め込みは $\lambda X^{\Box int}[\ulcorner\lambda x^{int}[x + [\![X]\!]]\urcorner](\ulcorner x * y\urcorner)$ と表現される．実際，この項を簡約すると，$\ulcorner\lambda x^{int}[x + x * y]\urcorner$ という項を得ることができる．一方，佐藤・桜井・亀山 [20] では，変数の束縛は発生せず，通常の λ 計算のように，代入時に変数の名前替えが行われ

$$\lambda X^{\Box int}[\ulcorner\lambda x^{int}[x + [\![X]\!]]\urcorner](\ulcorner x * y\urcorner) \longrightarrow \ulcorner\lambda z^{int}[z + x * y]\urcorner$$

と計算が進む．

Talcott [18]，Mason [9] らの文脈計算は λq と違い，文脈をメタな体系外部の概念として取り扱っているため，λq での合流性を得るための問題などは発生しない．Hashimoto, Ohori [6] や，Sato, Sakurai, Kameyama の体系 [15] では文脈が第一級の値として体系内部の概念として扱われている．Hashimoto, Ohori の文脈計算の体系には，穴を含む λ 抽象式の適用は簡約されないという条件が見られるが，これは λq での合流性を得るための R-Beta 規則の付帯条件に対応するものと考えられる．一方，Sato, Sakurai, Kameyama の体系は，明示的環境 [14] を導入することにより，そのような制限がないにもかかわらず，合流性が保証される体系になっている．R-Beta 規則の付帯条件を緩め，超変数が現れている項も簡約できるようにするためには，同様な型の改良などを行う必要があるが，一方で，R-Beta の付帯条件は，超変数 (穴) を含む項は (対象レベルでは) 不完全な項であり，穴埋めが済んで初めて通常の計算を進められるという立場からは妥当なものであると考えられる．

段階的計算 (staged computation) のための型付計算体系　近年，部分計算や動的コード生成などの段階的計算 (staged computation) を安全に行うための型システムの研究が盛んになされている．λq も，高いレベルからの計算を順に行うという意味で，段階的計算の一種と見なせる．ここでは，段階的計算のための代表的な計算体系として，様相論理 S4 に対応する $\lambda^{\Box}$ [3] と線形時相論理に対応し，オフライン部分計算で行われる (多段階) 束縛時解析 [4] を表現した $\lambda^{\bigcirc}$ [2] と λq を比較する．

これらの体系では，コードという，後 (の段階) で評価を行って値を計算することができる項を計算対象として扱っている．型システムは，λq と似た考え方で構成されており，コードの型は $\lambda^{\Box}$ では $\Box A$，$\lambda^{\bigcirc}$ では $\bigcirc A$ という形をしている．例えば，$\Box int$ は，整数を計算するコードの型を示す．$\lambda^{\Box}$ での計算の段階やコードといった概念は，λq のレベルや擬似引用などと対応すると考えられる．$\lambda^{\Box}$ と $\lambda^{\bigcirc}$ の大きな違いは，コードが (コードレベルの) 自由変数を含みうるかにある．$\lambda^{\Box}$ では Lisp の eval 関数のような，動的コード生成に対応する機構があるため，動的コード生成を安全に行うために，コード型の値は閉じていなければならないという制限が課されている．そのため λq での変数の動的束縛のような状況はそもそも発生することはない．一方，$\lambda^{\bigcirc}$ には，動的コード生成の機構はないが，λq が自由変数を含む疑似引用を扱えるのと同様に，自由変数を含むコードを扱うことができる．実際，型付け規則などは λq のものと非常に似通っている．ただし，計算においては λq に見られる変数の動的束縛は発生しない．これは，元々 $\lambda^{\bigcirc}$ が部分計算という多段階計算を表現しているためであると考えられる．つまり，部分計算の結果は，束縛時注釈を取り除

いた元のプログラムの意味を保存することが前提なので，元のプログラムが静的束縛を行う言語で書かれている限り，部分計算の過程で動的束縛が発生してはならない．

閉じたコードしか操作できないという $\lambda^{\Box}$ の欠点を克服するために，動的コード生成の下で自由変数のあるコードを安全に扱うための型システムとして [10, 17] などが提案されているが，これらを含め，著者の知る限りでは，段階的計算のための型付き言語として，動的束縛をとりいれているものは提案されていないようである．

6 おわりに

本論文では，擬似引用を表現できる型付計算体系 λq を定義し，その基本的な性質として，subject reduction，合流性を示した．型付けにおいては，引用のための型を導入することで，対象レベル・メタレベルの混同を防ぐことができた．λq では，異なるレベルに同じ名前の変数を使用することができるため，代入操作においては，(型付けされた) 項のレベルを考慮にいれる必要があった．これは，超変数への代入による対象レベルでの束縛の発生を形式化する上でも重要である．また，合流性を得るために簡約が高いレベルから起こるような制限を行ったが，これは，擬似引用された表現は超変数の値が決まらないと，その (対象レベルにおける) 意味がとれないことから考えても妥当な制限であると考えられる．

今後の課題としては，まず，subject reduction，合流性とともに基本的な性質である強正規化性 (strong normalizability) の証明を行うことが挙げられる．

謝辞

本研究の一部は文部科学省科学研究費特定領域研究「変数の動的束縛機構をもつ新しいソフトウェアの理論的研究: 引用と文脈の型理論」より援助を受けています．また，本論文に関して貴重なコメントを頂きました PPL 査読者に対して感謝の意を表します．

参考文献

[1] Laurent Dami. A lamda-calculus for dynamic binding. *Theoretical Computer Science*, Vol. 192, No. 2, pp. 201–231, 1998.

[2] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, pp. 184–195, July 1996.

[3] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, Vol. 48, No. 3, pp. 555–604, May 2001.

[4] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In *Proceedings of Programming Languages, Implementations, Logics and Programs (PLILP'95)*, Vol. 982 of *Lecture Notes in Computer Science*, pp. 259–278. Springer-Verlag, September 1995.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, Vol. 40, No. 1, pp. 143–184, 1992. Preliminary version in LICS'87.

[6] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, Vol. 266, No. 1–2, pp. 249–272, 2001.

[7] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, September 1998. Also appeared in *ACM SIGPLAN Notices*, Vol. 33, No. 9, October, 1998.

[8] Shinn-Der Lee and Daniel P. Friedman. Enriching the lambda calculus with contexts: Toward a theory of incremental program construction. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pp. 239–250, 1996.

[9] Ian Mason. Computing with contexts. *Higher-Order and Symbolic Computation*, Vol. 12, pp. 171–201, 1999.

[10] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *Proceedings of European Symposium on Programming (ESOP)*, Vol. 1576 of *Lecture Notes in Computer Science*, pp. 193–207. Springer-Verlag, 1999.

[11] Willard Van Orman Quine. *Mathematical logic.* Harvard Univ. Press., 1955.

[12] David Sands. Computing with contexts – a simple approach. In *Proceedings of Higher-Order Operational Techniques in Semantics (HOOTS II)*, Vol. 10 of *Electronic Notes in Theoretical Computer Science*, p. 16 pages, 1998.

[13] Masahiko Sato. Theory of judgments and derivations. In S. Arikawa and A. Shinohara, editors, *Proceedings of Discovery Science*, Vol. 2281 of *Lecture Notes in Artificial Intelligence*, pp. 78–122. Springer-Verlag, 2001.

[14] Masahiko Sato, Takafumi Sakurai, and Rod Burstall. Explicit environments. *Fundamenta Informaticae*, Vol. 45, No. 1–2, pp. 79–115, 2001. An extended abstract has appeared in Proceedings of *Typed Lambda Calculi and Applications (TLCA)*, Springer LNCS 1581, pages 340–354, 1999.

[15] Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama. A simply typed context calculus with first-class environments. *Journal of Functional and Logic Programming*, Vol. 2002, No. 4, pp. 1–41, March 2002.

[16] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, 2nd edition, 1990.

[17] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, January 2003.

[18] Carolyn Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, Vol. 112, No. 1, pp. 99–143, 1993.

[19] 高橋正子. 計算論　計算可能性とラムダ計算. 近代科学社, 1991.

[20] 佐藤雅彦, 桜井貴文, 亀山幸義. 引用と超変数. 日本ソフトウェア科学会大会論文集, 2002.