Type Checking and Inference Are Equivalent in Lambda Calculi with Existential Types

Yuki Kato* and Koji Nakazawa**

Graduate School of Informatics, Kyoto University, Kyoto 606-8501, Japan

Abstract. This paper shows that type-checking and type-inference problems are equivalent in domain-free lambda calculi with existential types, that is, type-checking problem is Turing reducible to type-inference problem and vice versa. In this paper, the equivalence is proved for two variants of domain-free lambda calculi with existential types: one is an implication and existence fragment, and the other is a negation, conjunction and existence fragment. This result gives another proof of undecidability of type inference in the domain-free calculi with existence.
Keywords. undecidability, existential type, type checking, type inference, domain-free type system.

1 Introduction

Existential types correspond to second-order existence in logic by the Curry-Howard isomorphism, and so they are a natural notion from the point of view of logic. They have been also studied actively from the point of view of computer science since Mitchell and Plotkin [11] showed that abstract data types are existential types. Furthermore, calculi with existential types work as suitable target calculi of continuation-passing-style (CPS) translations. Some studies on CPS translations for polymorphic calculi have shown that the negation ($\neg$, which corresponds to continuation types), conjunction ($\wedge$, which corresponds to product types), and existence ($\exists$) fragment of lambda calculus is an essence of a target calculus of CPS translations for various systems, such as the polymorphic lambda calculus [5], the lambda-mu calculus [3, 8], and delimited continuations. Hasegawa [9] showed that a $\neg \wedge \exists$-fragment is even more suitable as a target calculus of a CPS translation for delimited continuations such as `shift` and `reset` [2]. These can be seen as an extension of the study of Thielecke [18], in which he showed that the negation and conjunction fragment of a lambda calculus suffices as a target of CPS translations of various first-order calculi.

Domain-free type systems [1], which are in an intermediate style between Church and Curry style, are useful to study some extensions of polymorphic typed calculi and for theoretical studies on CPS translations. In domain-free style lambda calculi, types of parameters of functions are not explicitly annotated in lambda abstraction terms $\lambda x.M$ as in the Curry style, while as in the Church

* yuki@kuis.kyoto-u.ac.jp
** knak@kuis.kyoto-u.ac.jp

style, terms contain type information for second-order quantifiers, such as a type abstraction $\lambda X.M$ for $\forall$-introduction rule, and a term $\langle A, M\rangle$ with a witness A for $\exists$-introduction rule. In [7], it is shown that an extension of the Damas-Milner polymorphic type assignment system, which can be seen as a Curry-style formulation, with a control operator destroys the type soundness. Similarly, Fujita [3] showed that the Curry-style lambda-mu calculus, which is an extension of the polymorphic lambda calculus and introduced by Parigot [14], does not enjoy the subject reduction property. Fujita introduced a domain-free lambda-mu calculus to have the subject reduction. In addition, the $\neg \wedge \exists$-fragment of the domain-free typed lambda calculus works as a target calculus of a CPS translation for the domain-free lambda-mu calculus.

Some decision problems on typability of terms in typed calculi have been widely studied. One is *type-checking problem* (TC), which is a problem deciding whether $\Gamma \vdash M : A$ is derivable for given Γ, M, and A. *Type-inference problem* (TI) is another problem deciding whether there exist Γ and A such that $\Gamma \vdash M : A$ is derivable for given M. In the usual notation, TC asks $\Gamma \vdash M : A$? for given Γ, M, and A, and TI asks $? \vdash M :?$ for given M. In this paper, TC_0 and TI_0 denote type checking and inference for closed terms, respectively. These questions are fundamentally important in typed lambda calculi.

For polymorphic types, we have already had some results on these problems. Wells [19] showed that TC and TI in the Curry-style polymorphic lambda calculus are equivalent and these problems are undecidable. Two problems are said to be equivalent if one is Turing reducible to the other and vice versa, where a decision problem P is said to be Turing reducible to another problem Q when there exists computable function F such that for each instance p of P, $F(p)$ is an instance of Q which holds if and only if p holds. Nakazawa and Tatsuta [13] showed that TC and TI in the domain-free polymorphic lambda calculus are equivalent, and these are undecidable. On the other hand, despite of their computational importance, properties of existential types have not been studied enough yet. It is only recent that inhabitation problem, which corresponds to provability of formulas, in the $\neg \wedge \exists$-fragment was proved to be decidable in [17]. TC and TI in domain-free lambda calculi with existential types were proved to be undecidable in [12, 13]. However any direct relation between TC and TI for existential types has not been known yet.

This paper proves that TC and TI are equivalent in two variants of domain-free lambda calculi with existential types: implication and existence fragment $\mathsf{DF}\text{-}\lambda^{\to\exists}$, and negation, conjunction, and existence fragment $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. Moreover, this result gives another proof of undecidability of TI in $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

First, we prove that TC and TI are equivalent in $\mathsf{DF}\text{-}\lambda^{\to\exists}$. In $\mathsf{DF}\text{-}\lambda^{\to\exists}$, it is easy to prove that TI is Turing reducible to TC. The reduction from TC to TI is proved by adapting the idea of [13]. The key of the proof is the fact that, for given a closed term M and a type A, we can construct another closed term $J_{M,A}$ which is typable if and only if $\vdash M : A$ holds.

Secondly, we prove that TC and TI are equivalent in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. Similarly to $\mathsf{DF}\text{-}\lambda^{\to\exists}$, the proof of the reduction from TC to TI consists of two parts: the

reduction from TC to TC_0 and that from TC_0 to TI. However, since $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ does not have implication, we need a non-trivial idea to prove the reduction from TC to TC_0. In this paper, using the well-known fact that the implication $A \to B$ is (classically) equivalent to $\neg(A \wedge \neg B)$, we show that TC can be reduced to TC_0 in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. The proof of the other direction from TI to TC also consists of two parts: TI can be reduced to TI_0 , and TI_0 can be reduced to TC. In order to prove the former part, the above idea can be used.

Figure 1 summarizes the related results including ours. In the diagram, $P \leq Q$ means that the problem P is Turing reducible to Q, and $P \simeq Q$ means that P and Q are equivalent, that is, both $P \leq Q$ and $Q \leq P$ hold. F denotes the polymorphic lambda calculus. SUP means the semi-unification problem and 2UP means the second-order-unification problem. Since undecidability of SUP and 2UP has been already proved by Kfoury et al. [10] and Schubert [15], respectively, all of the problems in the diagram are undecidable. $\simeq^*$ is the main result of this paper, and it gives a new proof of undecidability of TI in $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

Curry style:	SUP	$\overset{[19]}{\leq}$	TC in F	$\overset{[19]}{\simeq}$	TI in F
domain-free style:	2UP	$\overset{[4]}{\leq}$	TC in $\mathsf{DF}\text{-}F$	$\overset{[13]}{\simeq}$	TI in $\mathsf{DF}\text{-}F$
			$\mid\!\wedge$ [12]		$\mid\!\wedge$ [12]
domain-free style:			TC in $\mathsf{DF}\text{-}\lambda^{\to\exists}/\lambda^{\neg\wedge\exists}$	$\simeq^*$	TI in $\mathsf{DF}\text{-}\lambda^{\to\exists}/\lambda^{\neg\wedge\exists}$

Fig. 1. TC and TI for polymorphic and existential types

The section 2 introduces the domain-free lambda calculi with existence: $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. We state our main theorems in the section 3, and we prove them in the sections 4 and 5.

2 Domain-Free Lambda Calculi with Existence

In this section, we define the domain-free lambda calculi with Existential types: $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

These calculi are expressive enough to represent every function which is representable in System F, because we can interpret every term of System F in each of $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ by CPS translations [5, 8]. Furthermore, as pointed out in [11], the existential types can be seen as the abstract data types in the following sense. If a term M has a type $B[X := A]$, we can hide the information of the

type A by constructing the term $\langle A, M\rangle$, which has the existential type $\exists X.B$. A term N of the existential type $\exists X.B$ can be used with $N[Xx.P]$, which intuitively means $\texttt{let}\ \langle X, x\rangle = N\ \texttt{in}\ P$. In this paper, we use the notation $M[Xx.N]$ following [12, 13]. We can write the term $\langle A, M\rangle$ in the style of modules of Standard ML as $\texttt{struct type}\ X = A\ \texttt{val}\ x = M\ \texttt{end}$.

Here are examples of terms with the existential type. First, we define a term of the type $\exists X.\neg X \wedge X$.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash F : \neg\texttt{int}\end{array}\quad \dfrac{}{\vdash 1 : \texttt{int}}\,(Ax)}{\vdash \langle F, 1\rangle : \neg\texttt{int} \wedge \texttt{int}}\,(\wedge I)}{\vdash \langle \texttt{int}, \langle F, 1\rangle\rangle : \exists X.\neg X \wedge X}\,(\exists I)$$

We can consider this term $\langle \texttt{int}, \langle F, 1\rangle\rangle$ as a module, which is implemented as $\langle F, 1\rangle$ by the type $\texttt{int}$, but the information of $\texttt{int}$ is hidden in the type $\exists X.\neg X \wedge X$. We can give a name to this module by the binding mechanism of λ-calculus such as $(\lambda m.m[Xp.(p\pi_1)(p\pi_2)])\langle \texttt{int}, \langle F, 1\rangle\rangle$. The term $\lambda m.m[Xp.(p\pi_1)(p\pi_2)]$ is typed as follows, where Γ denotes the type assignment $p : \neg X \wedge X$.

$$\dfrac{\dfrac{\dfrac{}{m : \exists X.\neg X \wedge X \vdash m : \exists X.\neg X \wedge X}\,(Ax) \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash p : \neg X \wedge X}\,(Ax)}{\Gamma \vdash p\pi_1 : \neg X}\,(\wedge E_1) \quad \dfrac{\dfrac{}{\Gamma \vdash p : \neg X \wedge X}\,(Ax)}{\Gamma \vdash p\pi_2 : X}\,(\wedge E_2)}{\Gamma \vdash (p\pi_1)(p\pi_2) : \bot}\,(\neg E)}{m : \exists X.\neg X \wedge X \vdash m[Xp.(p\pi_1)(p\pi_2)] : \bot}\,(\exists E)}{\vdash \lambda m.m[Xp.(p\pi_1)(p\pi_2)] : \neg(\exists X.\neg X \wedge X)}\,(\neg I)$$

The user of the module does not need to know which type is used in the implementation in the module. We can consider that the term $(\lambda m.m[Xp.(p\pi_1)(p\pi_2)])\langle \texttt{int}, \langle F, 1\rangle\rangle$ corresponds to the following program of Standard ML.

```
structure m = struct
  type X = int
  val p = (F,1)
end
let val (f,a) = m.p in
  f a
end
```

The signatures of Standard ML corresponds to type annotations to terms of the existential type such as $\langle A, M\rangle^{\exists X.B}$. We have no such annotations in the domain-free style.

2.1 Lambda Calculus with Implication and Existence

First, we define the domain-free lambda calculus $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$ with implication ($\rightarrow$) and existence ($\exists$).

Definition 1. *The types (denoted by A, B, ... , and called $\to\exists$-types) and the terms (denoted by M, N, ...) of* DF-$\lambda^{\to\exists}$ *are defined by*

$$A ::= X \mid A{\to}A \mid \exists X.A,$$
$$M ::= x \mid \lambda x.M \mid \langle A, M\rangle \mid MM \mid M[Xx.M].$$

X and x denote a type variable and a term variable, respectively. In the type $\exists X.A$, the variable X in A is bound. In the term $\lambda x.M$, the variable x in M is bound. In the term $N[Xx.M]$, the variables X and x in M are bound. A variable is free if it is not bound. A term is closed if it contains no free term variable. We use $\equiv$ to denote syntactic identity modulo renaming of bound variables.

For $n \geq 3$, $A_1{\to}\cdots{\to}A_{n-1}{\to}A_n$ denotes $A_1{\to}(\cdots{\to}(A_{n-1}{\to}A_n))$, and $M_1M_2\cdots M_n$ denotes $((M_1M_2)\cdots)M_n$. Γ denotes a finite set of type assignments in the form of $x : A$.

Typing rules of DF-$\lambda^{\to\exists}$ *are the following.*

$$\frac{}{\Gamma, x : A \vdash x : A}\ (Ax)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A{\to}B}\ (\to I) \qquad \frac{\Gamma_1 \vdash M : B{\to}A \quad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash MN : A}\ (\to E)$$

$$\frac{\Gamma \vdash N : A[X := B]}{\Gamma \vdash \langle B, N\rangle : \exists X.A}\ (\exists I) \qquad \frac{\Gamma_1 \vdash M : \exists X.A \quad \Gamma_2, x : A \vdash N : C}{\Gamma_1, \Gamma_2 \vdash M[Xx.N] : C}\ (\exists E)$$

In the rule ($\exists E$), Γ_2 and C must not contain X as a free variable.

2.2 Lambda Calculus with Negation, Conjunction, and Existence

Then we define the domain-free lambda calculus DF-$\lambda^{\neg\wedge\exists}$ with negation ($\neg$), conjunction ($\wedge$), and existence ($\exists$). From the point of view of computation, the negation corresponds to the type of continuations, and the conjunction to the product type.

Definition 2. *The types (denoted by A, B, ... , and called $\neg\wedge\exists$-types) and the terms (denoted by M, N, ...) of* DF-$\lambda^{\neg\wedge\exists}$ *are defined by*

$$A ::= X \mid \bot \mid \neg A \mid A \wedge A \mid \exists X.A,$$
$$M ::= x \mid \lambda x.M \mid \langle M, M\rangle \mid \langle A, M\rangle \mid MM \mid M\pi_1 \mid M\pi_2 \mid M[Xx.M].$$

Bound and free variables, and closed terms are defined similarly to DF-$\lambda^{\to\exists}$. *For $n \geq 3$, $A_1 \wedge \cdots \wedge A_{n-1} \wedge A_n$ denotes $A_1 \wedge (\cdots \wedge (A_{n-1} \wedge A_n))$.*

Typing rules of DF-$\lambda^{\neg\wedge\exists}$ *are the following.*

$$\frac{}{\Gamma, x : A \vdash x : A}\ (Ax)$$

$$\frac{\Gamma, x : A \vdash M : \bot}{\Gamma \vdash \lambda x.M : \neg A}\ (\neg I) \qquad \frac{\Gamma_1 \vdash M : \neg A \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash MN : \bot}\ (\neg E)$$

$$\frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash \langle M, N \rangle : A \wedge B}\ (\wedge I)$$

$$\frac{\Gamma \vdash M : A_1 \wedge A_2}{\Gamma \vdash M\pi_1 : A_1}\ (\wedge E_1) \qquad \frac{\Gamma \vdash M : A_1 \wedge A_2}{\Gamma \vdash M\pi_2 : A_2}\ (\wedge E_2)$$

$$\frac{\Gamma \vdash N : A[X := B]}{\Gamma \vdash \langle B, N \rangle : \exists X.A}\ (\exists I) \qquad \frac{\Gamma_1 \vdash M : \exists X.A \quad \Gamma_2, x : A \vdash N : C}{\Gamma_1, \Gamma_2 \vdash M[Xx.N] : C}\ (\exists E)$$

In the rule ($\exists E$), Γ_2 and C must not contain X as a free variable. Note that the typing rules of DF-$\lambda^{\neg\wedge\exists}$ *for the terms $\lambda x.M$ and MN differ from those of* DF-$\lambda^{\to\exists}$.

3 Type Checking and Type Inference

In this section, we introduce two decision problems on typability of terms, and state our main theorem.

Type checking (TC) is a problem deciding whether $\Gamma \vdash M : A$ is derivable for given Γ, M, and A. *Type inference* (TI) is a problem deciding whether there exist Γ and A such that $\Gamma \vdash M : A$ is derivable for given M. In the usual notation, TC asks $\Gamma \vdash M : A$? for given Γ, M, and A, and TI asks $? \vdash M :?$ for given M.

These two problems are equivalent in the Curry-style polymorphic lambda calculus [19], and in the domain-free polymorphic lambda calculus [13]. Two problems are said to be equivalent if one is Turing reducible to the other and vice versa. Hence the equivalence of TC and TI means that (i) for any instance $\Gamma \vdash M : A$? of TC, we can effectively construct a term N such that the answer of the given instance of TC is the same as that of the instance $? \vdash N :?$ of TI, and (ii) for any instance $? \vdash M :?$ of TI, we can effectively construct an instance $\Gamma \vdash N : A$? of TC whose answer is the same as the given instance of TI. TC_0 and TI_0 denote type checking and type inference for closed terms, respectively.

In general, if a decision problem P_1 is Turing reducible to another decision problem P_2, then decidability of P_2 implies decidability of P_1, and equivalently undecidability of P_1 implies undecidability of P_2. In [19, 13], they showed undecidability of TI in the polymorphic lambda calculi by the Turing reducibility of TC to TI. On the other hand, undecidability of TC and TI in the domain-free lambda calculi with existential types has been proved in [12, 13] by the reducibility of each problems for polymorphic types to those for existential types. However, direct relationship between TC and TI for existential types has not been known yet. In this paper, we will prove that TC and TI are equivalent in DF-$\lambda^{\to\exists}$ and DF-$\lambda^{\neg\wedge\exists}$.

Theorem 1. *1. Type checking and type inference are equivalent in* DF-$\lambda^{\to\exists}$*, that is, type checking in* DF-$\lambda^{\to\exists}$ *is Turing reducible to type inference in* DF-$\lambda^{\to\exists}$ *and vice versa.*

2. Type checking and type inference are equivalent in DF-$\lambda^{\neg\wedge\exists}$.

Since TC is undecidable in these calculi, this result gives another proof of undecidability of TI in them.

For each system, the proof of the reduction from TC to TI consists of two parts. First, we show that TC can be reduced to TC_0. Secondly, we show that TC_0 can be reduced to TI.

The key of the proof of the reduction from TC_0 to TI is the fact that we can effectively construct a term $J_{M,A}$ from a given pair of a closed term M and a type A such that the instance $\vdash M : A$ of TC_0 is equivalent to the instance $\vdash J_{M,A}$:? of TI. By this fact, we can conclude that TC_0 can be reduced to TI. In order to show that, we borrow the idea of [13] for polymorphic types.

In $\mathsf{DF}\text{-}\lambda^{\to\exists}$, the reduction from TC to TC_0 is easy, whereas the reduction is not easy to prove for $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ due to absence of implication. In our proof, we show that we can construct a $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$-term $\underline{\lambda}x.M$ for a $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$-term M and a variable x such that $\Gamma, x : A \vdash M : B$ holds if and only if $\Gamma \vdash \underline{\lambda}x.M : \neg(A \wedge \neg B)$ holds. By this construction, we can prove that TC can be reduced to TC_0 in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

Similarly, the proof of the reduction from TI to TC consists of two parts: the reduction from TI to TI_0, and the reduction from TI_0 to TC. For $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$, the proof of the reduction from TI to TI_0 has the similar difficulties to the case of the reduction from TC to TC_0. We can also use the same technique by $\underline{\lambda}x.M$ to prove it.

We prove the equivalence in $\mathsf{DF}\text{-}\lambda^{\to\exists}$ in the section 4, where we show how to construct $J_{M,A}$ from M and A. In the section 5, we show the reduction from problems for open terms to those for closed terms, and prove the equivalence in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

4 TC and TI are Equivalent in $\mathsf{DF}\text{-}\lambda^{\to\exists}$

In this section, we prove that TC and TI are equivalent in $\mathsf{DF}\text{-}\lambda^{\to\exists}$.

At first, we show that TI is Turing reducible to TC (that is denoted by TI $\leq$ TC).

Proposition 1. *TI in $\mathsf{DF}\text{-}\lambda^{\to\exists}$ is Turing reducible to TC in $\mathsf{DF}\text{-}\lambda^{\to\exists}$.*

Proof. For a given instance $? \vdash M :?$ of TI in $\mathsf{DF}\text{-}\lambda^{\to\exists}$, we can effectively construct the list $(x_1, \cdots, x_n)$ of all of the free variables in M. Then the TI problem $? \vdash M :?$ is equivalent to a TC problem $\vdash \lambda y.(\lambda x.y)(\lambda x_1.\cdots\lambda x_n.M) : X{\to}X?$, where x and y are fresh variables. In fact, if the term $\lambda y.(\lambda x.y)(\lambda x_1.\cdots\lambda x_n.M)$ has the type $X \to X$, then M has some type. Conversely, if $\Gamma \vdash M : A$ holds for some Γ and A, we have the following for some B,

$$\dfrac{\dfrac{\dfrac{}{y : X, x : B \vdash y : X}\,(Ax)}{y : X \vdash \lambda x.y : B{\to}X}\,(\to I) \qquad \begin{array}{c}\Gamma \vdash M : A\\ \vdots\\ \vdash \lambda x_1 \cdots \lambda x_n.M : B\end{array}}{\dfrac{y : X \vdash (\lambda x.y)(\lambda x_1 \cdots \lambda x_n.M) : X}{\vdash \lambda y.(\lambda x.y)(\lambda x_1 \cdots \lambda x_n.M) : X{\to}X}\,(\to I),}\,(\to E)$$

where we can suppose that $\{x_1, \cdots, x_n\} = \{z \mid (z : C) \in \Gamma\}$ without loss of generality because the left-hand side is the set of all of the free variables of M. $\square$

As we have stated in the previous section, the proof of TC $\leq$ TI consists of two steps: TC $\leq$ TC_0 and $\mathrm{TC}_0 \leq$ TI. It is easy to prove TC $\leq$ TC_0 for DF-$\lambda^{\rightarrow\exists}$. In the following, we show $\mathrm{TC}_0 \leq$ TI, that is, for a given instance $\vdash M : A?$ of TC_0, we effectively construct a DF-$\lambda^{\rightarrow\exists}$-term $J_{M,A}$ such that the instance $? \vdash J_{M,A} :?$ of TI is equivalent to the given instance of TC_0.

In the rest of this section, O is supposed to be a fixed type variable, and not to be bound by any existential quantifier. $\neg_O A$ denotes $A \rightarrow O$.

First, we define some auxiliary functions on types to prove the key lemma.

Definition 3. *1.* $\mathsf{lvar}(A)$ *is the leftmost variable of* A *when it is free in* A*, and otherwise* $\mathsf{lvar}(A)$ *is undefined.* $\mathsf{lvar}(A)$ *is defined by*

$$\begin{aligned} \mathsf{lvar}(X) &\equiv X, \\ \mathsf{lvar}(A \rightarrow B) &\equiv \mathsf{lvar}(A), \\ \mathsf{lvar}(\exists X.A) &\equiv \begin{cases} \mathsf{undefined} & (\mathsf{lvar}(A) = X), \\ \mathsf{lvar}(A) & (\text{otherwise}). \end{cases} \end{aligned}$$

2. The left depth $\mathsf{ldep}(A)$ *is the depth from the root to the leftmost variable in the syntax tree of* A*. It does not depend on whether the variable is free or bound.* $\mathsf{ldep}(A)$ *is defined by*

$$\begin{aligned} \mathsf{ldep}(X) &= 0, \\ \mathsf{ldep}(A \rightarrow B) &= \mathsf{ldep}(A) + 1, \\ \mathsf{ldep}(\exists X.A) &= \mathsf{ldep}(A) + 1. \end{aligned}$$

3. The left-bound-variable depth $\mathsf{lbdep}(A)$ *is* $\mathsf{ldep}(B)$ *when* A *includes a subexpression* $\exists X.B$ *and the leftmost variable of* A *is bound by this quantifier. When the leftmost variable of* A *is free,* lbdep *is undefined.* $\mathsf{lbdep}(A)$ *is defined by*

$$\begin{aligned} \mathsf{lbdep}(X) &= \mathsf{undefined}, \\ \mathsf{lbdep}(A \rightarrow B) &= \mathsf{lbdep}(A), \\ \mathsf{lbdep}(\exists X.A) &= \begin{cases} \mathsf{ldep}(A) & (\mathsf{lvar}(A) = X), \\ \mathsf{lbdep}(A) & (\text{otherwise}). \end{cases} \end{aligned}$$

Lemma 1. *1.* $\mathsf{ldep}(A[Y := B]) \neq \mathsf{ldep}(A)$ *implies* $\mathsf{lvar}(A) \equiv Y$*,*

2. $\mathsf{lvar}(A) \equiv X$ *implies* $\mathsf{lbdep}(A[X := B]) = \mathsf{lbdep}(B)$ *and* $\mathsf{lvar}(A[X := B]) \equiv \mathsf{lvar}(B)$*.*

Proof. 1. By induction on A.

When A is $X (X \not\equiv Y)$, $\mathsf{ldep}(X[Y := B]) = \mathsf{ldep}(X)$ holds.

When A is Y, we have $\mathsf{lvar}(Y) \equiv Y$.

When A is $C{\to}D$, we have

$$\begin{aligned}\mathsf{ldep}((C{\to}D)[Y := B]) &= \mathsf{ldep}((C[Y := B]){\to}(D[Y := B])) \\ &= \mathsf{ldep}(C[Y := B]) + 1\end{aligned}$$

and

$$\mathsf{ldep}(C{\to}D) = \mathsf{ldep}(C) + 1.$$

Therefore, if $\mathsf{ldep}((C{\to}D)[Y := B]) \neq \mathsf{ldep}(C{\to}D)$ holds, we have $\mathsf{ldep}(C[Y := B]) \neq \mathsf{ldep}(C)$. From the induction hypothesis, $\mathsf{ldep}(C[Y := B]) \neq \mathsf{ldep}(C)$ implies $\mathsf{lvar}(C) \equiv Y$. Hence, $\mathsf{lvar}(C{\to}D) \equiv Y$ holds.

When A is $\exists X.C$, we have

$$\mathsf{ldep}((\exists X.C)[Y := B]) = \mathsf{ldep}(C[Y := B]) + 1$$

and

$$\mathsf{ldep}(\exists X.C) = \mathsf{ldep}(C) + 1.$$

Therefore, if $\mathsf{ldep}((\exists X.C)[Y := B]) \neq \mathsf{ldep}(\exists X.C)$ holds, we have $\mathsf{ldep}(C[Y := B]) \neq \mathsf{ldep}(C)$. From the induction hypothesis, $\mathsf{ldep}(C[Y := B]) \neq \mathsf{ldep}(C)$ implies $\mathsf{lvar}(C) \equiv Y$. Hence, $\mathsf{lvar}(\exists X.C) \equiv Y$ holds.

2. By induction on A.

When A is a variable, since we have $\mathsf{lvar}(A) \equiv X$, A is X. Hence, we have $\mathsf{lbdep}(X[X := B]) = \mathsf{lbdep}(B)$ and $\mathsf{lvar}(X[X := B]) \equiv \mathsf{lvar}(B)$.

When A is $C{\to}D$, if $\mathsf{lvar}(C{\to}D) \equiv Y$ holds, we have $\mathsf{lvar}(C) \equiv Y$. From the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lbdep}(C[Y := B]) = \mathsf{lbdep}(B)$, and so we have

$$\begin{aligned}\mathsf{lbdep}((C{\to}D)[Y := B]) &= \mathsf{lbdep}(C[Y := B]{\to}D[Y := B]) \\ &= \mathsf{lbdep}(C[Y := B]) \\ &= \mathsf{lbdep}(B).\end{aligned}$$

Moreover, from the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B)$, and so we have

$$\begin{aligned}\mathsf{lvar}((C{\to}D)[Y := B]) &\equiv \mathsf{lvar}(C[Y := B]{\to}D[Y := B]) \\ &\equiv \mathsf{lvar}(C[Y := B]) \\ &\equiv \mathsf{lvar}(B).\end{aligned}$$

When A is $\exists X.C$, if $\mathsf{lvar}(\exists X.C) \equiv Y$ holds, we have $\mathsf{lvar}(C) \equiv Y \not\equiv X$. We can suppose that X is not contained freely in B by renaming the bound variable X. From the induction hypothesis, we have $\mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B) \not\equiv X$, and then we have

$$\begin{aligned}\mathsf{lbdep}((\exists X.C)[Y := B]) &= \mathsf{lbdep}(\exists X.C[Y := B]) \\ &= \mathsf{lbdep}(C[Y := B]).\end{aligned}$$

From the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lbdep}(C[Y := B]) = \mathsf{lbdep}(B)$. Hence we have $\mathsf{lbdep}((\exists X.C)[Y := B]) = \mathsf{lbdep}(B)$. Furthermore, from the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B)$ and so we have

$$\begin{aligned}\mathsf{lvar}((\exists X.C)[Y := B]) &\equiv \mathsf{lvar}(\exists X.C[Y := B]) \\ &\equiv \mathsf{lvar}(C[Y := B]) \\ &\equiv \mathsf{lvar}(B),\end{aligned}$$

since $\mathsf{lvar}(C[Y := B]) \not\equiv X$ holds. □

By Lemma 1, the following key lemma is proved.

Lemma 2. *If $\Gamma \vdash x\langle\neg_O\exists X.X, x\rangle : A$ is derivable in DF-$\lambda^{\rightarrow\exists}$, then Γ contains $x : \neg_O\exists X.X$.*

Proof. Suppose that $\Gamma \vdash x\langle\neg_O\exists X.X, x\rangle : A$ is derivable, and the type assignment for x in Γ be $x : C_x$.

Since the term $x\langle\neg_O\exists X.X, x\rangle$ is typable, its type derivation is as follows for some C.

$$\dfrac{\dfrac{}{\Gamma \vdash x : \exists Y.C{\rightarrow}A}\,(Ax) \quad \dfrac{\dfrac{}{\Gamma \vdash x : C[X := \neg_O\exists X.X]}\,(Ax)}{\Gamma \vdash \langle\neg_O\exists X.X, x\rangle : \exists Y.C}\,(\exists I)}{\Gamma \vdash x\langle\neg_O\exists X.X, x\rangle : A}\,(\rightarrow E)$$

From the form of (Ax) rules in the derivation, we have

$$\exists Y.C{\rightarrow}A \equiv C_x \equiv C[Y := \neg_O\exists X.X].$$

Then we have

$$\begin{aligned}&\mathsf{ldep}(C_x) = \mathsf{ldep}(\exists Y.C{\rightarrow}A) = \mathsf{ldep}(\exists Y.C) + 1 = \mathsf{ldep}(C) + 2, \\ &\mathsf{ldep}(C_x) = \mathsf{ldep}(C[Y := \neg_O\exists X.X]).\end{aligned}$$

Hence we have $\mathsf{ldep}(C[Y := \neg_O\exists X.X]) \neq \mathsf{ldep}(C)$, and then $\mathsf{lvar}(C) \equiv Y$ holds by Lemma 1.1. By Lemma 1.2, we have

$$\mathsf{lbdep}(C[Y := \neg_O\exists X.X]) = \mathsf{lbdep}(\neg_O\exists X.X) = \mathsf{lbdep}(\exists X.X) = \mathsf{ldep}(X) = 0.$$

On the other hand, since $\mathsf{lvar}(C) \equiv Y$ holds, we have

$$\mathsf{lbdep}(\exists Y.C{\rightarrow}A) = \mathsf{lbdep}(\exists Y.C) = \mathsf{ldep}(C).$$

Therefore, we have $\mathsf{ldep}(C) = 0$, and then C must be a variable. Hence, C is identical to Y because of $\mathsf{lvar}(C) \equiv Y$, and therefore C_x must be $\neg_O\exists X.X$. □

Then we can show the following proposition, from which $\mathrm{TC}_0 \leq \mathrm{TI}$ follows directly.

Proposition 2. *For a closed* DF-$\lambda^{\to\exists}$*-term* M *and a* $\to\exists$*-type* A*, we can effectively construct a closed* DF-$\lambda^{\to\exists}$*-term* $J_{M,A}$ *such that* $\vdash M : A$ *is derivable if and only if* $\vdash J_{M,A} : B$ *is derivable for some type* B*.*

Proof. Define $J_{M,A}$ as $\lambda x.(\lambda y.x\langle A, M\rangle)(x\langle\neg_O\exists X.X, x\rangle)$, where both x and y are fresh variables. It is easy to see that $\vdash M : A$ implies $\vdash J_{M,A} : \neg_O\neg_O\exists X.X$, because $x : \neg_O\exists X.X \vdash x\langle A, M\rangle : O$ is derivable as follows.

$$\dfrac{\dfrac{}{x : \neg_O\exists X.X \vdash x : \neg_O\exists X.X}\ (Ax) \quad \dfrac{\begin{array}{c}\vdots\\ \vdash M : A\end{array}}{\vdash \langle A, M\rangle : \exists X.X}\ (\exists I)}{x : \neg_O\exists X.X \vdash x\langle A, M\rangle : O}\ (\to E)$$

For the converse direction, we use Lemma 2. Suppose that $\vdash J_{M,A} : B$ is derivable for some B. Since $J_{M,A}$ includes $x\langle\neg_O\exists X.X, x\rangle$ as a subterm, the derivation of $\vdash J_{M,A} : B$ includes a derivation of $\Gamma \vdash x\langle\neg_O\exists X.X, x\rangle : B'$ for some Γ and B' as a subderivation. Then Γ contains $x : \neg_O\exists X.X$ by Lemma 2. Because of this, the derivation of $J_{M,A}$ has to include a subderivation of $x : \neg_O\exists X.X \vdash x\langle A, M\rangle : O$ which is the same as the above one. Hence it includes the derivation of $\vdash M : A$. □

Proof (of Theorem 1.1). TI $\leq$ TC is proved in Proposition 1. TC $\leq$ TC_0 is easily proved in a similar way to Proposition 1. $\mathrm{TC}_0 \leq$ TI immediately follows from Proposition 2. □

5 TC and TI are Equivalent in DF-$\lambda^{\neg\wedge\exists}$

In this section, we prove that TC and TI are equivalent in DF-$\lambda^{\neg\wedge\exists}$.

The proof is similar to DF-$\lambda^{\to\exists}$, and consists of the following four parts: (i) TC $\leq$ TC_0, (ii) $\mathrm{TC}_0 \leq$ TI, (iii) TI $\leq$ TI_0, and (iv) $\mathrm{TI}_0 \leq$ TC. In contrast to DF-$\lambda^{\to\exists}$, neither (i) nor (iii) is easy to prove for DF-$\lambda^{\neg\wedge\exists}$ due to absence of implication.

5.1 Translation to Closed Terms

First, we show TC$\leq$$\mathrm{TC}_0$ and TI$\leq$$\mathrm{TI}_0$. In DF-$\lambda^{\neg\wedge\exists}$, we cannot type the term $\lambda x_1.\cdots\lambda x_n.M$, because N has to be typed with $\bot$ in order to type the lambda abstraction $\lambda x.N$. Therefore, we define a construction $\underline{\lambda}x.M$, which can be considered as an interpretation of the implication introduction in DF-$\lambda^{\neg\wedge\exists}$. It should be noted that the construction can be defined as long as we have negation and conjunction, and so existence is not essential for the discussion in this subsection.

Definition 4. *For any* $\neg\wedge\exists$*-types* A *and* B*,* $A{\Rightarrow}B$ *denotes the type* $\neg(A \wedge \neg B)$*. Similarly to the ordinary implication* $\to$*,* $A_1{\Rightarrow}\cdots{\Rightarrow}A_n{\Rightarrow}B$ *denotes* $A_1{\Rightarrow}(\cdots{\Rightarrow}(A_n{\Rightarrow}B))$*. For a* DF-$\lambda^{\neg\wedge\exists}$*-term* M *and a variable* x*, we define* $\underline{\lambda}x.M$ *as* $\lambda c.(\lambda x.(c\pi_2)M)(c\pi_1)$*, where* c *is a fresh term variable.*

It is easy to see that the set of free variables of $\underline{\lambda}x.M$ is the set obtained by removing x from the set of free variables of M.

Lemma 3. *$\Gamma, x : A \vdash M : B$ holds if and only if $\Gamma \vdash \underline{\lambda}x.M : A \Rightarrow B$.*

Proof. Suppose that $\Gamma, x : A \vdash M : B$ holds, and then we have the following type derivation for $\Gamma \vdash \underline{\lambda}x.M : \neg(A \wedge \neg B)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{c : A \wedge \neg B \vdash c : A \wedge \neg B}}{c : A \wedge \neg B \vdash c\pi_2 : \neg B} \quad \begin{array}{c}\vdots\\ \Gamma, x : A \vdash M : B\end{array}}{\Gamma, c : A \wedge \neg B, x : A \vdash (c\pi_2)M : \bot}}{\Gamma, c : A \wedge \neg B \vdash \lambda x.(c\pi_2)M : \neg A} \quad \dfrac{\dfrac{}{c : A \wedge \neg B \vdash c : A \wedge \neg B}}{c : A \wedge \neg B \vdash c\pi_1 : A}}{\Gamma, c : A \wedge \neg B \vdash (\lambda x.(c\pi_2)M)(c\pi_1) : \bot}}{\Gamma \vdash \underline{\lambda}x.M : \neg(A \wedge \neg B)}$$

Conversely, if we have $\Gamma \vdash \underline{\lambda}x.M : \neg(A \wedge \neg B)$, then its derivation must be in the above form. □

Proposition 3. *1. TC in DF-$\lambda^{\neg\wedge\exists}$ is Turing reducible to TC_0 in DF-$\lambda^{\neg\wedge\exists}$.*
2. TI in DF-$\lambda^{\neg\wedge\exists}$ is Turing reducible to TI_0 in DF-$\lambda^{\neg\wedge\exists}$.

Proof. 1. For a given instance $x_1 : A_1, \cdots, x_n : A_n \vdash M : B$ of TC, we can effectively construct an instance $\vdash \underline{\lambda}x_1.\cdots\underline{\lambda}x_n.M : A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow B$ of TC_0, which is equivalent to the given instance by Lemma 3.

2. For a given DF-$\lambda^{\neg\wedge\exists}$-term M, let the list of free variables of M be $x_1, \cdots, x_n$. It should be noted that we can effectively construct the list. We show that the instance $\vdash \underline{\lambda}x_1.\cdots\underline{\lambda}x_n.M$:? of TI_0 is equivalent to the given instance $? \vdash M$:? of TI.

If $\Gamma \vdash M : B$ is derivable for some Γ and B, then $\Gamma' \vdash M : B$ is derivable, where Γ' is $\{x_1 : A_1, \cdots, x_n : A_n\}$ and each $x_i : A_i$ is contained in Γ. Then we have $\vdash \underline{\lambda}x_1.\cdots\underline{\lambda}x_n.M : A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow B$ by Lemma 3, hence $\underline{\lambda}x_1.\cdots\underline{\lambda}x_n.M$ has a type.

Conversely, if $\underline{\lambda}x_1.\cdots\underline{\lambda}x_n.M$ has a type, then M has some type because it is a subterm of $\underline{\lambda}x_1.\cdots\underline{\lambda}x_n.M$. □

From the point of view of logic, the translation $\underline{\lambda}x.M$ becomes clearer. The judgment $x : A \vdash M : B$ implicitly means the implication $A \to B$. Since DF-$\lambda^{\neg\wedge\exists}$ has no implication, we have to interpret the implication by means of negation and conjunction. In order to do that, we use the well-known fact that $A \to B$ is (classically) equivalent to $\neg(A \wedge \neg B)$, which is denoted by $A \Rightarrow B$ in this paper. Since we cannot conclude B from $A \Rightarrow B$ and A in the intuitionistic logic, $A \Rightarrow B$ is not an intuitionistic implication. We can consider an elimination rule for $\Rightarrow$ such as

$$\dfrac{\Gamma_1 \vdash M : A \Rightarrow B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M \underline{@} N : \neg\neg B},$$

where $M \underline{@} N$ is defined as $\lambda k.M\langle N, k\rangle$. We can consider that the constructions $\underline{\lambda}x.M$ and $M \underline{@} N$ realize the interpretation of the variant of implication, which is implicitly implemented by "$\vdash$", in DF-$\lambda^{\neg\wedge\exists}$.

The translation which maps $\lambda x.M$ to $\underline{\lambda} x.M$ and MN to $M\underline{@}N$ is also important from the point of view of computer science, because it can be considered as a variant of continuation-passing-style translations into the lambda calculus with continuation types and product types. Such translations have been studied in [18, 5, 8].

In addition, note that we can construct a simpler closed term N and a type C from a given instance $x_1 : A_1, \cdots, x_n : A_n \vdash M : B$ of TC. N and C can be defined as follows:

$$N \equiv \lambda c.(\lambda x_1.\cdots(\lambda x_{n-1}.(\lambda x_n.(c\pi^{n+1}_{n+1})M)(c\pi^{n+1}_n))(c\pi^{n+1}_{n-1})\cdots)(c\pi^{n+1}_1) : \bot,$$
$$C \equiv \neg(A_1 \wedge \cdots \wedge A_n \wedge \neg B),$$

where π^n_m is the m-th projection for n-tuples, which can be constructed by π_1 and π_2. Then we can show that $x_1 : A_1, \cdots, x_n : A_n \vdash M : B$ holds if and only if $\vdash N : C$ holds. This construction can be used to prove $\mathrm{TI} \leq \mathrm{TI}_0$ as well.

5.2 Proof of Equivalence

We complete the proof of equivalence in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. $\mathrm{TC}_0 \leq \mathrm{TI}$ can be proved similarly to $\mathsf{DF}\text{-}\lambda^{\to\exists}$ by replacing $\neg_O$ by $\neg$.

Lemma 4. *If $\Gamma \vdash x\langle\neg\exists X.X, x\rangle : A$ is derivable in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$, then Γ contains $x : \neg\exists X.X$.*

Proof. The definitions of the auxiliary functions for negation and conjunction are

$$\mathsf{lvar}(\neg A) \equiv \mathsf{lvar}(A),$$
$$\mathsf{lvar}(A \wedge B) \equiv \mathsf{lvar}(A),$$

$$\mathsf{ldep}(\neg A) = \mathsf{ldep}(A) + 1,$$
$$\mathsf{ldep}(A \wedge B) = \mathsf{ldep}(A) + 1,$$

$$\mathsf{lbdep}(\neg A) = \mathsf{lbdep}(A),$$
$$\mathsf{lbdep}(A \wedge B) = \mathsf{lbdep}(A),$$

and the same statement as Lemma 1 holds for $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. Hence the claim is proved similarly to Lemma 2. □

The following proposition is also proved similarly to $\mathsf{DF}\text{-}\lambda^{\to\exists}$.

Proposition 4. *For a closed $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$-term M and a $\neg \wedge \exists$-type A, we can effectively construct a closed $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$-term $J'_{M,A}$ such that $\vdash M : A$ is derivable if and only if $\vdash J'_{M,A} : B$ is derivable for some type B.*

Proof. Define $J'_{M,A}$ as $\lambda x.(\lambda y.x\langle A, M\rangle)(x\langle\neg\exists X.X, x\rangle)$, where both x and y are fresh variables. The proof is similar to Proposition 2, using Lemma 4. Note that the lambda abstractions and function applications in $J'_{M,A}$ correspond to the introduction and elimination rules of negation. □

The theorem for $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ is proved as follows.

Proof (of Theorem 1.2). $\mathrm{TC} \leq \mathrm{TC}_0$ and $\mathrm{TI} \leq \mathrm{TI}_0$ are proved by Proposition 3. $\mathrm{TC}_0 \leq \mathrm{TI}$ immediately follows from Proposition 4. $\mathrm{TI}_0 \leq \mathrm{TC}$ is easily proved similarly to $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$, that is, each instance $\vdash M :?$ of TI_0 can be translated to an equivalent instance $\vdash \lambda y.(\lambda x.y)M : \neg\bot$ of TC. □

6 Concluding Remarks

In this paper, we show equivalence between the type checking and the type inference in the domain-free lambda calculi with existential types: $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

As another style for existential types, we can consider a system with no type annotations in terms. The system was introduced in [16], and it has the following rules for existential types.

$$\frac{\Gamma \vdash N : A[X := B]}{\Gamma \vdash \langle \exists, N\rangle : \exists X.A}\ (\exists\mathsf{I}) \qquad \frac{\Gamma_1 \vdash M : \exists X.A \quad \Gamma_2, x : A \vdash N : C}{\Gamma_1, \Gamma_2 \vdash M[x.N] : C}\ (\exists\mathsf{E})$$

Fujita and Schubert [6] call such a style the *type-free style*, and they showed that TC and TI are undecidable in the type-free-style $\lambda^{\rightarrow\exists}$ by the reduction of the second-order unification problem. However, decidability of the problems in the type-free-style $\lambda^{\neg\wedge\exists}$ has not been studied. The direct relations between TC and TI in these type-free-style calculi with existence are not known, either. The technique in this paper essentially use type annotations in terms, and so it cannot be directly adapted to the type-free-style calculi.

The undecidability of the type inference is a negative result for automatic check on safety of program execution. Therefore it is also future work to study on type systems with the existential type in which the type inference problems are decidable. In particular, the type annotations for existential types such as $\langle A, M\rangle^{\exists X.B}$ correspond to the signatures in Starndard ML, and so it is important future work to study on the type-related problems in the type systems with such type annotations.

Acknowledgments We are grateful to Professor Santiago Escobar and the anonymous referees for comments to the previous version of this paper.

References

1. G. Barthe and M.H. Sørensen, Domain-free pure type systems. *Journal of Functional Programming* 10:412–452, 2000.

2. O. Danvy and A. Fillinski, Representing Control: a Study of the CPS Translation. *Mathematical Structures in Computer Science* 2(4):361–391, 1992.
3. K. Fujita, Explicitly typed $\lambda\mu$-calculus for polymorphism and call-by-value. In *Proceedings of 4th International Conference on Typed Lambda Calculi and Applications (TLCA 1999)*, LNCS 1581, pp. 162–177, 1999.
4. K. Fujita and A. Schubert, Partially Typed Terms between Church-Style and Curry-Style. In *International Conference IFIP TCS 2000*, LNCS 1872, pp. 505–520, 2000.
5. K. Fujita, Galois embedding from polymorphic types in to existential types. In *Proceedings of 7th International Conference on Typed Lambda Calculi and Applications (TLCA 2005)*, LNCS 3461, pp. 194–208, 2005.
6. K.Fujita and A.Schubert, Existential Type Systems with No Types in Terms. In *Proceedings of 9th International Conference on Typed Lambda Calculi and Applications (TLCA 2009)*, LNCS 5068, pp. 112–126, 2009.
7. R. Harper and M. Lillibridge, Polymorphic Type Assignment and CPS Conversion. *Lisp and Symbolic Computation*, 6:361–380, 1993.
8. M. Hasegawa, Relational parametricity and control. *Logical Methods in Computer Science*, 2(3:3):1–22, 2006.
9. M. Hasegawa, Unpublished manuscript, 2007.
10. A.J. Kfoury, J. Tiuryn and P. Urzyczyn, The Undecidability of the Semi-unification Problem. *Information and Computation* 102:83–101, 1993.
11. J.C. Mitchell and G.D. Plotkin, Abstract types have existential type. *ACM Transactions on Programming Languages and Systems* 10(3):470–502, 1988.
12. K. Nakazawa, M. Tatsuta, Y. Kameyama, and H. Nakano, Undecidability of Type-Checking in Domain-Free Typed Lambda-Calculi with Existence. In *the 17th EACSL Annual Conference on Computer Science Logic (CSL 2008)*, LNCS 5213, pp. 477–491, 2008.
13. K. Nakazawa and M. Tatsuta, Type Checking and Inference for Polymorphic and Existential Types. In *the 15th Computing: the Australasian Theory Symposium (CATS 2009)*, Conferences in Research and Practice in Information Technology (CRPIT), Vol. 94, 2009.
14. M. Parigot, $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LNCS 624, pp.190–201, 1992.
15. A. Schubert, Second-order unification and type inference for Church-style polymorphism. In *the 25th Annual ACM Symposium on Principles of Programming Languages (POPL '98)*, pp.279–288, 1998.
16. M. Tatsuta, Simple saturated sets for disjunction and second-order existential quantification. In *Proceedings of 8th International Conference on Typed Lambda Calculi and Applications (TLCA 2007)*, LNCS 4583, pp. 366–380, 2007.
17. M. Tatsuta, K. Fujita, R. Hasegawa, and H. Nakano, Inhabitance of Existential Types is Decidable in Negation-Product Fragment. In *Proceedings of 2nd International Workshop on Classical Logic and Computation (CLC2008)*, 2008.
18. H. Thielecke, Categorical Structure of Continuation Passing Style. Ph.D. Thesis, University of Edinburgh, 1997.
19. J.B. Wells, Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings of 9th Symposium on Logic in Computer Science (LICS '94)*, pp. 176–185, 1994.