Application and Composition in Functional Programming

HIROFUMI YOKOUCHI*

Functional programming is examined in a theoretical manner. We start with the consideration that there are two kinds of functional programming languages: (1) applicative languages (LISP, ML, KRC) and (2) compositional languages (FP). To formalize these types of programming languages, we will define two formal systems: the typed λ-calculus with product, corresponding to the applicative languages, and the composition calculus, corresponding to the compositional languages. Then, we will show that both formal systems are essentially the same. This result is an application of recent work about the relationship between the λ-calculus and category theory.

1. Introduction

This paper deals with functional programming languages in the theoretical sense. We focus on a difference in the basic operations that construct programs from primitive functions. Functional programming languages are classified into two types as follows:

(1) LISP, ML[5], KRC[8,9], etc., and

(2) FP(FFP)[1,2,3].

The former are based on *application* and *λ-abstraction*, and the latter are based on *composition*. We call these two types of languages *applicative languages* and *compositional languages*. We should notice especially that the compositional languages are defined without variables. This is the most remarkable characteristic of the compositional languages.

In this paper we formalize these two types of languages. For the applicative languages we use the λ-calculus. For the compositional languages, we introduce a formal system called the *composition calculus*. The composition calculus resembles Backus FP except that the composition calculus includes an operation $\Lambda(f)$ corresponding to the curried function of f. We examine the relationship between the λ-calculus and the composition calculus, and we show that the two formal systems have the same descriptive power. It is clear that composition can be represented by application and λ-abstraction. For example, the composite $g \circ f$ of two functions f and g is expressed as $\lambda x.g(f(x))$. However, the question is whether application and λ-abstraction can be represented by composition. In other words, can we naturally formalize the composition calculus so that it has the same descriptive power as the λ-calculus?

This paper mainly deals with typed languages for both applicative languages and compositional languages, because the concept of type is naturally introduced into functional programming. But the entire discussion in this paper can be immediately carried over to type-free languages. For typed languages it is important to choose the set of types. We take two operations $(-)\times(-)$ and $(-)^{(-)}$ that construct new types from given types. For each pair of types a and b, $a\times b$ is a type that intuitively represents the product of a and b, and b^a represents the set of functions from a to b. The mechanism of product is essential for compositional languages. This situation is different from the λ-calculus.

In Section 2, we define the typed λ-calculus with product, and in Section 3, we define the typed composition calculus. In Section4, we show that both formal system are essentially the same.

2. The typed λ-calculus with product

The *typed λ-calculus with product* is defined by adding pairing operations $\langle -, - \rangle$ and projections π_1 and π_2 to the usual λ-calculus[4]. In this paper, however, we define the formal system without using the projections, the reason being twofold: (1) to maintain a correspondence with conventional programming languages and (2) to simplify the proof of the Equivalence Theorem to be presented in Section 4. Also we intend to make the system explicitly express functions that have more than two variables. In the usual λ-calculus, the λ-abstraction is defined against only one variable. However, we directly define λ-abstraction against a sequence of variables. In most programming languages, more than two variables may be given as formal parameters in declarations of functions or procedures. In order to formalize this mechanism, we introduce a *tuple variable* that represents a sequence of variables.

Below we give the formal definition of the typed λ-calculus with product, which we simply call the λ-calculus.

Definition. (Type)

A set of types is a set P equipped with structure $(1, (-)\times(-), (-)^{(-)})$, where $1\in P$, and $(-)\times(-)$ and

*Science Institute, IBM Japan, Ltd.

$(-)^{(-)}$ are binary functions from $P \times P$ to P.

In this paper, a type may be regarded as a domain of functions. More rigorously a type is a name of a domain. Type 1 means a singleton set. Type 1 is introduced for a technical reason. We regard a 0-ary function as a function from type 1.

A language of the λ-calculus consists of variables and constants. Whenever a constant k is given, a type of k is uniquely determined. At least one constant with type 1 is always given. We arbitrarily choose one, denoted by $\langle\ \rangle$, and fix it. For each type a, the λ-calculus has infinitely many variables with type a. We regard a language as determined when a set of types and constants are specified.

We give some notations for languages of the λ-calculus.

Definition. (Tuple variable)

A *tuple variable* with type is inductively defined as follows:

(1) every variable with type a is a tuple variable with type a;
(2) if α and β are tuple variables with type a and b, respectively, then $\langle\alpha, \beta\rangle$ is also a tuple variable with type $a \times b$.

Moreover variables that appear in a tuple variable must be all distinct.

Informally speaking, a tuple variable is an extension of a variable-list that occurs next to a LAMBDA operator in LISP.

Definition. (Term of the λ-calculus)

A *term* with type is inductively defined as follows:

(1) every variable with type a is a term with type a;
(2) every constant with type a is a term with type a;
(3) if M and N are terms with type b^a and a, respectively, then $(M^{a,b} \cdot N)$ is a term with type b;
(4) if α is a tuple variable with type a and M is a term with type b, then $(\lambda\alpha.M)$ is a term with type b^a.

When no confusion occurs, we write MN instead of $(M^{a,b} \cdot N)$.

Notations.

(i) The type of a term M is denoted by type (M).
(ii) An equation is $M=N$, where M and N are terms with the same type.
(iii) The set of all variables that occur in a tuple variable α is denoted by var(α).
(iv) A free variable in a term M is defined in the usual way, and the set of all free variables in M is denoted by $FV(M)$.
(v) In general, $\alpha, \beta, \gamma, \ldots$ are tuple variables, $M, M_1, M_2, \ldots, N, N_1, N_2, \ldots, L, \ldots$ are terms, and $x, x_1, x_2, \ldots, y, y_1, y_2, \ldots, z, \ldots$ are variables.
(vi) $\vec{x}$ and $\vec{N}$ are abbreviations of $x_1, \ldots, x_n$ and $N_1, \ldots, N_n$, respectively.
(vii) $M[\vec{N}/\vec{x}]$ is the term obtained from a term M by substituting each x_i for N_i simultaneously, where type (N_i)=type (x_i) $(1 \leq i \leq n)$. When a conflict of variables occurs (e.g. $(\lambda y.M)[Ny/x]$), we replace the bound variables by other variables.

Definitions. (Axiom system of the λ-calculus)

We define the axiom system of the λ-calculus.

(1) $M=M$,
(2) $M=N \Rightarrow N=M$,
(3) $L=M, M=N \Rightarrow L=N$,
(4) $M_1=M_2, N_1=N_2 \Rightarrow \langle M_1, N_1\rangle=\langle M_2, N_2\rangle$,
(5) $M_1=M_2, N_1=N_2 \Rightarrow M_1N_1=M_2N_2$,
(6) $M_1=M_2 \Rightarrow \lambda\alpha.M_1=\lambda\alpha.M_2$
(7) $\lambda\alpha.M=\lambda\alpha'.M[\vec{y}/\vec{x}]$,
where α' is $\alpha[\vec{y}/\vec{x}]$ and $\{\vec{y}\} \cap FV(\lambda\alpha.M)=\phi$,
(8) $(\lambda\alpha.M)(\alpha[\vec{N}/\vec{x}])=M[\vec{N}/\vec{x}]$,
(9) $M=N$ if M and N are terms with type 1,
(10) $\lambda\alpha.M\alpha=M$,
where var$(\alpha) \cap FV(M)=\phi$.

In the above equations, types of the terms are assumed to be consistently defined.

When an equation $M=N$ is deduced from the axiom system of the λ-calculus, we say that $M=N$ is provable in the λ-calculus and write $\vdash_\lambda M=N$.

3. The typed composition calculus

We now define the *typed composition calculus* (simply called the composition calculus). The composition calculus has no variables. In the λ-calculus defined in Section 2, each constant has a single type. On the other hand, in the composition calculus each constant has an ordered pair of types. The ordered pair of types a and b is written as $a \to b$. When a constant f has $a \to b$, we write $f:a \to b$. Intuitively $f:a \to b$ means that f is a function from a to b.

The composition calculus always has the following constants called the *special constants*: $id_a:a \to a$, $!_a:a \to 1$, $p^{a,b}:a \times b \to a$, $q^{a,b}:a \times b \to b$ and $ev^{a,b}:b^a \times a \to b$ for each type a and b. Here $p^{a,b}$ and $q^{a,b}$ intuitively mean projections from $a \times b$, and $ev^{a,b}$ corresponds to 'apply' in LISP or FFP. We intend $ev^{a,b}$ to have the meaning: $ev^{a,b}(\langle f, x\rangle)=f(x)$.

Definition. (Term of the composition calculus)

A term with an ordered pair of types is inductively defined as follows:

(1) every constant $f:a \to b$ is a term with the same ordered pair of types;
(2) if $f:a \to b$ and $g:b \to c$ are terms, then $g \circ f:a \to c$ is a term;
(3) if $f:c \to a$ and $g:c \to b$ are terms, then $\langle f, g\rangle:c \to a \times b$ is a term;
(4) if $h:c \times a \to b$ is a term, then $\Lambda_{c,a}(h):c \to b^a$ is a term.

Whenever no conflict occurs, we omit the subscripts c and a of $\Lambda_{c,a}$. In the above, $\langle f, g\rangle$ is the same as in FP, and $\Lambda(h)$ corresponds to the curried function of h, that is, $\Lambda(h)(x)(y)=h(\langle x, y\rangle)$. In FP(FFP), 'apply' corresponding to $ev^{a,b}$ appears, but the operation $\Lambda(-)$ is not included in the system.

Definition. (Axiom system of the composition calculus)

We define the axiom system of the composition

calculus.

I. (1) $s=s$,
(2) $s=t \Rightarrow t=s$,
(3) $s=t,\ t=u \Rightarrow s=u$,
(4) $s_1=s_2,\ t_1=t_2 \Rightarrow t_1 \circ s_1=t_2 \circ s_2$,
(5) $s_1=s_2,\ t_1=t_2 \Rightarrow \langle s_1, t_1\rangle=\langle s_2, t_2\rangle$,
(6) $s_1=s_2 \Rightarrow \Lambda(s_1)=\Lambda(s_2)$,

II. (7) $(s \circ t) \circ u=s \circ (t \circ u)$,
(8) $s \circ id_a=s$, $id_b \circ s=s$ (where $s:a \to b$),
(9) $!_a \circ s=!_b$ (where $s:b \to a$),
(10) $p^{a,b} \circ \langle s, t\rangle=s$, $q^{a,b} \circ \langle s, t\rangle=t$,
(11) $\langle s, t\rangle \circ u=\langle s \circ u, t \circ u\rangle$,
(12) $\langle p^{a,b}, q^{a,b}\rangle=id_{a \times b}$,
(13) $ev^{a,b} \circ \langle \Lambda(u) \circ s, t\rangle=u \circ \langle s, t\rangle$,
(14) $\Lambda(s) \circ t=\Lambda(s \circ \langle t \circ p^{d,a}, q^{d,a}\rangle)$
(where $s:c \times a \to b$ and $t:d \to c$),
(15) $\Lambda(ev^{a,b})=id_{b^a}$.

Here a pair of terms in each equation has the same type. In the above equations, ordered pairs of types are assumed to be consistently defined. When an equation $s=t$ can be deduced from the above axiom system, we say that $s=t$ is provable in the composition calculus and write $\vdash_c s=t$.

In the above axiom system, axioms (13) and (14) are rather complicated. The other axioms are simple and intuitive. We can check axiom (13), by applying a variable x to both terms of the axiom. We can informally calculate the following.

$$\begin{aligned}(ev^{a,b} \circ \langle \Lambda(u) \circ s, t\rangle)(x) &= ev^{a,b}(\langle \Lambda(u)(s(x)), t(x)\rangle)\\ &= \Lambda(u)(s(x))(t(x))\\ &= u(\langle s(x), t(x)\rangle)\\ &= (u \circ \langle s, t\rangle)(x)\end{aligned}$$

On the other hand, if we apply z and x to both terms of axiom (14), then

$$\begin{aligned}\Lambda(s \circ \langle t \circ p^{d,a}, q^{d,a}\rangle)(z)(x) &= (s \circ \langle t \circ p^{d,a}, q^{d,a}\rangle)(\langle z, x\rangle)\\ &= s(\langle t(z), x\rangle)\\ &= \Lambda(s)(t(z))(x)\\ &= (\Lambda(s) \circ t)(z)(x)\end{aligned}$$

4. Equivalence Theorem

We show that the λ-calculus and the composition calculus are essentially the same. First we define the translation rule $[-]$ that interprets a term of the λ-calculus into a term of the composition calculus. Next, conversely, we define the rule $(-)^*$ that translates a term of the composition calculus into a term of the λ-calculus. Finally we prove that both $[-]$ and $(-)^*$ preserve meanings of terms.

Definition.

Let L be a language of the λ-calculus. We define the language $C(L)$ of the composition calculus as follows:

(1) the types of $C(L)$ are just the same as of L, and
(2) k is a constant with type a of L *iff* $f_k:1 \to a$ of $C(L)$.

For each pair of a tuple variable γ and a term M, when $FV(M) \subset \mathrm{var}(\gamma)$, we inductively define the term $[\lambda\gamma.M]$:type $(\gamma) \to$ type (M) of $C(L)$ as follows.

(1) $[\lambda\gamma.k]=f_k \circ !_{\mathrm{type}(\gamma)}$,
where k is a constant of L,
(2) $[\lambda x.x]=id_{\mathrm{type}(x)}$,
(3) $[\lambda\langle\alpha, \beta\rangle.x]=[\lambda\alpha.x] \circ p^{a,b}$ if $x \in \mathrm{var}(\alpha)$, and
$[\lambda\langle\alpha, \beta\rangle.y]=[\lambda\beta.y] \circ q^{a,b}$ if $y \in \mathrm{var}(\beta)$,
where $a=$type (α) and $b=$type (β),
(4) $[\lambda\gamma.\langle M_1, M_2\rangle]=\langle[\lambda\gamma.M_1], [\lambda\gamma.M_2]\rangle$,
(5) $[\lambda\gamma.M_1M_2]=ev^{a,b} \circ \langle[\lambda\gamma.M_1], [\lambda\gamma.M_2]\rangle$,
where $b^a=$type (M_1) and $a=$type (M_2),
(6) $[\lambda\gamma.(\lambda\alpha.M_1)]=\Lambda_{c,a}([\lambda\langle\gamma, \alpha'\rangle.M_1'])$,
where $c=$type (γ) and $a=$type (α).

Here α' is the tuple variable obtained from α by replacing all the variables contained in $\mathrm{var}(\gamma)$ by new variables with the same types, and M_1' is also the term obtained from M_1 by replacing the variables contained in $\mathrm{var}(\gamma)$ by the corresponding variables in $\mathrm{var}(\alpha')$.

Whenever we mention $[\lambda\gamma.M]$, we assume that $FV(M) \subset \mathrm{var}(\gamma)$.

Definition.

Let L be a language of the composition calculus. We define the language $\lambda(L)$ of the λ-calculus as follows:

(1) the types of $\lambda(L)$ are the same as those of L, and
(2) $k:a \to b$ is a non-special constant of L *iff* c_k is a constant with type b^a of $\lambda(L)$.

For each term $t:a \to b$ of L, we define a closed term t^* with type b^a of L.

(1) $k^*=\lambda x.c_k x$ if k is a non-special constant of L,
(2) $(id_a)^*=\lambda x.x$,
(3) $(!_a)^*=\lambda x.\langle\ \rangle$,
where $\langle\ \rangle$ is the constant with type 1 defined in Section 2,
(4) $(p^{a,b})^*=\lambda\langle x, y\rangle.x$,
(5) $(q^{a,b})^*=\lambda\langle x, y\rangle.y$,
(6) $(ev^{a,b})^*=\lambda\langle v, x\rangle.vx$,
where type $(v)=b^a$ and type $(x)=a$,
(7) $(s \circ t)^*=\lambda x.s^*(t^*x)$,
(8) $(\langle s, t\rangle)^*=\lambda z.\langle s^*z, t^*z\rangle$,
(9) $(\Lambda_{c,a}(s))^*=\lambda z.(\lambda x.s^*\langle z, x\rangle)$,
where type $(z)=c$ and type $(x)=a$.

Equivalence Theorem.

(i) $\vdash_\lambda M=N \Leftrightarrow \vdash_c [\lambda\gamma.M]=[\lambda\gamma.N]$.
(ii) $\vdash_c s=t \Leftrightarrow \vdash_\lambda s^*=t^*$.
(iii) $\vdash_\lambda [\lambda\gamma.M]^*=\lambda\gamma.M$,
if all $c_{(f_k)}\langle\ \rangle=k$ are added to the axiom system, where k is a constant of the λ-calculus.
(iv) $\vdash_c [t^*]=t$,
if all $f_{(c_k)}=\Lambda(k \circ q^{1,a})$ are added to the axiom system, where $k:a \to b$ is a non-special constant of the composition calculus.

Proof. See Appendix.

Note that $\vdash_c [k^*]=k$ under the assumption: $f_{(c_k)}=\Lambda(k \circ q^{1,a})$. Indeed we can calculate as follows:

$$\begin{aligned}[k^*] &= [\lambda z.c_k z]\\ &= ev^{a,b} \circ \langle[\lambda z.c_k], [\lambda z.z]\rangle\\ &= ev^{a,b} \circ \langle\Lambda(k \circ q^{1,a}) \circ !_a, id_a\rangle\\ &= k \circ q^{1,a} \circ \langle !_a, id_a\rangle\\ &= k.\end{aligned}$$

The Equivalence Theorem shows that $[-]$ and $(-)^*$ preserve meanings of terms. The λ-calculus and the composition calculus are essentially the same.

5. Concluding Remarks

We have discussed functional programming languages of two types: applicative languages and compositional languages. To formalize them, we have defined the typed λ-calculus with product and the typed composition calculus. We have formally shown that both formal systems are essentially the same.

As mentioned in Section 1, the compositional languages are defined without variables. This means that the compositional languages are naturally formalized as algebras. Indeed the composition calculus discussed in this paper can be regarded as an algebra. Backus introduced the FP algebra, which is used for examining properties of programs. It is known that the λ-calculus can also be defined without variables by use of the combinators K and S[4]. This mechanism is used effectively in applicative programming languages. The composition calculus is another method of deleting variables.

The arguments we have discussed are based on results found in working with the λ-calcalculus theory. Recently models of the λ-calculus have been examined in the light of category theory. It is known that models of the λ-calculus are closely related to *cartesian closed categories*. The composition calculus is coincident with cartesian closed categories. See [6], [7] and [4]. In particular, we refer the reader to [6].

In this paper we have dealt with typed languages. But all arguments can be immediately translated to type-free languages. If we regard all types besides 1 as being identical, then the entire discussion including the Equivalence Theorem still holds without any modifications.

References

1. BACKUS, J. Can Programming Be Liberated from the von Neuman Style? A Functional Style and Its Algebra of Programs, *CACM* **21**, 8, (1978), 613-614.
2. BACKUS, J. The Algebra of Functional Programs: Function Level Reasoning, Linear Equations, and Extended Definitions, *Proc. International Colloquium on the Formalization of Programming Concepts, Lecture Notes in Computer Science*, **107**, Springer-Verlag, (1981).
3. BACKUS, J. Functional Level Programs as Mathematical Objects, *Proc. Functional Programming languages* and *Computer Architecture*, (1980), 1-10.
4. BARENDREGT, H. The Lambda Calculus—Its Syntax and Semantics, Studies in Logic 103, Second Edition, North-Holland (1984).
5. GORDON, M. J., A. J. MILNER and C. P. WORDSWORTH Edinburgh LCF, *Lecture Notes in Computer Science*, 78, Springer-Verlag, (1979).
6. KOYMANS, C. P. J. Models of the Lambda Calculus, *Information* and *Control*, **52**, (1982), 306-332.
7. KOYMANS, C. P. J. Models of the Lambda Calculus, Ph. D Thesis, University of Utrecht, (1984).
8. TURNER, D. A. The Semantic Elegance of Applicative Languages, *Proc. Functional Programming Languages* and *Computer Architecture*, (1980), 85-93.
9. TURNER, D. A. Recursive Equations as a Programming Language, in Functional Programming and its Applications, Ed. J. Darington et al., Cambridge University Press, (1979).

Appendix. Proof of Equivalence Theorem

Lemma.

(i) $[\lambda\beta.\gamma] \circ [\lambda\alpha.\beta] = [\lambda\alpha.\gamma]$.

(ii) $[\lambda\langle\alpha, \beta\rangle.\gamma] = [\lambda\alpha.\gamma] \circ p^{a,b}$ if var $(\gamma) \subset$ var (α), and $[\lambda\langle\alpha, \beta\rangle.\gamma] = [\lambda\beta.\gamma] \circ q^{a,b}$ if var $(\gamma) \subset$ var (β), where a = type (α) and b = type (β).

(iii) $[\lambda\alpha.M] = [\lambda\alpha'.M[\vec{y}/\vec{x}]]$, where $\alpha' = \alpha[\vec{y}/\vec{x}]$.

(iv) $[\lambda\alpha.M] \circ [\lambda\beta.\alpha] = [\lambda\beta.M]$.

(v) $[\lambda\alpha.M] \circ [\lambda\beta.\alpha[\vec{N}/\vec{x}]] = [\lambda\beta.M[\vec{N}/\vec{x}]]$.

Proof. (i)-(iii). Easy.

(iv) We use induction on the structure of M. Consider the case that M is $\lambda\gamma.M'$. By (iii) we can assume that var $(\alpha) \cap$ var $(\gamma) = \phi$ and var $(\beta) \cap$ var $(\gamma) = \phi$.

$[\lambda\alpha.(\lambda\gamma.M')] \circ [\lambda\beta.\alpha]$
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M']) \circ [\lambda\beta.\alpha]$
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ \langle[\lambda\beta.\alpha] \circ p^{b,c}, q^{b,c}\rangle)$
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ [\lambda\langle\alpha, \gamma\rangle.\langle\alpha, \gamma\rangle] \circ \langle[\lambda\beta.\alpha] \circ p^{b,c}, q^{b,c}\rangle)$
(by the induction hypothesis)
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ \langle[\lambda\alpha.\alpha] \circ [\lambda\beta.\alpha] \circ p^{b,c}, [\lambda\gamma.\gamma] \circ q^{b,c}\rangle)$
(by (ii))
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ \langle[\lambda\beta.\alpha] \circ p^{b,c}, [\lambda\gamma.\gamma] \circ q^{b,c}\rangle)$
(by (i))
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ [\lambda\langle\beta, \gamma\rangle.\langle\alpha, \gamma\rangle])$
(by (ii))
$= \Lambda([\lambda\langle\beta, \gamma\rangle.M'])$
(by the induction hypothesis)
$= [\lambda\beta.(\lambda\gamma.M')]$

Here b = type (β) and c = type (γ). The rest is clear.

(v) We use induction on the structure of M. When M is $\lambda\gamma.M'$,

$[\lambda\alpha.(\lambda\gamma.M')] \circ [\lambda\beta.\alpha[\vec{N}/\vec{x}]]$
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M']) \circ [\lambda\beta.\alpha[\vec{N}/\vec{x}]]$
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ \langle[\lambda\beta.\alpha[\vec{N}/\vec{x}]] \circ p^{b,c}, q^{b,c}\rangle)$
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ [\lambda\langle\alpha, \gamma\rangle.\langle\alpha, \gamma\rangle] \circ \langle[\lambda\beta.\alpha[\vec{N}/\vec{x}]] \circ [\lambda\beta.\beta] \circ p^{b,c}, q^{b,c}\rangle)$
(by (iv))
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ \langle[\lambda\alpha.\alpha] \circ [\lambda\beta.\alpha[\vec{N}/\vec{x}]] \circ [\lambda\beta.\beta] \circ p^{b,c}, [\lambda\gamma.\gamma] \circ q^{b,c}\rangle)$
(by (ii))
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ \langle[\lambda\alpha.\alpha] \circ [\lambda\beta.\alpha[\vec{N}/\vec{x}]] \circ [\lambda\langle\beta, \gamma\rangle.\beta], [\lambda\langle\beta, \gamma\rangle.\gamma]\rangle)$
(by (ii))
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ \langle[\lambda\alpha.\alpha] \circ [\lambda\langle\beta, \gamma\rangle.\alpha[\vec{N}/\vec{x}]], [\lambda\gamma.\gamma] \circ [\lambda\langle\beta, \gamma\rangle.\gamma]\rangle)$
(by (iv) and (i))
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ [\lambda\langle\alpha, \gamma\rangle.\langle\alpha, \gamma\rangle] \circ [\lambda\langle\beta, \gamma\rangle.\langle\alpha[\vec{N}/\vec{x}], \gamma\rangle])$
(by (ii))
$= \Lambda([\lambda\langle\alpha, \gamma\rangle.M'] \circ [\lambda\langle\beta, \gamma\rangle.\langle\alpha[\vec{N}/\vec{x}], \gamma\rangle])$
(by (iv))
$= \Lambda([\lambda\langle\beta, \gamma\rangle.M'[\vec{N}/\vec{x}]])$

(by the induction hypothesis)
$=[\lambda\beta.(\lambda\gamma.M')[\vec{N}/\vec{x}]]$
Here $b=$type (β) and $c=$type (γ). The rest is clear.

Proof of Equivalence Theorem

The essential point of the proof is the only-if part of (i). The other parts are easily proved. To show the only-if part of (i), we use induction on the length of the proof of $M=N$. We only treat the essential axioms (8) and (10) of the λ-calculus. The other cases are clear.

Axiom (8) $(\lambda\alpha.M)(\alpha[\vec{N}/\vec{x}])=M[\vec{N}/\vec{x}]$.

$[\lambda\gamma.(\lambda\alpha.M)(\alpha[\vec{N}/\vec{x}])]$
$=ev^{a,b}\circ\langle\Lambda([\langle\gamma,\alpha\rangle.M]),[\lambda\gamma.\alpha[\vec{N}/\vec{x}]]\rangle$
$=[\lambda\langle\gamma,\alpha\rangle.M]\circ\langle id_c,[\lambda\gamma.\alpha[\vec{N}/\vec{x}]]\rangle$
$=[\lambda\langle\gamma,\alpha\rangle.M]\circ[\lambda\langle\gamma,\alpha\rangle.\langle\gamma,\alpha\rangle]$
$\circ\langle id_c,[\lambda\gamma.\alpha[\vec{N}/\vec{x}]]\rangle$
(by Lemma (iv))
$=[\lambda\langle\gamma,\alpha\rangle.M]\circ[\lambda\langle\gamma,\alpha\rangle.\langle\gamma,\alpha\rangle]$
$\circ\langle[\lambda\gamma.\gamma]\circ p^{c,a},q^{c,a}\rangle\circ\langle id_c,[\lambda\gamma.\alpha[\vec{N}/\vec{x}]]\rangle$
(using Lemma (i) and (ii))
$=[\lambda\langle\gamma,\alpha\rangle.M]\circ\langle[\lambda\gamma.\gamma]\circ p^{c,a},q^{c,a}\rangle$
$\circ\langle id_c,[\lambda\gamma.\alpha[\vec{N}/\vec{x}]]\rangle$
(by Lemma (iv))
$=[\lambda\langle\gamma,\alpha\rangle.M]\circ[\lambda\gamma.\langle\gamma,\alpha[\vec{N}/\vec{x}]\rangle]$
$=[\lambda\gamma.M[\vec{N}/\vec{x}]]$
(by Lemma (v))

Here $a=$type (α), $b=$type (M) and $c=$type (γ).

Axiom (10) $\lambda\alpha.M\alpha=M$.

$[\lambda\gamma.(\lambda\alpha.M\alpha)]$
$=\Lambda([\lambda\langle\gamma,\alpha\rangle.M\alpha])$
$=\Lambda(ev^{a,b}\circ\langle[\lambda\langle\gamma,\alpha\rangle.M],[\lambda\langle\gamma,\alpha\rangle.\alpha]\rangle)$
$=\Lambda(ev^{a,b}\circ\langle[\lambda\gamma.M]\circ[\lambda\langle\gamma,\alpha\rangle.\gamma],$
$[\lambda\langle\gamma,\alpha\rangle.\alpha]\rangle)$
(by Lemma (iv))
(Note that var $(\alpha)\cap FV(M)=\phi$.)
$=\Lambda(ev^{a,b}\circ\langle[\lambda\gamma.M]\circ[\lambda\gamma.\gamma]\circ p^{c,a},$
$[\lambda\alpha.\alpha]\circ q^{c,a}\rangle)$
$=\Lambda(ev^{a,b}\circ\langle[\lambda\gamma.M]\circ p^{c,a},[\lambda\alpha.\alpha]\circ q^{c,a}\rangle)$
(by Lemma (iv))
$=\Lambda(ev^{a,b}\circ\langle[\lambda\gamma.M]\circ p^{c,a},q^{c,a}\rangle)$
(by Axiom (12) of the composition calculus)
$=\Lambda(ev^{a,b})\circ[\lambda\gamma.M]$
$=[\lambda\gamma.M]$
(by Axiom (15) of the composition calculus)

Here $a=$type (α), $b=$type (M) and $c=$type (γ).

(Received June 10, 1985; revised November 12, 1985)