Regular Paper

Verification of Substitution Theorem Using HOL

Takayuki Koai[1,a)] Makoto Tatsuta[1,b)]

Received: October 3, 2011, Accepted: January 16, 2012

Abstract: Substitution Theorem is a new theorem in untyped lambda calculus, which was proved in 2006. This theorem states that for a given lambda term and given free variables in it, the term becomes weakly normalizing when we substitute arbitrary weakly normalizing terms for these free variables, if the term becomes weakly normalizing when we substitute a single arbitrary weakly normalizing term for these free variables. This paper formalizes and verifies this theorem by using the higher-order theorem prover HOL. A control path, which is the key notion in the proof, explicitly uses names of bound variables in lambda terms, and it is defined only for lambda terms without bound variable renaming. The lambda terms without bound variable renaming are formalized by using the HOL package based on contextual alpha-equivalence. The verification uses 10119 lines of HOL code and 326 lemmas.

Keywords: lambda calculus, theorem proving, HOL, Substitution Theorem

1. Introduction

Automated theorem proving has been actively studied. For example, the verification of the four color theorem by Coq was a surprising result [3]. The theorem provers such as Coq, Agda and HOL have been intensively developed. Its applications for hardware verification and software verification has been widely discussed.

Theorem proving for theorems in λ-calculus has been also actively studied. Church-Rosser theorem was verified by Ref. [6]. Several properties in pure type systems were formalized and verified by Ref. [5]. The contextual α-equivalence was proposed by Ref. [4] to formalize the α-conversion in λ-calculus in a better way. Nominal logic was introduced by Ref. [9] in order to give a better formalization of λ-calculus and he verified several properties including Church-Rosser theorem.

Substitution Theorem is a new theorem for untyped λ-calculus, which was proved in 2006 [7]. This theorem states that for a given λ-term and given free variables in it, the term becomes weakly normalizing when we substitute arbitrary weakly normalizing terms for these free variables, if the term becomes weakly normalizing when we substitute a single arbitrary weakly normalizing term for these free variables. The statement of the theorem is simple, but its proof uses complicated techniques that are hidden in the statement. Substitution Theorem is a part of deep results given in Ref. [7]. Since this theorem is new and significant, it is worth to verify.

In this paper we formalize and verify Substitution Theorem by using higher-order theorem prover HOL. We faithfully formalize mathematical content in Ref. [7]. To handle λ-terms without renaming bound variables, we use the contextual α-equivalence with the package in Ref. [4].

[1] National Institute of Informatics, Chiyoda, Tokyo 101–8430, Japan
[a)] koai@nii.ac.jp
[b)] tatsuta@nii.ac.jp

There have been several results for verification of properties of λ-terms without bound variable renaming [4], [5], [6], [9]. They developed new methods for handling α-equivalence and applied them to verification of several properties of λ-terms without bound variable renaming. However, most of them verified only well-known theorems such as Church-Rosser theorem. On the other hand we verify a new and significant theorem using λ-terms without bound variable renaming.

Our verification shows that the idea of the contextual α-equivalence given in Ref. [4] and the HOL package based on it are actually good at formalizing λ-terms without bound variable renaming. In Ref. [4] he applied his idea to verification of Church Rosser theorem. However, there have not been other results for application of his idea. In this paper, we use his HOL package for verification of a new and significant theorem, and it shows that his idea works well.

We have six challenges in our verification. First, we formalize and verify a new and significant theorem. Since untyped λ-calculus has been fully developed and it is difficult to obtain a new theorem, a new theorem in this field is often subtle and difficult. A significant theorem often requires difficult proofs. Indeed the proof of Substitution Theorem is complicated and uses subtle notions and difficult techniques.

Secondly, we verify proofs that require λ-terms without bound variable renaming as well as λ-terms with bound variable renaming. The key notion in the proof of Substitution Theorem is a control path, which is defined by using names of bound variables. Hence our formalization has to handle λ-terms without bound variable renaming.

Thirdly, we introduce the notion PWN_S for a set S of bound variable names, in order to solve difficulty of variable capturing. This replaces persistently weak normalization used in Ref. [7], and simplifies substitution caused by β-reduction when we show that some term has this property. Unless this notion, our formal proof would be much more complicated.

Fourthly, we use the technique of choosing an appropriate induction variable to verify the preservation of control paths by substitution. This property was implicitly used without proofs in Ref. [7], but its formal proof is difficult, because a control path is a notion defined for λ-terms without bound variable renaming, on the other hand substitution is a notion defined for λ-terms with bound variable renaming, and hence the formal statement of this property uses both styles of λ-terms in a mixed way. In order to solve this difficulty, we move an inner variable hidden in the statement to the top level and use it as our induction variable.

Fifthly, we introduce some predicate to express the negation of the existence of adjacent control paths. A faithful formalization of the existence of adjacent control paths does not correctly represent its implicit free variable condition. When we use its negation in assumptions, a formal proof cannot go further for this reason. In order to solve this problem, we introduce some predicate to express its negation so that it correctly represents the free variable condition.

Sixthly, we formalize the notion of adjacent control paths for ordinary λ-terms with α-conversion. Both the definition of control paths and the definition of adjacent variable occurrences require bound variable names. However we find that we can directly define adjacent control paths without using them. This definition works for the proof of Lemma 2.8 and simplifies our formalization.

For this verification of the proof given in Ref. [7], we use 10119 lines of HOL code and 326 lemmas Our HOL source code is available at http://research.nii.ac.jp/˜tatsuta/hol.

Section 2 describes the mathematical statement of Substitution Theorem and its mathematical proof as well as basic definitions for λ-terms. Section 3 explains the contextual α-equivalence. Section 4 gives our formalization of the theorem, and Section 5 discusses its formal proofs. Section 6 explains our challenges in this work, and Section 7 concludes.

2. Substitution Theorem

In this section, we give the statement of Substitution Theorem and its complete proof, which are taken from Ref. [7]. This section explains only mathematical contents, and all of them will be formalized after Section 4.

Substitution theorem is interesting, since it implies that if MXX is weakly normalizing for all weakly normalizing term X, then MXY is also weakly normalizing for all weakly normalizing terms X, Y.

We assume the standard definitions of λ-calculus and β-reduction. We use $x, y, z, w, t, u, v, \ldots$ to range over variables, and $M, N, L, P, X, Y, \ldots$ to range over λ-terms. We define Λ as the set of λ-terms. We use vector notations. $\vec{M}$ denotes the sequence $M_1, M_2, \ldots, M_n$ for $n \geq 0$ and $\vec{M}$ is empty when $n = 0$. $\lambda\overrightarrow{x}.M\overrightarrow{N}$ is an abbreviation for $\lambda x_1 \ldots x_n.MN_1 \ldots N_m$ where $\overrightarrow{x}$ is $x_1, \ldots, x_n$ (n could be 0). $\overrightarrow{N}$ is $N_1, \ldots, N_m$. $\vec{M} \in C$ means $N \in C$ for all $N \in \vec{M}$ where C is a set of λ-terms. By $M[x_1 := N_1, \ldots, x_n := N_n]$ and $M[\overrightarrow{x} := \overrightarrow{N}]$ we denote simultaneous capture-free substitutions. We use lh( ) to denote the vector length. For reduction $\rightarrow_\beta$ is one step of β-reduction and $\rightarrow_\beta^*$ is the transitive reflexive closure of the relation $\rightarrow_\beta$. We use = to denote syntactical equality. We use NF to denote the set of β-normal forms. We assume the Barendregt convention [1] (page 26), that is, we assume that all names of bound variables are different from each other and different from those of the free variables.

Bound variable renaming is a transformation of a λ-term by renaming bound variables in it. For example, $\lambda x.xxy$ is transformed into $\lambda w.wwy$ by bound variable renaming. Bound variable renaming is also called *α-conversion*. *Variable capturing* is the following situation: when we substitute a λ-term t for some variable in λ-term u and obtain u', some free variable in t becomes bound variables in u'. For example, when we substitute zx for y in $\lambda x.xxy$, we get $\lambda x.xx(zx)$, where x in the term zx is a free variable of zx, and x in the subterm zx of the term $\lambda x.xx(zx)$ is a bound variable of the term $\lambda x.xx(zx)$. Since the substitution with variable capturing is not meaningful, one usually uses bound variable renaming before substitution to avoid variable capturing. For example, first one transforms $\lambda x.xxy$ into $\lambda w.wwy$, and then one substitutes zx for y in $\lambda w.wwy$ to get $\lambda w.ww(zx)$.

λ-term is *weakly normalizing* if it reduces to some β-normal form. Let WN be the *set of weakly normalizing* λ-terms.

Definition 2.1 We say a λ-term M is *persistently weakly normalizing* if for all n and for all $X_1, \ldots, X_n \in$ WN we get $MX_1 \ldots X_n \in$ WN. We denote the set of persistently weakly normalizing λ-terms by PWN.

The next is our main theorem in this section.

Theorem 2.2 (Substitution Theorem for WN) *If $M[x_i := X, x_j := X] \in WN$ for all $X \in WN$ and for all i, j $(1 \leq i, j \leq n)$, then $M[x_1 := X_1, \ldots, x_n := X_n] \in WN$ for all $X_1, \ldots, X_n \in WN$.*

By taking n to be 2 in this theorem, we can show that $\forall X \in$ WN($MXX \in$ WN) implies $\forall XY \in$ WN($MXY \in$ WN).

A useful relation between WN and PWN is that the application of a λ-term in WN to a λ-term in PWN is a λ-term in WN.

Lemma 2.3 *If $M \in WN$ and $N \in PWN$ then $MN \in WN$.*

This is proved by induction on M.

We introduce the notions of *control path* and *adjacent control paths*.

When we define control paths and adjacent variable occurrences, we do not allow α-conversion on β-normal forms: this helps us to denote bound variable occurrences by their names. Although we use these conventions in our proofs, our results hold also for usual λ-calculus with α-conversion.

Definition 2.4 (Control Paths) Assume that M is in NF, x and y are variable occurrences in M, where they may be bound. The relation $x \leadsto_1 y$ in M is defined to hold if M has the subterm $x\overrightarrow{N}(\lambda\overrightarrow{y}.L)$ where the variable occurrence x is the one explicitly shown in this subterm, the variable occurrence y is in L, and the variable y is in $\overrightarrow{y}$.

A control path in M is defined as the sequence $(x_1, \ldots, x_n)$ of variable occurrences in M where $n \geq 1$, $x_i \leadsto_1 x_{i+1}$ in M for $1 \leq i < n$, and x_1 is a free variable occurrence in M. For a set $\mathcal{S}$ of free variable occurrences of M, we say a control path from $\mathcal{S}$ in M if it is the control path $(x_1, \ldots, x_n)$ in M for some $x_1 \in \mathcal{S}$.

The relation $x \leadsto y$ in M is defined to hold if there exists some control path $(x, x_1, \ldots, x_n, y)$ in M.

Note that the relation $x \leadsto y$ in M is the reflexive transitive closure of $z \leadsto_1 w$ in M with x free in M. For simplicity, we will often use $x \rightsquigarrow y$ to denote $(x, x_1, \ldots, x_n, y)$.

Example 2.5 Let $N = \lambda w.x(\lambda v.vvy)(\lambda t.t(\lambda uz.xz))$. We have $x \rightsquigarrow x$ and $x \rightsquigarrow v$. Moreover we have $x \rightsquigarrow z$ since $x \rightsquigarrow_1 t$ and $t \rightsquigarrow_1 z$ hold and its control path is (x, t, z). All the relations $x \rightsquigarrow_1 y$ in N are: $x \rightsquigarrow_1 v$, $x \rightsquigarrow_1 t$, and $t \rightsquigarrow_1 z$.

Definition 2.6 (Adjacent Control Paths) Suppose M is in NF and x and y are two variable occurrences in M where they may be bound and their variables may be the same.
The variable occurrences x and y are called adjacent in M if M has the subterm $x\overrightarrow{N}(\lambda\overrightarrow{z}.y\overrightarrow{L})$ where the variable occurrences x and y are those explicitly shown in this subterm.
Two control paths $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_m)$ in M are called adjacent control paths in M if the variable occurrences x_n and y_m are adjacent in M.

Example 2.7 Let $N = \lambda w.x(\lambda v.vvy)(\lambda t.t(\lambda uz.xz))$. v and y are adjacent, so $x \rightsquigarrow v$ and $y \rightsquigarrow y$ are adjacent control paths. Moreover the first occurrence v and the second occurrence v in N are adjacent, so $x \rightsquigarrow v$ and $x \rightsquigarrow v$ are adjacent control paths. The other adjacent control paths in N are represented by: $x \rightsquigarrow x$ and $x \rightsquigarrow v$; $x \rightsquigarrow x$ and $x \rightsquigarrow t$; $x \rightsquigarrow t$ and $x \rightsquigarrow x$; $x \rightsquigarrow x$ and $x \rightsquigarrow z$.

Lemma A13 in Ref. [2] gave the following lemma.

Lemma 2.8 *If there are adjacent control paths from x in $N \in NF$ then there is $X \in WN$ such that $N[x := X] \notin WN$.*

We can show that some weak normalization condition implies the non-existence of adjacent control paths.

Lemma 2.9 *If N is in NF and for all $X \in WN$ we have $N[x := X, y := X] \in WN$, then there are no adjacent control paths from $\{x, y\}$ in N.*

Proof. To show its contraposition, assume that there are adjacent control paths from x, y in N. By letting N be $N[y := x]$ in Lemma 2.8, since there are adjacent control paths from x in $N[y := x]$, we have $X \notin$ WN such that $N[y := x][x := X] \notin$ WN, that is, $N[x := X, y := X] \notin$ WN. □

The key lemma for proving Substitution Theorem is the reverse of Lemma 2.9.

Lemma 2.10 (Key Lemma) *If there are no adjacent control paths from $\vec{x}$ in $N \in NF$, then $N[\vec{x} := \vec{X}] \in WN$ for all $\vec{X} \in WN$.*

Proof. The proof is by induction on N. Let $\vec{x} = x_1, \ldots, x_n$ and $N = \lambda\vec{y}.z\vec{L}$. We will write U' as short for $U[\overrightarrow{x} := \overrightarrow{X}]$.

Case 1. $z \notin \vec{x}$. We have $N' = \lambda\vec{y}.z\vec{L}'$. Since there are no adjacent control paths from $\vec{x}$ in L_i $(1 \le i \le \mathrm{lh}(\vec{L}))$, by induction hypothesis we have $L_i' \in$ WN. So N' is in WN.

Case 2. $z = x_h$ $(1 \le h \le n)$. Suppose $L_l = \lambda\vec{u}.v\vec{P} \in \vec{L}$ $(1 \le l \le lh(\vec{L}))$ with $lh(\vec{u}) = m$. We can observe that v is free in L_l. Otherwise $x_h \rightsquigarrow x_h$ and $x_h \rightsquigarrow v$ would be adjacent control paths from $\{x_h\}$ in N. Moreover there are no adjacent control paths from $\{u_i, u_j\}$ $(1 \le i, j \le m)$ in $v\vec{P}$. Otherwise we would have $u_i \rightsquigarrow a$ and $u_j \rightsquigarrow b$ for some adjacent a, b in $v\vec{P}$ and $x_h \rightsquigarrow a$ and $x_h \rightsquigarrow b$ would be adjacent control paths from $\{x_h\}$ in N, since $x_h \rightsquigarrow u_i$ and $x_h \rightsquigarrow u_j$. Lastly there are no adjacent control paths from $\{x_i, u_j\}$ $(1 \le i \le n, 1 \le j \le m)$ in $v\vec{P}$. Otherwise we would have $x_i \rightsquigarrow a$ and $u_j \rightsquigarrow b$ for some adjacent a, b in $v\vec{P}$ and $x_i \rightsquigarrow a$ and $x_h \rightsquigarrow b$ would be adjacent control paths from $\{x_i, x_h\}$ in N, since $x_h \rightsquigarrow u_j$. Thus there are no adjacent control paths from $\vec{x}, \vec{u}$ in $v\vec{P}$. Therefore by induction hypothesis for all $X_1, \ldots, X_n, U_1, \ldots, U_m \in$ WN we get $v\vec{P}[\vec{x} := \vec{X}, \vec{u} := \vec{U}] \in$ WN. Then we get $L_l' \in$ PWN, since $L_l'U_1 \ldots U_m V_1 \ldots V_k$ reduces to $v\vec{P}[\vec{x} := \vec{X}, \vec{u} := \vec{U}]\vec{V} \in$ WN for all $U_1, \ldots, U_m, V_1, \ldots, V_k \in$ WN. From Lemma 2.3, we get $X_h\vec{L}' \in$ WN. Hence we have $N' = \lambda\vec{y}.X_h\overrightarrow{L'} \in$ WN for all n and all $X_1, \ldots, X_n \in$ WN. □

We can now show the Substitution Theorem.

Proof of Theorem 2.2. (1) Let $\overrightarrow{x} = x_1, \ldots, x_n$ and N be the β-normal form of M. By Lemma 2.9 there are no adjacent control paths from any $\{x_i, x_j\}$ $(1 \le i, j \le n)$ in N. Hence there are no adjacent control paths from $\overrightarrow{x}$. By Lemma 2.10 we get $N[\vec{x} := \vec{X}] \in$ WN for all $\overrightarrow{X} \in$ WN. Since $M[\vec{x} := \vec{X}]$ reduces to $N[\vec{x} := \vec{X}]$, we have $M[\vec{x} := \vec{X}] \in$ WN for all $\overrightarrow{X} \in$ WN. □

3. Contextual α-Equivalence

This section mathematically describes the contextual α-equivalence, pre-λ-calculus, and pure λ-calculus. They are taken from Ref. [4]. Their formalization will be given after Section 4.

In order to handle λ-calculus without bound variable renaming, in Ref. [4] he gave the contextual α-equivalence as well as two kinds of λ-calculi. One is the pre-λ-calculus and the other is the pure λ-calculus. The pre-λ-calculus handles λ-terms without renaming variables. The contextual α-equivalence is defined for pre-λ-calculus in order to defined the α-equivalence. The pure λ-calculus is the quotient of the pre-λ-calculus by the α-equivalence.

The pre-λ-calculus can handle a control path as well as other notions that do not assume bound variable renaming. On the other hand, the pure λ-calculus can handle substitution and the β-reduction.

Definition 3.1 (Pre-λ-Calculus) We use c for constants, and v for variables. We define terms t of pre-λ-calculus by $t ::= c|v|tt|\lambda v.t$. We will use Λ_1 to denote the set of terms of pre-λ-calculus.

The context α-equivalence enables us to define the α-equivalence. We use ($a \ =_{\mathrm{def}} \ b$) to say that a is defined as b.

Definition 3.2 (α-Equivalence) The contextual α-equivalence $x\,{}^{xs}_{\mathrm{var}}\!\equiv^{ys}_{\alpha} y$ for variables x, y and variable lists xs, ys is defined as follows.

$$w\,{}^{x::xs}_{\mathrm{var}}\!\equiv^{y::ys}_{\alpha} z \quad =_{\mathrm{def}}$$
$$(w = x \wedge z = y \wedge \|xs\| = \|ys\|) \vee (w \neq x \wedge z \neq y \wedge w\,{}^{xs}_{\mathrm{var}}\!\equiv^{ys}_{\alpha} z),$$
$$w\,{}^{[]}_{\mathrm{var}}\!\equiv^{[]}_{\alpha} z \quad =_{\mathrm{def}} \quad (w = z).$$

The contextual α-equivalence $t\ {}^{xs}\!\equiv^{ys}_{\alpha} u$ for terms t, u and variable lists xs, ys is defined as follows.

$$x\ {}^{xs}\!\equiv^{ys}_{\alpha} y \quad =_{\mathrm{def}} \quad x\,{}^{xs}_{\mathrm{var}}\!\equiv^{ys}_{\alpha} y,$$
$$t_1u_1\ {}^{xs}\!\equiv^{ys}_{\alpha} t_2u_2 \quad =_{\mathrm{def}} \quad t_1\ {}^{xs}\!\equiv^{ys}_{\alpha} t_2 \wedge u_1\ {}^{xs}\!\equiv^{ys}_{\alpha} u_2,$$
$$\lambda x.t_1\ {}^{xs}\!\equiv^{ys}_{\alpha} \lambda y.t_2 \quad =_{\mathrm{def}} \quad t_1\ {}^{x::xs}\!\equiv^{y::ys}_{\alpha} t_2.$$

The α-equivalence $t \equiv_{\alpha} u$ for terms t, u is defined as follows.

$$t \equiv_{\alpha} u \quad =_{\mathrm{def}} \quad t\ {}^{[]}\!\equiv^{[]}_{\alpha} u.$$

The relation $t_1\ {}^{xs}\!\equiv^{ys}_{\alpha} t_2$ means that t_1 is α-equivalent to t_2 when each variable in the list xs is replaced by the corresponding variable in the list ys.

Definition 3.3 (Pure λ-Calculus) We define the set Λ of

terms of pure λ-calculus as the quotient $\Lambda_1/\equiv_\alpha$ of the set of terms of pre-λ-calculus by the α-equivalence relation.
We use $\lfloor t \rfloor$ to denote the equivalence class of t. We assume a fixed representative for each equivalence class. We use $\lceil T \rceil$ to denote the representative of the equivalence class T.

4. Formalization

This section formalizes Sections 2 and 3. We will give formalization of the goal formula, λ-calculus, and control paths.

4.1 Goal Formula

This subsection will give our formalization of the statement of Substitution Theorem.

We faithfully formalize all the mathematical contents given in Section 2. This means that our work verifies the whole proof of Substitution Theorem for weak normalization given in Ref. [7]. We have also verified Lemma 2.8 except basic properties of λ-calculus, which are Lemmas 2, 6, 11, 12, 13, 15, 16, and 18 of Section 2, and the lemmas of Sections 4 and 5 except Lemma 41 in Ref. [8]. The formalization of the proof of Lemma 2.8, which is Lemma A13 of Ref. [2], is given in Ref. [8] and the details of its verification will be fully discussed in another paper.

We give the statement of Substitution Theorem (Theorem 2.2) for comparing it with its formalization.

Substitution Theorem for WN: If $M[x_i := X, x_j := X] \in \mathrm{WN}$ for all $X \in \mathrm{WN}$ and for all i, j $(1 \le i, j \le n)$, then $M[x_1 := X_1, \ldots, x_n := X_n] \in \mathrm{WN}$ for all $X_1, \ldots, X_n \in \mathrm{WN}$.

We formalize it in HOL as follows.

```
g ‘     (~(xl = ([]:(var list)))) ==>
        ((LIST_TO_SET xl) SUBSET (FV M)) ==>
        (CF M) ==>
        (!X xi xj.
           CF X /\ WN X /\ MEM xi xl /\ MEM xj xl ==>
           WN (M <[ [(xi,X); (xj,X)])) ==>
        !XL. EVERY WN XL /\ (LENGTH xl = LENGTH XL) ==>
           WN (M <[ (ZIP(xl,XL)))‘;
```

It has the following meaning. We assume the list of variables `xl` is not empty, the set of members of `xl` is a subset of the free variables of the λ-term `M`, and `M` does not contain any constants. Assume that for any λ-term `X`, any variables `xi` and `xj`, if `X` does not contain any constants and is weakly normalizing, and `xi` and `xj` are members of `xl`, then the λ-term obtained from `M` by substituting `X` for `xi` and `xj` in `M` is weakly normalizing. For every list `XL` of λ-terms, if all members of `XL` are weakly normalizing and the lengths of `xl` and `XL` are the same, then the λ-term obtained from `M` by substituting each member of `XL` for the corresponding member of `xl` in `M` is weakly normalizing.

The code `(LIST_TO_SET xl)` represents the set of the members of the list `xl`. The code `(FV M)` represents the set of the free variables of λ-term `M`. The code `(s SUBSET t)` represents that the set `s` is a subset of `t`. The code `(CF M)` represents that λ-term `M` does not contain any constants. The code `(WN X)` represents that λ-term `X` is weakly normalizing. The code `(M <[ [(xi,X); (xj,X)])` represents the substitution which replaces `xi` with `X` and `xj` with `X` in `M`. The code `(M <[ (ZIP(xl,XL)))` represents the substitution which replaces each member of `xl` with the corresponding member of `XL`.

4.2 λ-Calculus

In this subsection, we will give formalization of pre-λ-calculus, pure λ-calculus, and β-reduction. This formalization is taken from the HOL package by Ref. [4].

The λ-terms in pre-λ-calculus are defined in HOL as follows. They have the type `term1` with constructors `Con1`, `Var1`, `App1` and `Lam1`.

```
val _ = Hol_datatype
        ‘ term1 = Con1 of ’a
                | Var1 of var
                | App1 of term1 => term1
                | Lam1 of var => term1 ‘ ;
```

The code `(ALPHA N M)` represents the α-equivalence $N \equiv_\alpha M$ for pre-λ terms N and M. The theorem `ALPHA_EQUIV` says that `ALPHA` is an equivalence relation.

```
val ALPHA_EQUIV = |- EQUIV ALPHA : thm
```

Pure λ-calculus is formalized by the following codes with the function `define_quotient_types` for quotient types. This function takes the type `term1` and the equivalence relation `ALPHA`, and creates a new type `term` which is isomorphic to the quotient set of the type `term1` by `ALPHA`.

```
val [...] =
    define_quotient_types
    {types = [{name = "term", equiv = ALPHA_EQUIV}], ... };
```

This creates `term` as a new type for Λ. The functions `Con`, `Var`, `App` and `Lam` are the corresponding constructor functions. For the pre-λ-term `t`, the notation $\lfloor t \rfloor$ is represented by the code `(term_ABS t)`. For the pure λ-term `T`, the notation $\lceil T \rceil$ is represented by the code `(term_REP T)`.

The code `(t <[ [(x1,t1); ... ; (xn,tn)])` represents the pure λ-term which is obtained from the pure λ-term `t` by substituting the pure λ-terms `t1`,...,`tn` for the free variables `x1`, ...,`xn` respectively. The substitution is defined as a function which has the following property. To avoid variable capturing, the function changes names of bound variables before substitutions.

```
|- (!a s. Con a <[ s = Con a) /\ (!x s. Var x <[ s = SUB s x) /\
   (!t u s. App t u <[ s = App (t <[ s) (u <[ s)) /\
   !x u s.
     Lam x u <[ s =
     (let x’ = variant x (FV_subst s (FV u DIFF {x})) in
        Lam x’ (u <[ (x,Var x’)::s))
```

The code (`RED1 R t1 t2`) represents that there is a 1-step R-reduction `t1` $\rightarrow_R$ `t2` for the pure λ-terms `t1` and `t2`. This code is defined so that the following holds.

```
|- (!R t1 t2. R t1 t2 ==> RED1 R t1 t2) /\
   (!R t1 u t2. RED1 R t1 t2 ==> RED1 R (App t1 u) (App t2 u)) /\
   (!R t u1 u2. RED1 R u1 u2 ==> RED1 R (App t u1) (App t u2)) /\
   !R x t1 t2. RED1 R t1 t2 ==> RED1 R (Lam x t1) (Lam x t2)
```

The code (`RED R t1 t2`) represents that there is a R-reduction `t1` $\rightarrow_R^*$ `t2` for the pure λ-terms `t1` and `t2` where $\rightarrow_R^*$ is the transitive reflexive closure of the relation $\rightarrow_R$. This code is defined so that the following holds.

```
|- (!R t1 t2. RED1 R t1 t2 ==> RED R t1 t2) /\
   (!R t1. RED R t1 t1) /\
    !R t1 t3. (?t2. RED R t1 t2 /\ RED R t2 t3) ==> RED R t1 t3
```

The code (`REQUAL R t1 t2`) means that the pure λ-terms `t1` and `t2` are R-equivalent. This relation is defined as the smallest equivalence relation which includes $\rightarrow_R^*$. This code is defined so that the following holds.

```
|- (!R t1 t2. RED R t1 t2 ==> REQUAL R t1 t2) /\
   (!R t2 t1. REQUAL R t1 t2 ==> REQUAL R t2 t1) /\
   !R t1 t3. (?t2. REQUAL R t1 t2 /\ REQUAL R t2 t3)
          ==> REQUAL R t1 t3
```

The code (NORMAL_FORM R N) means that the pure λ-term N is an R-normal form, that is, there is no 1-step R-reduction from the term N. This code is defined so that the following holds.

```
|- !R a. NORMAL_FORM R a = !a'. ~RED1 R a a'
```

The code (NORMAL_FORM_OF R N M) means that the pure λ-term N is an R-normal form of the pure λ-term M, that is, M is R-equivalent to N, and N is an R-normal form. This code is defined so that the following holds.

```
|- !R a b. NORMAL_FORM_OF R a b = NORMAL_FORM R a /\ REQUAL R b a
```

The code (BETA_R N M) means that the pure λ-term M is the result of a contraction of the β-redex N that is a pure λ-term. This code is defined so that the following holds.

```
|- !x u t. BETA_R (App (Lam x u) t) (u <[ [(x,t)])
```

By using these predefined functions in the package, we define our formalization of weak normalization. The code (WN N) means that the pure λ-term N is weakly normalizing, that is, it is β-equivalent to some β-normal form. This code is defined so that the following holds.

```
|- !M. WN M = ?N. NORMAL_FORM_OF BETA_R N M
```

4.3 Control Paths

In this subsection we will explain our formalization of control paths and adjacent control paths.

The code (FV1 N) is defined in the package as the set of names of the free variables of the pre-λ-term N. We define the following in Λ_1. The code (BV1 N) means the set of names of the bound variables of N. The codes (mApp1 X Nl) and (mLam1 yl M) mean terms like $X\vec{N}$ and $\lambda\vec{y}.M$ where Nl and yl represent $\vec{N}$ and $\vec{y}$ respectively.

In order to formalize the notion of a subterm, we introduce an indicator, which is a list of natural numbers. For a given pre-λ-term, we use this number to denote one of its maximal proper subterms. If M is $\lambda x.N$, then 0 denotes N. If M is NL, then 0 denotes N and 1 denotes L. The indicator $(n_1,\ldots,n_k)$ in M denotes the subterm obtained from M by taking the subterm denoted by n_1, and then taking its subterm denoted by n_2, ..., and then taking its subterm denoted by n_k. For example, let $M = t(\lambda uz.xz)$. The indicator (1,0,0,1) denotes the subterm z in M. The code (SUBTERM1 M p = N) represents that the indicator p in the pre-λ-term M denotes the subterm N.

The code (FVo1 N) means the set of indicators which denote the free variable occurrences in the pre-λ-term N. The code (CF1 N) means that the pre-λ-term N does not contain any constants. We use it since our λ-calculus given in Ref. [7] does not contain any constants and on the other hand the HOL package defines λ-calculus with constants. The code (BC_OK N) means that the pre-λ-term N satisfies Barendregt convention, that is, all names of bound variables are different from each other and different from those of the free variables.

We add the following for Λ. The codes (mApp X Nl) and (mLam yl M) mean terms like $X\vec{N}$ and $\lambda\vec{y}.M$ where Nl and yl represent $\vec{N}$ and $\vec{y}$ respectively.

In the following definitions, the code M is a pre-λ-term. The relation $x \rightsquigarrow_1 y$ is represented by the code (control1 M x y) as follows.

```
val control1 =
    Define
   `control1 M xo yo =
      (NORMAL_FORM BETA_R (term_ABS M)) /\
      (?M1 Nl L yl x y.
        (  (SUBTERM1 M xo = (Var1 x))
         /\(SUBTERM1 M yo = (Var1 y))
         /\(SUBTERM1 M M1 =
              (App1 (mApp1 (Var1 x) Nl) (mLam1 yl L)))
         /\(xo=(M1++[0]++(MAP (\x.0) Nl)))
         /\(?yo_in_L.
              ((yo=(M1++[1]++(MAP (\x.0) yl)++yo_in_L)) /\
              (yo_in_L IN (FVo1 L)))
           )
         /\(MEM y yl))
      )`;
```

Section 2 defines the relation $x \rightsquigarrow y$ in M to hold if there exists some control path $(x, x_1, \ldots, x_n, y)$ in M. We formalize this notion for the path $(x, x_1, \ldots, x_n, y)$ instead of the variables x, y. The following code (control_path M $(x, x_1, \ldots, x_n, y)$) represents that the sequence $(x, x_1, \ldots, x_n, y)$ is a control path in M.

```
val control_path =
    Define
   `control_path M xol =
              (NORMAL_FORM BETA_R (term_ABS M)) /\
              (LENGTH xol >= 1) /\
              (!i.
                  (0 <= i /\ i < (LENGTH xol - 1))
                   ==> (control1 M (EL i xol) (EL (SUC i) xol))
              ) /\
              ((EL 0 xol) IN (FVo1 M))`;
```

The following code (control_path_from S M $(x_0, \ldots, x_{n-1})$) represents that the sequence $(x_0, \ldots, x_{n-1})$ is a control path from S in M.

```
val control_path_from =
    Define
   `control_path_from s M xol =
        (s SUBSET (FV1 M)) /\
        (control_path M xol) /\
        ?x.((SUBTERM1 M (EL 0 xol) = (Var1 x)) /\ (x IN s))`;
```

The following code (adjacent M x y) represents that the variable occurrences x and y are adjacent in M.

```
val adjacent =
    Define
   `adjacent M xo yo =
      (NORMAL_FORM BETA_R (term_ABS M)) /\
      (?M1 Nl Ll zl x y.
        (  (SUBTERM1 M xo = (Var1 x)) /\
           (SUBTERM1 M yo = (Var1 y)) /\
           (SUBTERM1 M M1 =
               (App1 (mApp1 (Var1 x) Nl)
                   (mLam1 zl (mApp1 (Var1 y) Ll)))
           )  /\
           (xo=(M1++[0]++(MAP (\x.0) Nl))) /\
           (yo=(M1++[1]++(MAP (\x.0) zl)++(MAP (\x.0) Ll)))
        )
      )`;
```

The following code (adjacent_control_paths M $(x_0, \ldots, x_{n-1})$ $(y_0, \ldots, y_{m-1})$) represents that the sequences $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{m-1})$ are adjacent control paths in M.

```
val adjacent_control_paths =
   Define
   ‘adjacent_control_paths M xol yol =
               (control_path M xol) /\
               (control_path M yol) /\
               (adjacent M (LAST xol) (LAST yol))‘;
```

The following code (adjacent_control_paths_from S M $(x_0, \ldots, x_{n-1})$ $(y_0, \ldots, y_{m-1})$) represents that the sequences $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{m-1})$ are adjacent control paths from S in M.

Section 2 does not explicitly define adjacent control paths from S in M. We define the predicate adjacent_control_paths_from by combining control_path_from and adjacent_control_paths as follows: Two control paths $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{m-1})$ from S in M are called adjacent control paths from S in M if the two paths $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{m-1})$ are adjacent control paths in M.

```
val adjacent_control_paths_from =
   Define
   ‘adjacent_control_paths_from s M xol yol =
        (control_path_from s M xol) /\
        (control_path_from s M yol) /\
        (adjacent_control_paths M xol yol)‘;
```

In order to prove the key lemma (Lemma 2.10), we have to separately define its negation as follows. We will explain the reason in Section 6.5.

```
val no_adjacent_control_paths_from =
   Define
   ‘no_adjacent_control_paths_from s M =
        ~(?yol zol.
           ( (adjacent_control_paths M yol zol) /\
             (?y.((SUBTERM1 M (EL 0 yol) = (Var1 y)) /\
                  (y IN s) /\
                  (y IN FV1(M))
                )) /\
             (?z.((SUBTERM1 M (EL 0 zol) = (Var1 z)) /\
                  (z IN s) /\
                  (z IN FV1(M))
                ))
           )
         )‘;
```

5. Formal Proofs

This subsection explains our formal proofs.

Lemma 2.8 is formalized as follows. Note that this is the lemma 2.7 in Ref. [7].

```
val lem_2_7 =
      mk_thm ([],
‘‘!N.
    ( CF1 N ==>
     (NORMAL_FORM BETA_R (term_ABS N)) ==>
     (?yol zol.adjacent_control_paths_from {x} N yol zol) ==>
     ?X.
       ((WN X) /\ (CF X) /\ ~(WN ((term_ABS N) <[ [(x,X)]))))‘‘);
```

Lemma 2.9 is formalized as follows.

```
g ‘!N.
     (CF1 N) ==>
     (BC_OK N) ==>
     (NORMAL_FORM BETA_R (term_ABS N)) ==>
     (!X.
         (
           ((WN X) /\ (CF X)) ==>
           (WN ((term_ABS N) <[ [(x,X); (y,X)]))
         )
     ) ==>
```

```
       ~(?vol wol.adjacent_control_paths_from {x;y} N vol wol)‘;
```

Lemma 2.10 is as follows: If there are no adjacent control paths from $\vec{x}$ in $N \in$ NF, then $N[\vec{x} := \vec{X}] \in$ WN for all $\vec{X} \in$ WN.

To prove this lemma smoothly, we have to change it in two points. First, in order to use Lemma 6.2, we have to assume Barendregt convention as well as the condition that for all X_i in $\vec{X}$, X_i must not have any bound variables of N as its free variables. Secondly, for the reason explained in Section 6.5, we have to use no_adjacent_control_paths_from in its formalization. Then we reach the next lemma and its formalization.

Lemma 5.1 (New Key Lemma) *Assume there are no adjacent control paths in $N \in NF$ from $\vec{x}$, and each x in $\vec{x}$ is not in the set of the bound variables of N. For all $\vec{X}$, if for all X in $\vec{X}$, $X \in WN$, and the set of the bound variables of N and the set of the free variables of X are disjoint, then $N[\vec{x} := \vec{X}] \in WN$.*

This is formalized as follows.

```
g ‘!N xl.
      (BC_OK N) ==>
      (CF1 N) ==>
      (NORMAL_FORM BETA_R (term_ABS N)) ==>
      ((BV1 N) INTER (LIST_TO_SET xl) = EMPTY) ==>
      (no_adjacent_control_paths_from (LIST_TO_SET xl) N) ==>
      !XL.
          ((EVERY WN XL) /\
           (LENGTH xl = LENGTH XL) /\
           (EVERY (\X.((BV1 N) INTER (FV X) = EMPTY)) XL)
           ==> (WN ((term_ABS N) <[ (ZIP (xl,XL))))
          )‘;
```

For the entire verification of Substitution Theorem, we used 326 lemmas and we wrote 10119 lines of HOL code. The whole verification takes 28 minutes by HOL4 Kananaskis 3, and 1 hour 49 minutes by HOL4 Kananaskis 7, on IBM ThinkPad G41 with Mobile Intel Pentium4 Processor 552 (3.46 GHz), 1.0 GB Memory, 80 GB HDD (5,400 rpm), and Microsoft Windows XP Professional SP3.

6. Challenges

In this section, we explain several challenges in our verification.

6.1 Verification of New and Significant Theorem in λ-Calculus

Substitution theorem is a new theorem in untyped λ-calculus. Since untyped λ-calculus has been fully developed and it is difficult to obtain a new theorem, a new theorem in this field is often subtle and difficult.

The statement of Substitution Theorem is simple. However, its proof is complicated and uses subtle notions and difficult techniques, which are hidden in the statement. Substitution Theorem is a part of deep results given in Ref. [7], which solved some open question on the model $\mathcal{HL}_\infty$, and it is sufficiently significant. One of our main challenges in our verification is to formalize and verify a new and significant theorem.

6.2 λ-Terms Without Bound Variable Renaming

The key notion in the proof of Substitution theorem is a control path. This notion is defined by using names of bound variables. Hence this notion is meaningful only for λ-terms without bound variable renaming. In our formalization, we have to handle λ-

terms without bound variable renaming.

Control paths are defined for bound variable occurrences and bound variable names. Even though we formalize bound variable occurrences by natural number sequences by standard technique, we have to use bound variable names, since only bound variable names can describe which λ-abstraction captures a given bound variable occurrence. In our work we do not take an approach by coding of variables such as de Bruijn indexes, since bound variable names are more readable.

Substitution requires bound variable renaming in order to avoid variable capturing. β-reduction is defined by using substitution. Hence substitution and β-reduction are notions only for λ-terms with bound variable renaming.

In our formalization, we use the contextual α-equivalence, and two kinds of λ-terms. One is pre-λ-terms that do not have bound variable renaming, and the other is pure λ-terms that have bound variable renaming. A pure λ-term is an equivalence class of pre-λ-terms with respect to α-equivalence.

We formalize a control path as a notion for pre-λ-terms, and we formalize substitution and β-reduction as a notion for pure λ-terms. Verifying properties across these two kinds of λ-terms is complicated and challenging.

6.3 Parametrized Persistent Weak Normalization

It is difficult to show that $N \in \mathrm{PWN}$ in our subgoals, because of variable capturing. In order to solve this difficulty, we introduce the notion PWN_S for a set S of variable names. Since S gives some conditions for free variables, we can avoid variable capturing. The set PWN_S is a superset of PWN, but this set is sufficient for our verification. We will explain this in details in this subsection.

Persistent weakly normalizing terms PWN is defined in Definition 2.1 as follows: We say a λ-term M is *persistently weakly normalizing* if for all n and for all $X_1, \ldots, X_n \in \mathrm{WN}$ we get $MX_1 \ldots X_n \in \mathrm{WN}$. We denote the set of persistently weakly normalizing λ-terms by PWN.

Lemma 2.3 states that if $M \in \mathrm{WN}$ and $N \in \mathrm{PWN}$ then $MN \in \mathrm{WN}$. When we use this lemma to show $MN \in \mathrm{WN}$, we have to show $N \in \mathrm{PWN}$, that is, we have to show $\forall L_1 \ldots L_n \in \mathrm{WN}(NL_1 \ldots L_n \in \mathrm{WN})$. When we show $\forall L_1 \ldots L_n \in \mathrm{WN}(NL_1 \ldots L_n \in \mathrm{WN})$, we use β-reduction for $NL_1 \ldots L_n$. This β-reduction may cause some substitution with variable capturing, and hence it is difficult to handle this situation.

In order to solve this difficulty, we introduce the notion PWN_S for a set S of variable names. The notion PWN_S is weaker than PWN. When we show $N \in \mathrm{PWN}_S$, it is sufficient to show $NL_1 \ldots L_n \in \mathrm{WN}$ for only terms $L_1 \ldots L_n \in \mathrm{WN}$ whose free variables are not in S. By taking S to be some set of the bound variables in N, we can avoid variable capturing in β-reduction for $NL_1 \ldots L_n$.

Definition 6.1 (PWN_S) Assume S is a set of variable names. We say a λ-term M S-free persistently weakly normalizing λ-term if for all n and $X_0, \ldots, X_{n-1} \in \mathrm{WN}$ such that $\mathrm{FV}(X_0) \cap S = \ldots = \mathrm{FV}(X_{n-1}) \cap S = \emptyset$, we get $MX_0 \ldots X_{n-1} \in \mathrm{WN}$.
We denote the set of S-free persistently weakly normalizing λ-terms by PWN_S.

The set PWN_S is formalized as follows. The formula `(PWN_c S M)` represents $M \in \mathrm{PWN}_S$.

```
val PWN_c = Define
‘PWN_c s M =
  !XL.
  (
    ((EVERY WN XL) /\ (EVERY (\X.(((FV X) INTER s) = EMPTY)) XL))
      ==>
    (WN (mApp M XL))
  )‘;
```

Instead of Lemma 2.3, we use the next lemma.

Lemma 6.2 (New Lemma for PWN_S) *Assume S is a finite set of variable names. If $FV(M) \cap S = FV(N) \cap S = \emptyset$, $M \in WN$, and $N \in PWN_S$, then we have $MN \in WN$.*

This lemma is formalized as follows.

```
g ‘(WN M) /\ (FINITE s) /\ (PWN_c s N) /\
 (((FV M) INTER s) = EMPTY) /\ (((FV N) INTER s) = EMPTY)
 ==> (WN (App M N)) /\ (((FV (App M N)) INTER s) = EMPTY)‘;
```

This improvement enables us to prove Lemma 5.1 in a feasible way. We explain an example of some subgoal in our formal proof of the lemma. The next is one of our subgoals, where ... shows some abbreviation for assumptions.

```
  PWN_c (BV1 (mApp1 (Var1 z) Ll))
    (term_ABS (mLam1 yl’ (mApp1 (Var1 v) Ll’)) <[ ZIP (xl,XL’))
  -----------------------------------
          ...
     : proof
```

By some tactics including expansion of the definition of `PWN_c`, this subgoal is transformed into the next subgoal. We obtain the assumption 33 from the new condition added to `PWN_c`.

```
 REQUAL BETA_R
  (mApp
   (mLam yl’ (mApp (Var v) (MAP term_ABS Ll’) <[ ZIP (xl,XL’)))
    XL1
  )
  (mApp (Var v) (MAP term_ABS Ll’) <[ ZIP (xl ++ yl’,XL’ ++ XL1))
 -----------------------------------
              ...
  33.  EVERY (\X. FV X INTER BV1 (mApp1 (Var1 z) Ll) = {}) XL’’
              ...
    : proof
```

The term `(mApp (mLam yl’ (mApp (Var v) (MAP term_ABS Ll’) <[ ZIP (xl,XL’))) XL1)` represents $(\lambda y_1 \ldots y_n.u)t_1 \ldots t_n$ where `yl’` represents $y_1, \ldots, y_n$, the code `XL1` represents $t_1, \ldots, t_n$, and the code `(mApp (Var v) (MAP term_ABS Ll’) <[ ZIP (xl,XL’))` represents u. The assumption 33 guarantees that there is no variable capturing by $y_1, \ldots, y_n$ when we β-reduce the redex $(\lambda y_1 \ldots y_n.u)t_1 \ldots t_n$. This improvement enables us to prove the subgoal.

6.4 Preservation of Control Paths by Substitution

In our verification, it is difficult to prove the subgoal that states that a control path is preserved by substitution. This property seems mathematically trivial and Section 2 uses it without proofs. However, its formal proof is difficult because a control path is a notion defined for pre-λ-terms, that is, λ-terms without bound variable renaming, and on the other hand substitution is a notion defined for pure λ-terms, that is, λ-terms with bound variable renaming. Hence the formal statement of this property uses both styles of λ-terms in a mixed way, and it is difficult to find an appropriate induction. A naive formalization of this property is given as follows.

```
    |- !N x y vo wo.
         BC_OK N ==>
         ~(x IN BV1 N) ==>
         ~(y IN BV1 N) ==>
         control1 N vo wo ==>
         control1 (term_REP (term_ABS N <[ [(y,Var x)])) vo wo
 : thm
```

This formalization means the following. Assume the Barendregt convention. If $v \rightsquigarrow_1 w$ in N, then $v \rightsquigarrow_1 w$ in $\lceil \lfloor N \rfloor [y := x] \rceil$, where N is a pre-λ-term, $\lfloor N \rfloor$ is a pure λ-term that is the α-equivalence class of N, and $\lceil \lfloor N \rfloor [y := x] \rceil$ is a pre-λ-term that is a representative of the equivalence class $\lfloor N \rfloor [y := x]$.

The direct way of proving it is by induction on N. However, it is difficult to use induction hypothesis, since N is inside the nest of $\lceil . \rceil$ and $\lfloor . \rfloor$. In order to solve this difficulty, we produce the following lemma.

```
    |- !M1 N x y vo wo Nl L yl x' y' yo_in_L.
         BC_OK N ==>
         ~(x IN BV1 N) ==>
         ~(y IN BV1 N) ==>
         NORMAL_FORM BETA_R (term_ABS N) /\
         (SUBTERM1 N vo = Var1 x') /\
         (SUBTERM1 N wo = Var1 y') /\
         (SUBTERM1 N M1 =
           App1 (mApp1 (Var1 x') Nl) (mLam1 yl L)) /\
         (vo = M1 ++ [0] ++ MAP (\x. 0) Nl) /\
         (wo = M1 ++ [1] ++ MAP (\x. 0) yl ++ yo_in_L) /\
         yo_in_L IN FVo1 L /\ MEM y' yl ==>
         ?Nl L yl x' y'.
         (SUBTERM1 (term_REP (term_ABS N <[ [(y,Var x)])) vo =
           Var1 x') /\
         (SUBTERM1 (term_REP (term_ABS N <[ [(y,Var x)])) wo =
           Var1 y') /\
         (SUBTERM1 (term_REP (term_ABS N <[ [(y,Var x)])) M1 =
           App1 (mApp1 (Var1 x') Nl) (mLam1 yl L)) /\
         (vo = M1 ++ [0] ++ MAP (\x. 0) Nl) /\
         (wo = M1 ++ [1] ++ MAP (\x. 0) yl ++ yo_in_L) /\
         yo_in_L IN FVo1 L /\ MEM y' yl : thm
```

It is equivalent to the previous lemma, and we can prove it by induction on the indicator `M1`. This technique solves our difficulty.

6.5 Negation of Existence of Adjacent Control Paths

A mathematician defines some notion first, and then he defines another notion by using the first notion. Some conditions for free variables are often subtle and implicit. It may happen that the first notion assumes some free variable condition and the second notion implicitly assumes another free variable condition. This difficulty happens for the definition of `adjacent_control_paths_from`. This definition works when we use it positively in assumptions, but it does not works when we use its negation in assumptions, because of implicit free variable condition. In order to solve this difficulty, we introduce the predicate `no_adjacent_control_paths_from` to express its negation. We will explain it in details in this subsection.

Our mathematical proof in Section 2 defines "a control path from S in M," but it does not define "adjacent control paths from S in M." Following a usual way of mathematical description, Section 2 assumes that "adjacent control paths from S in M" is implicitly defined by combining the two notions of "a control path from S in M" and adjacent control paths. In Definition 2.4 of "a control path from S in M," it assumes that S is a set of free variables.

Our formalization faithfully follows Section 2. First we define `control_path_from` as follows. We assume `(s SUBSET (FV1 M))` because Definition 2.4 also includes this condition.

```
val control_path_from =
    Define
   `control_path_from s M xol =
        (s SUBSET (FV1 M)) /\ (control_path M xol) /\
         ?x.((SUBTERM1 M (EL 0 xol) = (Var1 x)) /\ (x IN s))`;
```

The predicate `adjacent_control_paths_from` is defined by combining `control_path_from` and `adjacent_control_paths` as follows.

```
val adjacent_control_paths_from =
   Define
   `adjacent_control_paths_from s M xol yol =
        (control_path_from s M xol) /\
        (control_path_from s M yol) /\
        (adjacent_control_paths M xol yol)`;
```

However, the predicate `adjacent_control_paths_from` does not work when we use its negation, because `control_path_from` assumes the free variable condition and this is not the free variable condition implicitly assumed for "adjacent control paths from S in M" in Section 2. For example, when we prove some statement that contains the negation of "adjacent control paths from S in M" in its assumption, we have to show some extra free variable condition, and our formal proof does not go further.

In order to solve this difficulty, we introduce the predicate `no_adjacent_control_paths_from` to separately define the negation of `adjacent_control_paths_from`. Its code is as follows:

```
val no_adjacent_control_paths_from =
   Define
   `no_adjacent_control_paths_from s M =
        ~(?yol zol.
           ( (adjacent_control_paths M yol zol) /\
             (?y.((SUBTERM1 M (EL 0 yol) = (Var1 y)) /\
                  (y IN s) /\
                  (y IN FV1(M))
                )) /\
             (?z.((SUBTERM1 M (EL 0 zol) = (Var1 z)) /\
                  (z IN s) /\
                  (z IN FV1(M))
                ))
           )
         )`;
```

Lemma 2.10, which is the key lemma in Section 2, is an example. Its statement contains the negation of "adjacent control paths from S in M" in its assumption. This improvement enables us to complete our formal proof of this lemma.

6.6 Direct Definition of Adjacent Control Paths

In the original paper [7], adjacent control paths are defined by using control paths and adjacent variable occurrences. Both the definition of control paths and the definition of adjacent variable occurrences require bound variable names. However we find that we can directly define adjacent control paths without using them. By this technique, we formalize the notion of adjacent control paths for ordinary λ-terms with α-conversion. This definition works for the proof of Lemma 2.8 and simplifies our formalization.

For a normal pure λ-term M, a set S of variables, and a number n, we define $\mathrm{AC}(M, S, n)$ by induction on n as follows:

$$
\begin{aligned}
&(1)\ M = H[\lambda\vec{g}.x\vec{N}(\lambda\vec{u}.y\vec{L})\vec{G}] \wedge \mathrm{HP}(H, s) \wedge S = \{x, y\} \wedge \\
&\qquad x, y \notin s \cup \vec{g} \wedge y \notin \vec{u} \wedge n = 0, \\
&\text{or } (2)\ M = H[\lambda\vec{g}.x\vec{N}(\lambda\vec{u}^{\,j}.M^*)\vec{G}] \wedge \mathrm{HP}(H, s) \wedge S = \{x, y\} \wedge \\
&\qquad x, y \notin s \cup \vec{g} \wedge y \notin \vec{u}^{\,j} \wedge \\
&\qquad (x \neq y \wedge \mathrm{AC}(M^*, \{y, u_j\}, m) \wedge n = m + 1 \vee \\
&\qquad x = y \wedge M^* = \lambda\vec{v}.u_j\vec{L} \wedge u_j \notin \vec{v} \wedge n = 1), \\
&\text{or } (3)\ M = H[\lambda\vec{g}.x\vec{N}(\lambda\vec{u}^{\,k}.M^*)\vec{G}] \wedge \mathrm{HP}(H, s) \wedge S = \{x\} \wedge \\
&\qquad x \notin s \cup \vec{g} \wedge j \leq k \wedge u_j \notin \vec{u}^{\,k}_{j+1} \wedge \\
&\qquad \mathrm{AC}(M^*, \{u_j, u_k\}, m) \wedge n = m + 2,
\end{aligned}
$$

where in each clause we suppose the free variables except M, S, n are existentially quantified, and $\mathrm{HP}(H, s)$ means that H is a hereditary head normal context and s is its bound variables for the hole.

$\mathrm{AC}(M, S, n)$ means that there are adjacent control paths $x \rightsquigarrow z$ and $y \rightsquigarrow q$ such that $x, y \in S$ and the sum of the lengths of these paths is n.

The proof of Lemma 2.8 was not given in Ref. [7] and it was given in Ref. [2]. The proof in Ref. [2] is complicated and confusing. We can simplify their proof by using $\mathrm{AC}(M, S, n)$.

The formula $\mathrm{AC}(M, S, n)$ works for ordinary λ-term with α-conversion. Since Lemma 2.8 does not require intensive analysis of control paths and adjacent variable occurrences, we can finish its proof within pure λ-terms without using the notion of control paths and adjacent variable occurrences.

7. Conclusion

We have formalized and verified Substitution Theorem for weak normalization in Ref. [7] by using HOL. This verification is challenging, since Substitution Theorem is a new and significant theorem in λ-calculus. For our verification, we have handled λ-terms without bound variable renaming as well as λ-terms with bound variable renaming in a mixed way. We have solved several difficulties such as parametrized persistent weak normalization, preservation of control paths by substitution, and negation of the existence of adjacent control paths.

We have investigated techniques for handling bound variables and free variables in formal proofs. Our techniques may help to verify other theorems such that bound variables and free variables are crucial in them. In particular, our techniques can apply to the verification of Substitution Theorem for strong normalization in Ref. [7] and Lemma A13 in Ref. [2].

Both of them are new and deep theorems in λ-calculus, and use the notion of control paths, which is the key notion in our formalization. Their verification would be our future work.

Reference

[1] Barendregt, H.P.: *The Lambda Calculus*: *its Syntax and Semantics*, North-Holland (1984).

[2] Dezani-Ciancaglini, M., Honsell, F. and Motohama, Y.: Compositional Characterization of λ-terms using Intersection Types, *Theor. Comput. Sci.*, Vol.340, No.3, pp.459–495 (2005).

[3] Gonthier, G.: Formal Proof—The Four-Color Theorem, *Notices of the American Mathematical Society*, Vol.55, No.11, pp.1382–1393 (2008).

[4] Homeier, P.V.: A proof of the church-rosser theorem for the lambda calculus in higher order logic, *Supplemental Proc. 14th International Conference on Theorem Proving in Higher Order Logics* (*TPHOLs 2001*), pp.207–222 (2001).

[5] McKinna, J. and Pollack, R.: Pure Type Systems Formalized, *Lecture Notes in Computer Science*, Vol.664, pp.289–305 (1993).

[6] Shankar, N.: A mechanical proof of the Church-Rosser theorem, *JACM*, Vol.35, No.3, pp.475–522 (1988).

[7] Tatsuta, M., and Dezani-Ciancaglini, M.: Normalisation is Insensible to lambda-term Identity or Difference, *Proc. 21st Annual IEEE Symposium on Logic in Computer Science*, pp.327–336 (2006).

[8] Tatsuta, M.: Formalization of Lemma for Adjacent Replacement Paths, NII Technical Report, NII-2012-002E (2012).

[9] Urban, C.: Nominal Techniques in Isabelle/HOL, *Journal of Automated Reasoning*, Vol.40, No.4, pp.327–356 (2008).

Takayuki Koai was born in 1975. He received his M.S. from Kyoto University in 2002, and his withdrawal from the doctoral program with the completion of course Requirements from the Graduate University for Advanced Studies in 2010. He was engaged in research on automated theorem proving and type theory with National Institute of Informatics from 2007. He became a project researcher at National Institute of Informatics in 2010. His current research interests are automated theorem proving and type theory.

Makoto Tatsuta was born in 1960. He received his M.S. and Ph.D. from the University of Tokyo in 1987 and 1993 respectively. At Tohoku University he became a research associate and an associate professor in 1989 and 1994 respectively. He became an associate professor at Kyoto University in 1996, and a professor at National Institute of Informatics in 2001. His current research interests are theoretical computer science and mathematical logic, in particular, type theory and constructive logic.