TECHNICAL REPORT 95-1

On Implicit Arguments

Masami Hagiya and Yozo Toda

January 1, 1995

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TECHNICAL REPORT 95-1

TITLE
On Implicit Arguments

AUTHORS
Masami Hagiya and Yozo Toda

KEY WORDS AND PHRASES
implicit argument, implicit syntax, pure type system, typed lambda-calculus, type theory

ABSTRACT
A typechecker for a typed λ-calculus having implicit arguments is presented. The typechecker works in such a way that the uniqueness of implicit arguments is always preserved during reduction. Consequently, when it compares two terms by reduction, it can reduce them without inferring implicit arguments. Before describing the typechecker, we analyze various situations where the uniqueness of implicit arguments is not preserved by naïvely defined reduction.

REPORT DATE	**WRITTEN LANGUAGE**
January 1, 1995	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
31	12

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
The preliminary version of this paper appeared in *Logic, Language and Computation,* Lecture Notes in Computer Science, Vol.792, 1994, pp.10-30.

DISTRIBUTION STATEMENT
First issue 40 copies. This technical report is available via anonymous FTP from `ftp.is.s.u-tokyo.ac.jp` (directory `/pub/tech-reports`).

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

On Implicit Arguments

Masami Hagiya
Department of Information Science, University of Tokyo
hagiya@is.s.u-tokyo.ac.jp

Yozo Toda
Information Processing Center, Chiba University
yozo@yuri.ipc.chiba-u.ac.jp

January 1, 1995

Abstract

A typechecker for a typed λ-calculus having implicit arguments is presented. The typechecker works in such a way that the uniqueness of implicit arguments is always preserved during reduction. Consequently, when it compares two terms by reduction, it can reduce them without inferring implicit arguments. Before describing the typechecker, we analyze various situations where the uniqueness of implicit arguments is not preserved by naïvely defined reduction.

1 Introduction

When we write terms in a typed λ-calculus, we often want to omit some arguments of a function that can be easily inferred from the other arguments of the function. For example, in Calculus of Constructions, if we define `andI` (and introduction) by

$$\texttt{andI} \;=\; \lambda X{:}*.\lambda Y{:}*.\lambda x{:}X.\lambda y{:}Y.\lambda Z{:}*.\lambda z{:}(X{\to}Y{\to}Z).zxy,$$

the first two arguments of `andI` corresponding to the parameters X and Y can be inferred from the next two arguments corresponding to the parameters x and y, because X and Y are the types of x and y.

Some implementations of typed λ-calculi have introduced the notion of implicit arguments. In LEGO [4], for example, one can declare a function whose type is of the form

$$\Pi x\,|\,A.\Pi y{:}B(x).C(x,y).$$

A function of this type take two arguments. While the second argument corresponding to the parameter y is an ordinary explicit argument, the first one corresponding to the parameter x is an implicit argument and inferred from the type of the second one. If the type of the second argument is $B(a)$, then the first argument is inferred to be a.

A Π-abstraction of the form $\Pi x|A.B$ is called an implicit Π-abstraction in this paper. Corresponding to an implicit Π-abstraction is an implicit λ-abstraction of the form $\lambda x|A.M(x)$. By using an implicit λ-abstraction, `andI` could be defined as follows.

$$\texttt{andI} \;=\; \lambda X|*.\lambda Y|*.\lambda x{:}X.\lambda y{:}Y.\lambda Z{:}*.\lambda z{:}(X{\to}Y{\to}Z).zxy.$$

In this definition, arguments corresponding to X and Y are implicit, and hence we write (`andI` a b) instead of (`andI` A B a b).

In this paper, we deal with two different calculi, *explicit calculus* and *implicit calculus.* In the implicit calculus, some arguments of a function are implicit and are not written. In the explicit calculus, on the other hand, all the arguments are written explicitly, though implicit arguments are distinguished from the other arguments. Since the explicit calculus is essentially an ordinary typed λ-calculus, it has good syntactical and semantical properties. However, it seems difficult to directly give a foundation to the implicit calculus. In this paper, as in the approach taken in [10], the meaning of the implicit calculus is given by defining the translation from the implicit calculus to the explicit calculus. The implicit calculus can be considered as an informal language for humans who are lazy and do not want to write long terms in the explicit calculus. In other words, it is a user-interface of the explicit calculus.

However, using the implicit calculus as a user-interface of the explicit calculus, we humans tend to think in terms of the implicit calculus rather than in terms of the explicit calculus. Therefore it seems natural that the typechecker also works at the level of the implicit calculus and is able to directly check terms in the implicit calculus. This is the approach taken in this paper. We propose a typechecker that directly checks terms in the implicit calculus.

There are some practical benefits in our approach.

- If the typechecker directly checks terms in the implicit calculus, then it can report type errors more appropriately than a typechecker that indirectly checks terms in the explicit calculus.

- In some situations, it is more efficient to do typechecking at the level of the implicit calculus than to work in the explicit calculus after completely inferring implicit arguments.

- If the typechecker directly checks terms in the implicit calculus, it can immediately reuse terms in the implicit calculus that have already been checked and stored because implicit arguments need not be inferred before they are reused.

The problem of directly checking terms in the implicit calculus is that the uniqueness of implicit arguments may not be preserved by reduction. Assume that types A and B in the implicit calculus are translated to types A_e and B_e in the explicit calculus, and A_e and B_e must be equal for a certain term to be well-typed. Since we want to do typechecking at the level of the implicit calculus, we compare A and B by reducing them to see if A_e and B_e are equal. However, even if A and B are reduced to a common reduct C, A_e and B_e may not be equal because C may not have a unique translation. If C has a unique translation, then it must be a common reduct of A_e and B_e.

$$\begin{array}{ccccccc} A & \Longrightarrow & A_e & \simeq & B_e & \Longleftarrow & B \\ \downarrow & & \downarrow & & \downarrow & & \downarrow \\ C & \Longrightarrow & C_e & \stackrel{?}{\equiv} & C'_e & \Longleftarrow & C \end{array}$$

Related Works

LEGO [4, 10] is a well-known typechecker having implicit arguments. In order to make its typechecking algorithm simple, LEGO requires that an implicit argument should precede an ordinary argument. For example, one cannot declare the constant `nil` in LEGO because it has only an implicit argument. Moreover, an implicit argument should be determined only from its succeeding ordinary arguments.

ELF [8, 9] is another typechecker with implicit arguments. A characteristic feature of ELF is that it uses deterministic higher-order unification for instantiating unknowns introduced by implicit arguments. ELF has almost no restriction on the use of implicit arguments. It only requires unknowns to be instantiated by its unification procedure.

However, according to the authors' knowledge, this paper is the first that analyses the situations where the uniqueness of a translation is not preserved by reduction. Consequently, the typechecker reported in this paper is also a new one, which directly typechecks a term in the implicit calculus.

Semantics of the functional language ML is also given by translating ML into an explicit polymorphic calculus, which is called XML in [3]. Investigations have been done not on the uniqueness of a translation, but on the uniqueness of the interpretation of (possibly more than one) translations. This problem is called coherence in general.

It seems, however, difficult to solve our problem at the semantic level, because we allow not only types but arbitrary terms as implicit arguments. Uniqueness is required at the syntactic level in our approach.

Overview of the Paper

In the rest of the paper, we first describe the two calculi of this paper. We then define the translation from the implicit calculus to the explicit calculus, and the notion of validity based on the translation. Roughly speaking, a term in the implicit calculus is said to be valid if it has a unique translation.

After the definition of validity, we analyze various situations where validity is not preserved by reduction. These analyses led to restrictions and extensions of the calculi. Our typechecker is also based on the analyses.

The typechecker is designed such that if a term is successfully typechecked, then it is guaranteed to be valid. Moreover, typability by the typechecker is preserved by reduction. Therefore, if it typechecks term M and M is reduced to M', then the unique translation of M is reduced to the unique translation of M'.

We first give an abstract formulation of the typechecker. We define typing rules for the implicit calculus and rules for translating judgements and derivations from the implicit calculus to the explicit calculus. We show that if a judgement is derived by the typing rules for the implicit calculus, then the judgement can be translated into the explicit calculus and the term in the judgement is valid.

The typechecker described in the succeeding section is considered to construct a derivation in the implicit calculus.

The treatment of implicit λ-abstractions is postponed until the last section of the paper.

2 Two Calculi and Translation

In this section, we define the two calculi and the translation between them.

Explicit Calculus

The explicit calculus gives a foundation to our approach, because it is a variant of a PTS (pure type system), and its consistency is an almost obvious consequence of that of an ordinary PTS [1, 12].

The explicit calculus is a calculus having two kinds of application and abstraction. In addition to ordinary application and abstraction, it has implicit application and implicit abstraction. The two kinds of application and abstraction in the explicit calculus have no essential difference. Only the distinction is important.

Syntax of the explicit calculus is defined by adding the following rules to that of a PTS.

$$M \ ::= \ M\,|\,N \ \mid \ \lambda x\,|\,A.M \ \mid \ \Pi x\,|\,A.B$$

$M\,|\,N$ is called *an implicit application.* In implicit application $M\,|\,N$, N is called *an implicit argument,* though it is explicitly written in the explicit calculus. $\lambda x\,|\,A.M$ and $\Pi x\,|\,A.B$ are called *implicit abstractions.* Implicit applications have the same precedence as that of ordinary applications, and implicit λ- and Π- abstractions have the same precedence as that of ordinary λ- and Π- abstractions, respectively. The β-reduction is also defined for implicit arguments.

$$(\lambda x\,|\,A.M)\,|\,N \ \ \rightarrow_\beta \ \ M[x{:=}N]$$

The usual typing rules for a PTS are augmented by the following ones.

$$\frac{\Gamma \ \vdash \ A \in s_1 \quad \Gamma,x{:}A \ \vdash \ B \in s_2}{\Gamma \ \vdash \ \Pi x\,|\,A.B \in s_3} \ ((s_1,s_2,s_3) \in \texttt{Rule})$$

$$\frac{\Gamma,x{:}A \ \vdash \ M \in B \quad \Gamma \ \vdash \ \Pi x\,|\,A.B \in s}{\Gamma \ \vdash \ \lambda x\,|\,A.M \in \Pi x\,|\,A.B} \ ((s_1,s_2,s_3) \in \texttt{Rule})$$

$$\frac{\Gamma \ \vdash \ M \in \Pi x\,|\,A.B \quad \Gamma \ \vdash \ N \in A}{\Gamma \ \vdash \ M\,|\,N \in B[x{:=}N]}$$

The definition of the explicit calculus is summarized in Table 1.

For the correctness of the typechecker, we require that the explicit calculus be **functional**. This means that a term has at most one type (modulo reduction).

abstract syntax

$$M ::= s \mid x \mid (\Pi x{:}M.M) \mid (\Pi x \,|\, M.M) \mid (\lambda x{:}M.M) \mid (\lambda x \,|\, M.M) \mid (MM) \mid (M \,|\, M)$$

reduction rules

$$(\lambda x{:}A.M)N \longrightarrow M[x{:=}N]$$

$$(\lambda x \,|\, A.M) \,|\, N \longrightarrow M[x{:=}N]$$

typing rules

$$\emptyset \ \texttt{valid} \qquad \frac{\Gamma \ \texttt{valid} \quad \Gamma \vdash A \in s}{\Gamma, x{:}A \ \texttt{valid}}$$

$$\frac{\Gamma \ \texttt{valid}}{\Gamma \vdash s_1 \in s_2} \ (\ (s_1, s_2) \in \texttt{Axiom}\ ) \qquad \frac{\Gamma \ \texttt{valid}}{\Gamma \vdash x \in A} \ (\ (x{:}A) \in \Gamma\ )$$

$$\frac{\Gamma \vdash A \in s_1 \quad \Gamma, x{:}A \vdash B \in s_2}{\Gamma \vdash (\Pi x{:}A.B) \in s_3} \ (\ (s_1, s_2, s_3) \in \texttt{Rule}\ )$$

$$\frac{\Gamma \vdash A \in s_1 \quad \Gamma, x{:}A \vdash B \in s_2}{\Gamma \vdash (\Pi x \,|\, A.B) \in s_3} \ (\ (s_1, s_2, s_3) \in \texttt{Rule}\ )$$

$$\frac{\Gamma, x{:}A \vdash M \in B \quad \Gamma \vdash (\Pi x{:}A.B) \in s}{\Gamma \vdash (\lambda x{:}A.M) \in (\Pi x{:}A.B)} \qquad \frac{\Gamma \vdash M \in (\Pi x{:}A.B) \quad \Gamma \vdash N \in A}{\Gamma \vdash (MN) \in (B[x{:=}N])}$$

$$\frac{\Gamma, x{:}A \vdash M \in B \quad \Gamma \vdash (\Pi x \,|\, A.B) \in s}{\Gamma \vdash (\lambda x \,|\, A.M) \in (\Pi x \,|\, A.B)} \qquad \frac{\Gamma \vdash M \in (\Pi x \,|\, A.B) \quad \Gamma \vdash N \in A}{\Gamma \vdash (M \,|\, N) \in (B[x{:=}N])}$$

$$\frac{\Gamma \vdash M \in A \quad \Gamma \vdash B \in s \quad A \simeq B}{\Gamma \vdash M \in B}$$

Table 1: Explicit Calculus

abstract syntax

$$M ::= s \mid x \mid (\Pi x{:}M.M) \mid (\Pi x\,|\,M.M) \mid (\lambda x{:}M.M) \mid (\lambda x\,|\,M.M) \mid (MM)$$

reduction rules

$$(\lambda x{:}A.M)N \longrightarrow M[x{:=}N]$$

Table 2: Implicit Calculus: first attempt.

Implicit Calculus

Terms in the implicit calculus are the terms we actually read and write. In other words, the implicit calculus is used as a user-interface between a human and a proof checker. Although the implicit calculus is natural to humans, it seems rather difficult to give it a foundation directly. In our approach, we define translation from the implicit calculus to the explicit calculus. The meaning of the implicit calculus is then given by the translation.

Syntax of the implicit calculus is defined by adding implicit Π- and λ-abstractions to that of a PTS. The definition of the implicit calculus is summarized in Table 2.

Translation

The translation from the implicit calculus to the explicit calculus is defined in a natural-semantics style.

$M \Longrightarrow_{\Gamma_e} M_e$ is read: implicit term M is translated to explicit term M_e under context Γ_e. Γ_e denotes a context in the explicit calculus. We also define the translation from contexts to contexts. $\Gamma \Longrightarrow \Gamma_e$ is read: context Γ in the implicit calculus is translated to context Γ_e in the explicit calculus. See Table 3 for the translation rules.

Validity

A context or a term in the explicit calculus is called a translation of a context or a term in the implicit calculus, if the latter is translated to the former. We identify two translations if they become identical after implicit arguments are reduced. Therefore, a context or a term is said to have a unique translation if its translations are all equivalent *modulo reduction inside implicit arguments.*

A context or a term in the implicit calculus is said to be valid if it has a unique translation. Following are the formal definitions.

Definition 1 (Context Translation) We say that Γ_e is a *translation* of Γ, if $\Gamma \Longrightarrow \Gamma_e$. □

Definition 2 (Context Validity) Context Γ is said to be *valid* if one of the following two conditions is satisfied.

- Γ is an empty context.
- Γ is of the form $\Gamma', x{:}A$, where Γ' is valid, and for any translation Γ'_e of Γ', there uniquely exists A_e such that $\Gamma'_e, x{:}A_e$ is a translation of $\Gamma', x{:}A$.

$$\frac{\Gamma_e \ \texttt{valid}}{s \Longrightarrow_{\Gamma_e} s} \qquad \frac{\Gamma_e \vdash x \in A_e}{x \Longrightarrow_{\Gamma_e} x}$$

$$\frac{A \Longrightarrow_{\Gamma_e} A_e \quad B \Longrightarrow_{\Gamma_e, x:A_e} B_e}{(\Pi x{:}A.B) \Longrightarrow_{\Gamma_e} (\Pi x{:}A_e.B_e)} \qquad \frac{A \Longrightarrow_{\Gamma_e} A_e \quad B \Longrightarrow_{\Gamma_e, x:A_e} B_e}{(\Pi x \,|\, A.B) \Longrightarrow_{\Gamma_e} (\Pi x \,|\, A_e.B_e)}$$

$$\frac{A \Longrightarrow_{\Gamma_e} A_e \quad M \Longrightarrow_{\Gamma_e, x:A_e} M_e}{(\lambda x{:}A.M) \Longrightarrow_{\Gamma_e} (\lambda x{:}A_e.M_e)} \qquad \frac{A \Longrightarrow_{\Gamma_e} A_e \quad M \Longrightarrow_{\Gamma_e, x:A_e} M_e}{(\lambda x \,|\, A.M) \Longrightarrow_{\Gamma_e} (\lambda x \,|\, A_e.M_e)}$$

$$\frac{M \Longrightarrow_{\Gamma_e} M_e \quad N \Longrightarrow_{\Gamma_e} N_e \quad \Gamma_e \vdash (M_e N_e) \in A_e}{(MN) \Longrightarrow_{\Gamma_e} (M_e N_e)}$$

$$\frac{M \Longrightarrow_{\Gamma_e} M_e \quad \Gamma_e \vdash (M_e \,|\, N_e) \in A_e}{M \Longrightarrow_{\Gamma_e} (M_e \,|\, N_e)}$$

$$\emptyset \Longrightarrow \emptyset \qquad \frac{\Gamma \Longrightarrow \Gamma_e \quad A \Longrightarrow_{\Gamma_e} A_e}{\Gamma, x{:}A \Longrightarrow \Gamma_e, x{:}A_e}$$

Table 3: Translation from the Implicit Calculus to the Explicit Calculus: first attempt.

□

Proposition 3 **(Context Unique Translation)** If Γ is valid, then Γ has a unique translation. □

Definition 4 **(Translation and Validity)** Assume that Γ is valid, whose unique translation is Γ_e. We say that M_e is a translation of M under Γ, if $M \Longrightarrow_{\Gamma_e} M_e$ and $\Gamma_e \vdash M_e \in A_e$ for some A_e. M is said to be *valid* under Γ if M has a unique translation under Γ. □

The above definition of validity seems natural from the viewpoint that the implicit calculus is a user-interface. Since we specify a term in the explicit calculus by writing a term in the implicit calculus, we want the former term to be uniquely determined by the latter one. However, a more serious motivation for requiring the uniqueness of a translation is that we want to compare two terms at the level of the implicit calculus.

3 Analyses of Validity

In this and next several sections, we study the calculus without implicit λ-abstractions. Implicit λ-abstractions are later introduced and discussed in Section 7.

We want to define reduction on the implicit calculus such that if M is reduced to N and M has a unique translation M_e, then N also has a unique translation N_e and M_e is reduced to N_e in the explicit calculus. If such reduction is defined, we can compare two valid terms in the implicit calculus by reducing them and checking if the results are identical.

However, naïvely defined reduction does not preserve the uniqueness of translations in general. In this section, we examine various situations where the above definition of validity causes problems. In particular, we enumerate the situations where even if term M in the implicit calculus has a unique translation, the result of naïvely reducing M does not have a unique translation.

Consider the following context.

$$\begin{aligned}
&\texttt{Nat} : *\\
&\texttt{List} : * \to *\\
&\texttt{id} : \Pi X|*.X \to X\\
&\texttt{nil} : \Pi X|*.\texttt{List}\ X\\
&\texttt{length} : \Pi X|*.\texttt{List}\ X \to \texttt{Nat}\\
&\texttt{sum} : \texttt{List Nat} \to \texttt{Nat}
\end{aligned}$$

Notice that this context is valid. In fact, it is its only translation. The following situations assume this context or its extensions.

Situation 1

(`length` ((λx:`List Nat`.x) `nil`)) has a unique translation:

$$\texttt{length}\ |\texttt{Nat}\ ((\lambda x{:}\texttt{List Nat}.x)\ (\texttt{nil}|\texttt{Nat})).$$

By naïvely reducing (`length` ((λx:`List Nat`.x) `nil`)), we obtain (`length nil`). However, (`length nil`) does not have a unique translation. It has as a translation

$$\texttt{length}\ |X\ (\texttt{nil}|X),$$

where X may be an arbitrary type available in the context.
Reason: By naïvely reducing ((λx:`List Nat`.x) `nil`) to `nil`, the information in (`List Nat`) is lost and the type of `nil` becomes undetermined.
Solution: We introduce an explicit coercion in β-reduction, i.e., ((λx:`List Nat`.x) `nil`) is reduced to (`nil`:(`List Nat`)).

In general, $(\lambda x{:}A.M)N$ is reduced to $M[x{:=}N{:}A]$, if the information in A is needed to determine implicit arguments in M or in N. If the type A is completely determined only from N, we can reduce $(\lambda x{:}A.M)N$ to $M[x{:=}N]$.

Situation 2

Extend the above context by the following variable declarations.

$$\begin{aligned}
&P : \texttt{Nat} \to *\\
&Q : *\\
&q : Q
\end{aligned}$$

Then, consider the following term.

$$\begin{array}{l}\lambda y{:}P\ (\texttt{length nil}).\\ \lambda f{:}(P\ (\texttt{length}\ ((\lambda x{:}\texttt{List Nat}.x)\ \texttt{nil})) \to Q).\\ (\lambda z{:}Q.q)\ (f\ y)\end{array}$$

It has a unique translation:

$$\begin{array}{l}\lambda y{:}P\ (\texttt{length}\ \texttt{|Nat}\ (\texttt{nil|Nat})).\\ \lambda f{:}(P\ (\texttt{length}\ \texttt{|Nat}\ ((\lambda x{:}\texttt{List Nat}.x)\ (\texttt{nil|Nat}))) \to Q).\\ (\lambda z{:}Q.q)\ (f\ y).\end{array}$$

On the other hand,

$$\begin{array}{l}\lambda y{:}P\ (\texttt{length nil}).\\ \lambda f{:}(P\ (\texttt{length}\ ((\lambda x{:}\texttt{List Nat}.x)\ \texttt{nil})) \to Q).q\end{array}$$

does not have a unique translation, because X can be an arbitrary type in

$$\begin{array}{l}\lambda y{:}P\ (\texttt{length}\ \texttt{|}X\ (\texttt{nil|}X)).\\ \lambda f{:}(P\ (\texttt{length}\ \texttt{|Nat}\ ((\lambda x{:}\texttt{List Nat}.x)\ (\texttt{nil|Nat}))) \to Q).q\end{array}$$

Reason: In the λ-abstraction $(\lambda y{:}P\ (\texttt{length nil}).\cdots)$, the type $(P\ (\texttt{length nil}))$ does not have a unique translation if y is not referenced in the body of the abstraction. After y vanishes by reduction, there is no way to determine implicit arguments in $(P\ (\texttt{length nil}))$.
Solution: We require that type A in $(\lambda x{:}A.M)$, $(\lambda x\,\texttt{|}A.M)$, $(\Pi x{:}A.B)$, $(\Pi x\,\texttt{|}A.B)$ or $(M{:}A)$ be valid under the context that these terms appear, i.e., A has a unique translation. This means that implicit arguments in A are uniquely determined by only examining A.

Situation 3

We further extend the context as follows.

$$\begin{array}{l}a : \Pi F{:}(\texttt{Nat} \to *).\Pi n{:}\texttt{Nat}.F\ n \to \texttt{Nat}\\ b : \Pi F{:}(\texttt{Nat} \to *).\Pi n\,\texttt{|Nat}.F\ n\end{array}$$

$((\lambda F{:}(\texttt{Nat} \to *).a\ F\ 3\ (b\ F))\ (\lambda n{:}\texttt{Nat.Nat}))$ has a unique translation:

$$(\lambda F{:}(\texttt{Nat} \to *).a\ F\ 3\ (b\ F\ \texttt{|}3))\ (\lambda n{:}\texttt{Nat.Nat}).$$

However, $(a\ (\lambda n{:}\texttt{Nat.Nat})\ 3\ (b\ (\lambda n{:}\texttt{Nat.Nat})))$ does not have a unique translation, because it has as a translation

$$a\ (\lambda n{:}\texttt{Nat.Nat})\ 3\ (b\ (\lambda n{:}\texttt{Nat.Nat})\ \texttt{|}k),$$

where k is arbitrary.

Reason: The uniqueness of the implicit argument 3 in (b F `|`3) depends on the fact that F is a variable. When (λn:`Nat.Nat`) is substituted for F, the typing constraint that determines the implicit argument disappears.
Solution: We restrict the process of determining implicit arguments such that even after a term is substituted for a bound variable, typing constraints on implicit arguments are preserved. This means that we can decompose an equation

$$f M_1 \cdots M_n \;=\; f N_1 \cdots N_n$$

into equations

$$M_1 = N_1,\ \cdots,\ M_n = N_n,$$

only when f is a constant, i.e., a variable for which a term is never substituted.

Situation 4

(`sum` (`id nil`)) has two translations:

$$\texttt{sum}\ (\texttt{id}\ \texttt{|}(\Pi X \texttt{|}{*}.\texttt{List}\ X)\ \texttt{nil}\ \texttt{|Nat}),$$
$$\texttt{sum}\ (\texttt{id}\ \texttt{|}(\texttt{List Nat})\ (\texttt{nil}\texttt{|Nat})).$$

On the other hand, (`sum` ((λX:$*$.λx:X.`id` x) (ΠX`|`$*$.`List` X) `nil`)) has a unique translation:

$$\texttt{sum}\ ((\lambda X{:}*.\lambda x{:}X.\texttt{id}\texttt{|}X\ x)\ (\Pi X\texttt{|}{*}.\texttt{List}\ X)\ \texttt{nil}\ \texttt{|Nat}).$$

Notice that (`sum` ((λX:$*$.λx:X.`id` x) (ΠX`|`$*$.`List` X) `nil`)) is reduced to (`sum` (`id nil`)) in two steps.
Reason: (λX:$*$.λx:X.`id` x) has a unique translation because X is a type variable. When an implicit Π-abstraction is substituted for X, the uniqueness is not preserved.
Solution: We should not allow implicit Π-abstractions to appear at an implicit or explicit argument position of a function application.

Situation 5

Let 3 be a term of type `Nat`. (`sum` ((λn:`Nat.nil`) 3)) has two translations:

$$\texttt{sum}\ ((\lambda n{:}\texttt{Nat.nil}\texttt{|Nat})\ 3),$$
$$\texttt{sum}\ ((\lambda n{:}\texttt{Nat.nil})\ 3\ \texttt{|Nat})$$

Therefore, the term (`sum` ((λn:`Nat.nil`) 3)) is not valid according to the above definition of validity.
Reason: In the second translation, the variable `nil` is used without being applied to an argument.
Solution: We may require that when a variable whose type contains implicit Π-abstractions is referenced, it should be applied to an enough number of implicit arguments so that the type of the application does not contain an implicit Π-abstraction.

abstract syntax

$$\begin{array}{rcl} M & ::= & M_0 \mid (\Pi x \mid M_0.M) \\ M_0 & ::= & s \mid x \mid (\Pi x{:}M_0.M) \mid (\lambda x{:}M_0.M_0) \mid (M_0 M_0) \mid (M_0^{\dagger} M_0) \mid (M_0{:}M_0) \end{array}$$

reduction rules

$$\begin{array}{rcl} (\lambda x{:}A.M)N & \longrightarrow & M[x{:=}N] \\ (\lambda x{:}A.M)^{\dagger}N & \longrightarrow & M[x{:=}(N{:}A)] \\ (\lambda x{:}A.M){:}B & \longrightarrow & \lambda x{:}A.M \end{array}$$

Table 4: Implicit Calculus.

4 Changes to the Calculi

According to the analyses, we make the following changes to the two calculi. See Table 4 for the implicit calculus, and Table 5 for the translation rules. Correspondingly, the abstract syntax of the explicit calculus is restricted as in Table 6.

Extensions

To overcome the problem in Situation 1, we introduce explicit coercions in the implicit calculus. A term of the form $(N{:}A)$ means that the type of term N is coerced to type A. The typechecker assumes that some occurrences of applications in the implicit calculus have been annotated with the symbol † and have the form $M^{\dagger}N$. While an ordinary β-redex $(\lambda x{:}A.M)N$ is reduced to $M[x{:=}N]$, a β-redex of the form $(\lambda x{:}A.M)^{\dagger}N$ is reduced to $M[x{:=}(N{:}A)]$. This annotation can be added while terms are typechecked. For simplicity, however, we assume that terms are annotated before typechecking.

More rigorously, we extend the syntax of the implicit calculus by adding the following rules.

$$M \ ::= \ (M{:}M) \ \mid \ M^{\dagger}M$$

Explicit coercions and annotated applications have the same precedence as that of ordinary applications. Annotated applications are translated to ordinary applications in the explicit calculus.

Restrictions

As we noted before, we first restrict the calculi so that implicit λ-abstractions are not allowed. We also impose the following conditions on the use of implicit Π-abstractions according to the above analyses, particularly Situations 4 and 5.

- An implicit Π-abstraction may only appear in a context or in the body of a Π-abstraction.
  - In a context $\Gamma_1, x{:}A, \Gamma_2$, the type A can be an implicit Π-abstraction.

$$\frac{\Gamma \Longrightarrow \Gamma_e \quad \Gamma_e \ \texttt{valid}}{s \Longrightarrow_{\Gamma_e} s}$$

$$\frac{\Gamma \Longrightarrow \Gamma_e \quad \Gamma_e \vdash (x \mid L_1 \cdots \mid L_k) \in A_e}{x \Longrightarrow_{\Gamma_e} (x \mid L_1 \cdots \mid L_k)}$$
(A_e is not an implicit Π-abstraction.)

$$\frac{A \Longrightarrow_{\Gamma_e} A_e \quad B \Longrightarrow_{\Gamma_e, x:A_e} B_e}{(\Pi x{:}A.B) \Longrightarrow_{\Gamma_e} (\Pi x{:}A_e.B_e)} \qquad \frac{A \Longrightarrow_{\Gamma_e} A_e \quad B \Longrightarrow_{\Gamma_e, x:A_e} B_e}{(\Pi x \mid A.B) \Longrightarrow_{\Gamma_e} (\Pi x \mid A_e.B_e)}$$

$$\frac{A \Longrightarrow_{\Gamma_e} A_e \quad M \Longrightarrow_{\Gamma_e, x:A_e} M_e}{(\lambda x{:}A.M) \Longrightarrow_{\Gamma_e} (\lambda x{:}A_e.M_e)}$$

$$\frac{M \Longrightarrow_{\Gamma_e} M_e \quad N \Longrightarrow_{\Gamma_e} N_e \quad \Gamma_e \vdash (M_e \ N_e \mid L_1 \cdots \mid L_k) \in A_e}{(M N) \Longrightarrow_{\Gamma_e} (M_e \ N_e \mid L_1 \cdots \mid L_k)}$$
(A_e is not an implicit Π-abstraction.)

$$\frac{M \Longrightarrow_{\Gamma_e} M_e \quad N \Longrightarrow_{\Gamma_e} N_e \quad \Gamma_e \vdash (M_e \ N_e \mid L_1 \cdots \mid L_k) \in A_e}{(M^\dagger N) \Longrightarrow_{\Gamma_e} (M_e \ N_e \mid L_1 \cdots \mid L_k)}$$
(A_e is not an implicit Π-abstraction.)

$$\frac{M \Longrightarrow_{\Gamma_e} M_e \quad A \Longrightarrow_{\Gamma_e} A_e \quad \Gamma_e \vdash M_e \in A_e}{(M{:}A) \Longrightarrow_{\Gamma_e} M_e}$$

$$\emptyset \Longrightarrow \emptyset \qquad \frac{\Gamma \Longrightarrow \Gamma_e \quad A \Longrightarrow_{\Gamma_e} A_e}{\Gamma, x{:}A \Longrightarrow \Gamma_e, x{:}A_e}$$

Table 5: Translation from the Implicit to the Explicit Calculus.

$$\begin{array}{rcl} M & ::= & M_0 \mid (\Pi x \mid M_0.M) \\ M_0 & ::= & s \mid x \mid (\Pi x{:}M_0.M) \mid (\lambda x{:}M_0.M_0) \mid (M_0 M_0) \mid (M_0 \mid M_0) \end{array}$$

Table 6: Abstract Syntax of the Explicit Calculus.

– In a Π-abstraction $\Pi x{:}A.B$ or $\Pi x|A.B$, the type B can be an implicit Π-abstraction.

Implicit Π-abstractions may not appear in other cases.

- When a term whose type contains an implicit Π-abstraction occurs, it must be applied to an enough number of implicit arguments so that the type of the application is not an implicit Π-abstraction.

These conditions are reflected in the abstract syntax in Table 4 and the translation rules in Table 5.

Reduction Rules

For the extended syntax, we have the following set of reduction rules in the implicit calculus.

$$\begin{array}{rcl} (\lambda x{:}A.M)N & \longrightarrow & M[x{:=}N] \\ (\lambda x{:}A.M)^{\dagger}N & \longrightarrow & M[x{:=}(N{:}A)] \\ (\lambda x{:}A.M){:}B & \longrightarrow & \lambda x{:}A.M \end{array}$$

Notice that the second rule also appears in the calculus of Tsuiki [11], although our motivation for introducing it is different from his.

We show that reductions in the implicit calculus can be translated to reductions in the explicit calculus.

Lemma 5 Assume that N is not an implicit Π-abstraction. If $M \Longrightarrow_{\Gamma_e} M'$ and $N \Longrightarrow_{\Gamma_e} N'$, then $M[x{:=}N] \Longrightarrow_{\Gamma_e} M'[x{:=}N']$. □

Proof) This substitution property of the translation rules is easily proved by induction on the structure of M. ■

Proposition 6 (Reduction Translation) If term M in the implicit calculus is translated to M_e and is reduced to M', then there exists a term M'_e in the explicit calculus such that M' is translated to M'_e and M_e is reduced to M'_e.

$$\begin{array}{ccc} M & \Longrightarrow_{\Gamma_e} & M_e \\ \downarrow & & \downarrow \\ M' & \Longrightarrow_{\Gamma_e} & M'_e \end{array}$$

□

Proof) We prove the proposition by induction on the structure of M. Almost all the cases are trivial. We check the cases of ordinary and annotated applications. When an ordinary application is a redex $(\lambda x{:}A.M)N$, then its translation is

$$((\lambda x{:}A_e.M_e)\ N_e\ |L_1\ \cdots\ |L_k).$$

Hence we have the following diagram.

$$\begin{array}{ccc} (\lambda x{:}A.M)N & \Longrightarrow_{\Gamma_e} & ((\lambda x{:}A_e.M_e)\ N_e\ |L_1 \cdots\ |L_k) \\ \downarrow & & \downarrow \\ M[x{:=}N] & \Longrightarrow_{\Gamma_e} & (M_e[x{:=}N_e]\ |L_1\ \cdots\ |L_k) \end{array}$$

When an annotated application is a redex $(\lambda x{:}A.M)^{\dagger}N$, then its translation is

$$((\lambda x{:}A_e.M_e)\ N_e\ |L_1\ \cdots\ |L_k).$$

We can write the following diagram.

$$\begin{array}{ccc} (\lambda x{:}A.M)^{\dagger}N & \Longrightarrow_{\Gamma_e} & ((\lambda x{:}A_e.M_e)\ N_e\ |L_1 \cdots\ |L_k) \\ \downarrow & & \downarrow \\ M[x{:=}(N{:}A)] & \Longrightarrow_{\Gamma_e} & (M_e[x{:=}N_e]\ |L_1\ \cdots\ |L_k) \end{array}$$

In the above cases, we use the subject reduction property of the explicit calculus and the substitution property of the translation rules. ■

With the existence of explicit coercions, the head of a term is defined as follows.

Definition 7 (Head) Variable x is called the *head* of term M, if

- M is x,
- M is $M_1 M_2$ or $M_1^{\dagger} M_2$ and x is the head of M_1, or
- M is $N{:}A$ and x is the head of N.

□

5 Typing Rules for the Implicit Calculus

In this section, we introduce typing rules for the implicit calculus. It is easy to show that the typechecker presented in the next section is sound with respect to these typing rules. Therefore, the soundness of the typechecker is a consequence of that of the typing rules, which is stated as follows:

> If a term in the implicit calculus is well-typed by the typing rules, then it is valid, i.e., it has a unique translation in the explicit calculus.

We prove the soundness by translating a derivation in the implicit calculus to a derivation in the explicit calculus.

We first introduce two kinds of typing judgement for terms in the implicit calculus. One is of the form $\Gamma \vdash M \Uparrow A$, and the other is of the form $\Gamma \vdash M \Downarrow A$. The judgement $\Gamma \vdash M \Uparrow A$ roughly means that the type of term M is obtained without referring to type A, and the obtained type is convertible to A. The judgement $\Gamma \vdash M \Downarrow A$, on the other hand, allows type A to be referred to while term M is checked to have a type convertible to A. The typing rules for these judgements will be given later.

As in the explicit calculus, we also introduce validity judgements for contexts in the implicit caluclus. Judgements of the form Γ `valid` are derived by the following rules.

$$\emptyset \texttt{ valid} \qquad \frac{\Gamma \texttt{ valid} \quad \Gamma \vdash A \Uparrow s}{\Gamma, x{:}A \texttt{ valid}}$$

Finally, we introduce *translation judgements* between a judgement in the implicit calculus and a judgement in the explicit calculus. We have the following three forms for translation judgements.

$$\begin{array}{ll} \textbf{(i)} & \Gamma \texttt{ valid} \implies \Gamma_e \texttt{ valid} \\ \textbf{(ii)} & \Gamma \vdash M \Uparrow A \implies \Gamma_e \vdash M_e \in A_e \\ \textbf{(iii)} & \Gamma \vdash M \Downarrow A \implies \Gamma_e \vdash M_e \in A_e \end{array}$$

In the following, we will show that if $\Gamma \vdash M \Uparrow A$, then $\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$ for some Γ_e, M_e and A_e. We also show that if $\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$, then M_e is a unique translation of M under Γ, i.e., M is valid under Γ.

To each typing rule for the implicit calculus corresponds a rule for translation judgements (called a *translation rule* in this section). For example, corresponding to the above rules for contexts in the implicit calculus, we have the following two translation rules.

$$\emptyset \texttt{ valid} \implies \emptyset \texttt{ valid}$$

$$\frac{\Gamma \texttt{ valid} \implies \Gamma_e \texttt{ valid} \quad \Gamma \vdash A \Uparrow s \implies \Gamma_e \vdash A_e \in s}{\Gamma, x{:}A \texttt{ valid} \implies \Gamma_e, x{:}A_e \texttt{ valid}}$$

We now give the typing rules for judgements in the implicit calculus together with their corresponding translation rules.

Provided that $(s_1, s_2) \in$ `Axiom`, we have the following rule.

$$\frac{\Gamma \texttt{ valid}}{\Gamma \vdash s_1 \Uparrow s_2}$$

Following is the corresponding translation rule.

$$\frac{\Gamma \texttt{ valid} \implies \Gamma_e \texttt{ valid}}{\Gamma \vdash s_1 \Uparrow s_2 \implies \Gamma_e \vdash s_1 \in s_2}$$

Provided that $(s_1, s_2, s_3) \in$ `Rule`, we have the following rule for Π-abstractions.

$$\frac{\Gamma \vdash A \Uparrow s_1 \quad \Gamma, x{:}A \vdash B \Uparrow s_2}{\Gamma \vdash \Pi x{:}A.B \Uparrow s_3}$$

For satisfying the syntactic restrictions in the preceding section, we require that A not be an implicit Π-abstraction. Following is the corresponding translation rule.

$$\frac{\begin{array}{c}\Gamma \vdash A \Uparrow s_1 \implies \Gamma_e \vdash A_e \in s_1 \\ \Gamma, x{:}A \vdash B \Uparrow s_2 \implies \Gamma_e, x{:}A_e \vdash B_e \in s_2\end{array}}{\Gamma \vdash \Pi x{:}A.B \Uparrow s_3 \implies \Gamma_e \vdash \Pi x{:}A_e.B_e \in s_3}$$

Similar rules are prepared for implicit Π-abstractions.

As for λ-abstractions, we have the following rule.

$$\frac{\Gamma, x{:}A \ \vdash \ M \Uparrow B \quad \Gamma \ \vdash \ \Pi x{:}A.B \Uparrow s}{\Gamma \ \vdash \ \lambda x{:}A.M \Uparrow \Pi x{:}A.B}$$

Neither A nor M must not be an implicit Π-abstraction. Following is the corresponding translation rule.

$$\frac{\begin{array}{c}\Gamma, x{:}A \ \vdash \ M \Uparrow B \implies \Gamma_e, x{:}A_e \ \vdash \ M_e \in B_e \\ \Gamma \ \vdash \ \Pi x{:}A.B \Uparrow s \implies \Gamma_e \ \vdash \ \Pi x{:}A_e.B_e \in s\end{array}}{\Gamma \ \vdash \ \lambda x{:}A.M \Uparrow \Pi x{:}A.B \implies \Gamma_e \ \vdash \ \lambda x{:}A_e.M_e \in \Pi x{:}A_e.B_e}$$

For coercions, we have the following rule.

$$\frac{\Gamma \ \vdash \ A \Uparrow s \quad \Gamma \ \vdash \ M \Downarrow A}{\Gamma \ \vdash \ (M{:}A) \Uparrow A}$$

Neither A nor M must not be an implicit Π-abstraction. Following is the corresponding translation rule.

$$\frac{\Gamma \ \vdash \ A \Uparrow s \implies \Gamma_e \ \vdash \ A_e \in s \quad \Gamma \ \vdash \ M \Downarrow A \implies \Gamma_e \ \vdash \ M_e \in A_e}{\Gamma \ \vdash \ (M{:}A) \Uparrow A \implies \Gamma_e \ \vdash \ M_e \in A_e}$$

We have the following rule for type convertibility.

$$\frac{\Gamma \ \vdash \ M \Uparrow A' \quad A \searrow\!\!\swarrow A'}{\Gamma \ \vdash \ M \Downarrow A}$$

$A \searrow\!\!\swarrow A'$ means that A and A' have a common reduct in the implicit calculus. The reader may wonder why no condition is required on the type A. This is because whenever a judgement of the form $\Gamma \vdash M \Downarrow A$ occurs in a derivation, $\Gamma \vdash A \Uparrow s$ is guaranteed to hold for some s. Following is the corresponding translation rule.

$$\frac{\Gamma \ \vdash \ M \Uparrow A' \implies \Gamma_e \ \vdash \ M_e \in A'_e \quad A_e \ \simeq \ A'_e}{\Gamma \ \vdash \ M \Downarrow A \implies \Gamma_e \ \vdash \ M_e \in A_e}$$

$A_e \simeq A'_e$ means that A_e is convertible to A'_e in the explicit calculus by β-reduction. By the Church-Rosser property of the explicit calculus, $A_e \simeq A'_e$ implies that A_e and A'_e have a common reduct.

For type convertibility, we also have the following rule.

$$\frac{\Gamma \ \vdash \ M \Uparrow A' \quad A' \text{ is reduced to } A}{\Gamma \ \vdash \ M \Uparrow A}$$

Instead of having a translation rule that literally corresponds to the above one, we introduce the following translation rule.

$$\frac{\begin{array}{c}\Gamma \ \vdash \ M \Uparrow A' \implies \Gamma_e \ \vdash \ M_e \in A'_e \\ \Gamma \ \vdash \ A \Uparrow s \implies \Gamma_e \ \vdash \ A_e \in s \\ A_e \ \simeq \ A'_e\end{array}}{\Gamma \ \vdash \ M \Uparrow A \implies \Gamma_e \ \vdash \ M_e \in A_e}$$

This translation rule is particularly necessary for the subject reduction property of translation judgements. For converting contexts, we also prepare the following translation rules.

$$\frac{\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma'_e \vdash M_e \in A_e \quad \Gamma \ \mathtt{valid} \Longrightarrow \Gamma_e \ \mathtt{valid} \quad \Gamma_e \simeq \Gamma'_e}{\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e}$$

$$\frac{\Gamma \vdash M \Downarrow A \Longrightarrow \Gamma'_e \vdash M_e \in A_e \quad \Gamma \ \mathtt{valid} \Longrightarrow \Gamma_e \ \mathtt{valid} \quad \Gamma_e \simeq \Gamma'_e}{\Gamma \vdash M \Downarrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e}$$

In these rules, $\Gamma_e \simeq \Gamma'_e$ means that Γ_e is convertible to Γ'_e in the explicit calculus by β-reduction.

We finally define rules for variables and applications. They are defined according to the number of arguments and whether each application is annotated or not. For example, for terms of the form $xM_2^\dagger M_3$, we have the following rule.

$$\frac{\Gamma \ \mathtt{valid} \quad \Gamma \vdash M_1 \Uparrow A_1 \quad \Gamma \vdash M_2 \Uparrow A_2 \quad \Gamma \vdash M_3 \Downarrow A_3}{\Gamma \vdash xM_2^\dagger M_3 \Downarrow A} \ (*)$$

Neither A nor M_i must not be an implicit Π-abstraction. The side condition $(*)$ consists of the following conditions.

- $x{:}A_0$ is a member of context Γ.
- A_0 is reduced to $\Pi x_1 \,|\, B_1.C_1[x_1]$.
- $C_1[x_1]$ is reduced to $\Pi x_2{:}B_2[x_1].C_2[x_1, x_2]$.
- $C_2[x_1, M_2]$ is reduced to $\Pi x_3{:}B_3[x_1].C_3[x_1, x_3]$.
- $B_2[x_1] = A_2$ and $C_3[x_1, (M_3{:}B_3[x_1])] = A$ imply $x_1 = M_1$. $(**)$
- $B_1 \searrow\!\swarrow A_1$.
- $B_2[M_1] \searrow\!\swarrow A_2$
- $C_3[M_1, (M_3{:}B_3[M_1])] \searrow\!\swarrow A$.
- $B_3[M_1]$ is used as A_3.

We have used notation such as $B_2[x_1]$ and $B_2[M_1]$ in the following sense: If $B_2[x_1]$ denotes some term B, then $B_2[M_1]$ denotes $B[x_1{:=}M_1]$. The condition $(**)$ is called a *uniqueness condition* and means that one can obtain the equation $x_1 = M_1$ from the equations $B_2[x_1] = A_2$ and $C_3[x_1, (M_3{:}B_3[x_1])] = A$ by reduction and by decomposition with constant heads. This will be explained later in detail.

Following is the corresponding translation rule.

$$\frac{\begin{array}{c}\Gamma \ \mathtt{valid} \Longrightarrow \Gamma_e \ \mathtt{valid} \\ \Gamma \vdash M_1 \Uparrow A_1 \Longrightarrow \Gamma_e \vdash M_{1e} \in A_{1e} \\ \Gamma \vdash M_2 \Uparrow A_2 \Longrightarrow \Gamma_e \vdash M_{2e} \in A_{2e} \\ \Gamma \vdash M_3 \Downarrow A_3 \Longrightarrow \Gamma_e \vdash M_{3e} \in A_{3e}\end{array}}{\Gamma \vdash xM_2^\dagger M_3 \Downarrow A \Longrightarrow \Gamma_e \vdash x \,|\, M_{1e}\ M_{2e}\ M_{3e} \in A_e} \ (*)$$

The side condition $(*)$ consists of the following.

- $x{:}A_{0e}$ is the member of context Γ_e corresponding to the member $x{:}A_0$ of Γ.
- A_0 is reduced to $\Pi x_1 \,|\, B_1.C_1[x_1]$. Correspondingly, A_{0e} is reduced to $\Pi x_1 \,|\, B_{1e}.C_{1e}[x_1]$.
- $C_1[x_1]$ is reduced to $\Pi x_2{:}B_2[x_1].C_2[x_1, x_2]$. Correspondingly, $C_{1e}[x_1]$ is reduced to $\Pi x_2{:}B_{2e}[x_1].C_{2e}[x_1, x_2]$.
- $C_2[x_1, M_2]$ is reduced to $\Pi x_3{:}B_3[x_1].C_3[x_1, x_3]$. Correspondingly, $C_{2e}[x_1, M_{2e}]$ is reduced to $\Pi x_3{:}B_{3e}[x_1].C_{3e}[x_1, x_3]$.
- $B_2[x_1] = A_2$ and $C_3[x_1, (M_3{:}B_3[x_1])] = A$ imply $x_1 = M_1$. $(**)$
- $B_{1e} \simeq A_{1e}$.
- $B_{2e}[M_{1e}] \simeq A_{2e}$
- $C_{3e}[M_{1e}, M_{3e}] \simeq A_e$.
- $B_3[M_1]$ is used as A_3. Correspondingly, $B_{3e}[M_{1e}]$ is used as A_{3e}.

The reductions in the implicit calculus and their corresponding reductions in the explicit calculus must be related by Proposition 6. Notice that the uniqueness condition is also included in the side condition.

The function part of an application can be any term. For example, corresponding to the above rule, we also have the following one.

$$\frac{\Gamma \;\vdash\; M_0 \Uparrow A_0 \quad \Gamma \;\vdash\; M_1 \Uparrow A_1 \quad \Gamma \;\vdash\; M_2 \Uparrow A_2 \quad \Gamma \;\vdash\; M_3 \Downarrow A_3}{\Gamma \;\vdash\; M_0 M_2^{\dagger} M_3 \;\Downarrow\; A} \;(*)$$

The side condition $(*)$ is defined similarly.

In the above typing rule, the conclusion is of the form $\Gamma \vdash x M_2^{\dagger} M_3 \Downarrow A$. We also prepare a rule whose conclusion is of the form $\Gamma \vdash x M_2^{\dagger} M_3 \Uparrow A$. We enumerate such typing rules according to the number of arguments and whether each application is annotated or not. Moreover, we may have variations in the side condition according to how uniqueness conditions and reductions are arranged. For example, we must prepare a variant of the above typing rule with the following side condition.

- $x{:}A_0$ is a member of context Γ.
- A_0 is reduced to $\Pi x_1 \,|\, B_1.C_1[x_1]$.
- $C_1[x_1]$ is reduced to $\Pi x_2{:}B_2[x_1].C_2[x_1, x_2]$.
- $B_2[x_1] = A_2$ implies $x_1 = M_1$. $(**)$
- $C_2[M_1, M_2]$ is reduced to $\Pi x_3{:}B_3.C_3[x_3]$.
- $B_1 \searrow\!\!\swarrow A_1$.
- $B_2[M_1] \searrow\!\!\swarrow A_2$
- $C_3[(M_3{:}B_3[M_1])] \searrow\!\!\swarrow A$.

- B_3 is used as A_3.

Notice that the uniqueness condition $(**)$ only cares about the equation $B_2[x_1] = A_2$.

As another example, consider terms of the form xM_3. We have the following typing rule.

$$\frac{\Gamma \texttt{ valid} \quad \Gamma \vdash M_1 \Uparrow A_1 \quad \Gamma \vdash M_2 \Uparrow A_2 \quad \Gamma \vdash M_3 \Uparrow A_3}{\Gamma \vdash xM_3 \Uparrow A} \ (*)$$

One possibility of the side condition $(*)$ is as follows.

- $x{:}A_0$ is a member of context Γ.
- A_0 is reduced to $\Pi x_1 | B_1.\Pi x_2 | B_2[x_2].\Pi x_3{:}B_3[x_1, x_2].C_3[x_1, x_2, x_3]$.
- $B_3[x_1, x_2] = A_3$ implies $x_2 = M_2$. $(**)$
- $B_2[x_1] = A_2$ implies $x_1 = M_1$. $(**)$
- $B_1 \searrow\!\swarrow A_1$.
- $B_2[M_1] \searrow\!\swarrow A_2$.
- $B_3[M_1, M_2] \searrow\!\swarrow A_3$.
- $C_3[M_1, M_2]$ is used as A.

In general, uniqueness conditions are constructed as follows. Assume that the conclusion of the rule is of the form $\Gamma \vdash M_0 \cdots \Downarrow A$.

1. Compute the type A_0 of the function part M_0. Let C_0 be A_0 and let i be 1.
2. Reduce C_{i-1}.
3. If C_{i-1} is not a Π-abstraction then record the equation $C_{i-1} = A$.
4. If C_{i-1} is an implicit Π-abstraction $\Pi x_i | B_i.C'[x_i]$, then let C_i be $C'[x_i]$.
5. If C_{i-1} is an explicit Π-abstraction $\Pi x_i{:}B_i.C'[x_i]$ and the corresponding application is not annotated, then record the equation $B_i = A_i$ and let C_i be $C'[M_i]$.
6. If the corresponding application is annotated, then let C_i be $C[(M_i{:}B_i)]$.
7. Select some recorded equations $B_{k_1} = A_{k_1}, \cdots, B_{k_m} = A_{k_m}$ and construct a uniqueness condition:

   $B_{k_1} = A_{k_1}, \cdots, B_{k_m} = A_{k_m}$ imply $x_j = M_j$.

   Substitute M_j for x_j in C_i and the equations recorded so far. Record the equation $B_j = A_j$.
8. Either repeat 7 or increment i and go to 2 (if C_{i-1} is a Π-abstraction). If C_{i-1} is not a Π-abstraction, one can stop here.

If the conclusion is of the form $\Gamma \vdash M_0 \cdots \Uparrow A$, Step 3 is omitted.

The *decomposition* operation used in a uniqueness condition of a typing rule is defined as follows.

- $M_1 M_2 = N_1 N_2$ is decomposed to $M_1 = N_1$ and $M_2 = N_2$ if M_1 and N_1 share the same constant head. $M_1 M_2 = N_1^\dagger N_2$, $M_1^\dagger M_2 = N_1 N_2$ and $M_1^\dagger M_2 = N_1^\dagger N_2$ are handled similarly.
- $M_1{:}M_2 = N_1{:}N_2$ is decomposed to $M_1 = N_1$ and $M_2 = N_2$ if M_1 and N_1 share the same constant head.
- $\lambda x{:}A.M = \lambda x{:}B.N$ is decomposed to $A = B$ and $M = N$.
- $\Pi x{:}A.M = \Pi x{:}B.N$ is decomposed to $A = B$ and $M = N$.
- Similarly for implicit abstractions.

In the decomposition operation, we distinguish between variables that can be bound by an abstraction and those that cannot. This means that we have divided the set of variables into two disjoint subsets. Variables in one subset can be bound by an abstraction, while variables in the other subset are called *constants* and cannot be bound by an abstraction.

The following lemma is used to prove the main theorem of this section.

Lemma 8 If $x = M$ is obtained from $M_1[x] = N_1$, $\cdots$, $M_n[x] = N_n$ by reduction and by decomposition with constant heads, and $M_i[M']$ and N_i have translations (under some context) that are convertible to each other (for $1 \leq i \leq n$), then M must have a translation that is convertible to some translation of M' in the translation of $M_i[M']$. □

Proof) By Proposition 6. ■

Following are some fundamental properties of translation judgements.

Proposition 9 (Validity in the Explicit Calculus) **(i)** If Γ `valid` $\Longrightarrow \Gamma_e$ `valid`, then Γ_e `valid` (in the explicit calculus). **(ii)** If $\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$, then $\Gamma_e \vdash M_e \in A_e$. **(iii)** If $\Gamma \vdash A \Uparrow s \Longrightarrow \Gamma_e \vdash A_e \in s$ and $\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$, then $\Gamma_e \vdash M_e \in A_e$. □

Proposition 10 (Translation) **(i)** If Γ `valid` $\Longrightarrow \Gamma_e$ `valid`, then $\Gamma \Longrightarrow \Gamma_e$ (Γ is translated to Γ_e). **(ii)** If $\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$, then $M \Longrightarrow_{\Gamma_e} M_e$ (M is translated to M_e under Γ_e). **(iii)** If $\Gamma \vdash A \Uparrow s \Longrightarrow \Gamma_e \vdash A_e \in s$ and $\Gamma \vdash M \Downarrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$, then $M \Longrightarrow_{\Gamma_e} M_e$. □

Proposition 11 (Substitution Property) Let x be a variable that is not a constant. Assume the following.

$$\Gamma \vdash M \Uparrow A \implies \Gamma_e \vdash M_e \in A_e$$

(i) If $\Gamma, x{:}A, \Gamma'[x]$ `valid` $\Longrightarrow \Gamma_e, x{:}A_e, \Gamma'_e[x]$ `valid`, then we also have

$$\Gamma, \Gamma'[M] \text{ `valid`} \implies \Gamma_e, \Gamma'_e[M_e] \text{ `valid`}.$$

(ii) If $\Gamma, x{:}A, \Gamma'[x] \vdash N[x] \Uparrow B[x] \Longrightarrow \Gamma_e, x{:}A_e, \Gamma'_e[x] \vdash N_e[x] \in B_e[x]$, then we also have

$$\Gamma, \Gamma'[M] \vdash N[M] \Uparrow B[M] \implies \Gamma_e, \Gamma'_e[M_e] \vdash N_e[M_e] \Uparrow B_e[M_e].$$

(iii) If $\Gamma, x{:}A, \Gamma'[x] \vdash B[x] \Uparrow s \Longrightarrow \Gamma_e, x{:}A_e, \Gamma'_e[x] \vdash B_e[x] \in s$ and $\Gamma, x{:}A, \Gamma'[x] \vdash N[x] \Downarrow B[x] \Longrightarrow \Gamma_e, x{:}A_e, \Gamma'_e[x] \vdash N_e[x] \in B_e[x]$, then we also have

$$\Gamma, \Gamma'[M] \ \vdash \ N[M] \Downarrow B[M] \ \Longrightarrow \ \Gamma_e, \Gamma'_e[M_e] \ \vdash \ N_e[M_e] \in B_e[M_e].$$

□

Proof) By induction on derivations. Uniqueness conditions are preserved under substitution because we do not allow decomposition with a non-constant variable as a head. ■

Proposition 12 (Subject Reduction) **(i)** If Γ `valid` $\Longrightarrow \Gamma_e$ `valid` and Γ is reduced to Γ', then Γ' `valid` $\Longrightarrow \Gamma'_e$ `valid`, where Γ_e is reduced to Γ'_e corresponding to the reduction from Γ to Γ'. **(ii)** If $\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$ and Γ and M are reduced to Γ' and M', respectively, then $\Gamma' \vdash M' \Uparrow A \Longrightarrow \Gamma'_e \vdash M'_e \in A_e$, where Γ_e and M_e are reduced to Γ'_e and M_e, respectively. **(iii)** If $\Gamma \vdash A \Uparrow s \Longrightarrow \Gamma_e \vdash A_e \in s$, $\Gamma \vdash M \Downarrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$ and Γ and M are reduced to Γ' and M', respectively, then $\Gamma' \vdash M' \Downarrow A \Longrightarrow \Gamma'_e \vdash M'_e \in A_e$, where Γ_e and M_e are reduced to Γ'_e and M_e, respectively. □

Proof) By induction on derivations and by substitution property (Proposition 11). We should have shown appropriate generation lemmata.

As a typical case, consider the following instance of a typing rule.

$$\frac{\begin{array}{c}\Gamma \ \vdash \ (\lambda x{:}A.M[x]) \Uparrow \Pi x{:}A'.B'[x] \ \Longrightarrow \ \Gamma_e \ \vdash \ (\lambda x{:}A_e.M_e[x]) \in \Pi x{:}A'_e.B'_e[x] \\ \Gamma_e \ \vdash \ N \Uparrow A'' \ \Longrightarrow \Gamma_e \ \vdash \ N_e \in A''_e \quad A'_e \ \simeq \ A''_e\end{array}}{\Gamma \ \vdash \ (\lambda x{:}A.M)N \Uparrow B'[N] \ \Longrightarrow \ \Gamma_e \ \vdash \ (\lambda x{:}A_e.M_e)N_e \in B'_e[N_e]}$$

By an appropriate generation lemma, we have

$$\Gamma, x{:}A \ \vdash \ M \Uparrow B \ \Longrightarrow \ \Gamma_e, x{:}A_e \ \vdash \ M_e \in B_e,$$

where

$$\Pi x{:}A_e.B_e[x] \ \simeq \ \Pi x{:}A'_e.B'_e[x].$$

By the Church-Rosser property of the explicit calculus, we have

$$A_e \ \simeq \ A'_e \ \text{ and } \ B_e \ \simeq \ B'_e.$$

Therefore, $A_e \simeq A''_e$ holds, and

$$\Gamma_e \ \vdash \ N \Uparrow A \ \Longrightarrow \Gamma_e \ \vdash \ N_e \in A_e.$$

By the type conversion rule, we obtain

$$\Gamma, x{:}A \ \vdash \ M \Uparrow B' \ \Longrightarrow \ \Gamma_e, x{:}A_e \ \vdash \ M_e \in B'_e.$$

Then, by substitution property,

$$\Gamma \ \vdash \ M[N] \Uparrow B'[N] \ \Longrightarrow \ \Gamma_e \ \vdash \ M_e[N_e] \in B'_e[N_e].$$

■

Proposition 13 **(Type Translation)** If $\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$, then $\Gamma \vdash A \Uparrow s \Longrightarrow \Gamma_e \vdash A_e \in s$ for some s, or A is some sort. □

Proof) By substitution property (Proposition 11) and subject reduction (Proposition 12). ■

Proposition 14 **(Validity in the Implicit Calculus)** **(i)** If Γ `valid` $\Longrightarrow \Gamma_e$ `valid`, then Γ is valid (Definition 2), i.e., Γ has a unique translation. **(ii)** If $\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$, then M is valid under Γ (Definition 4), i.e., M has a unique translation under Γ. **(iii)** If $\Gamma \vdash A \Uparrow s \Longrightarrow \Gamma_e \vdash A_e \in s$ and $\Gamma \vdash M \Downarrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$, then M is valid under Γ. □

Proof) By induction on derivations. The uniqueness condition of a typing rule for applications guarantees the uniqueness of each implicit argument.

Consider the following very simple case.

$$\frac{\begin{array}{l}\Gamma \vdash M_0 \Uparrow A_0 \Longrightarrow \Gamma_e \vdash M_{0e} \in A_{0e} \\ \Gamma \vdash M_1 \Uparrow A_1 \Longrightarrow \Gamma_e \vdash M_{1e} \in A_{1e} \\ \Gamma \vdash M_2 \Uparrow A_2 \Longrightarrow \Gamma_e \vdash M_{2e} \in A_{2e}\end{array}}{\Gamma \vdash M_0 M_2 \Uparrow A \Longrightarrow \Gamma_e \vdash M_{0e}\,|\,M_{1e}\ M_{2e} \in A_e} \quad (*)$$

The side condition $(*)$ is simplified as follows.

- A_0 is reduced to $\Pi x_1|B_1.\Pi x_2{:}B_2[x_1].C_2[x_1,x_2]$. Correspondingly, A_{0e} is reduced to $\Pi x_1|B_{1e}.\Pi x_2{:}B_{2e}[x_1].C_{2e}[x_1,x_2]$.
- $B_2[x_1] = A_2$ implies $x_1 = M_1$. $(**)$
- $B_1 \simeq A_1$.
- $B_2[M_1] \simeq A_2$.
- $C_2[M_1, M_2]$ is used as A. Correspondingly, $C_{2e}[M_{1e}, M_{2e}]$ is used as A_e.

Let M'_{1e} be any implicit argument corresponding to x_1. By type unicity of the explicit calculus, which is a consequence of the functionalilty assumption, $B_{1e}[M'_{1e}] \simeq A_{2e}$ must hold if $M_{0e}\,|\,M'_{1e}\ M_{2e}$ is well-typed in the explicit calculus. Let M'_1 be some term in the implicit calculus that has M'_{1e} as its translation. Since $B_1[M'_1]$ can be translated to $B_{1e}[M'_{1e}]$, $B_1[M'_1]$ and A_2 have translations that are convertible to each other. By Lemma 8 and the uniqueness condition $(**)$, M_1 must have a translation that is convertible to M'_{1e}.

By the induction hypothesis for the derivation of M_1, M_1 has a unique translation. This menas that any implicit argument M'_{1e} is convertible to this unique translation. ■

Theorem 15 **(i)** If Γ `valid`, then Γ `valid` $\Longrightarrow \Gamma_e$ `valid` for some Γ_e. **(ii)** If $\Gamma \vdash M \Uparrow A$, then $\Gamma \vdash M \Uparrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$ for some Γ_e, M_e and A_e. **(iii)** If $\Gamma \vdash A \Uparrow s \Longrightarrow \Gamma_e \vdash A_e \Uparrow s$ and $\Gamma \vdash M \Downarrow A$, then $\Gamma \vdash M \Downarrow A \Longrightarrow \Gamma_e \vdash M_e \in A_e$ for some M_e. □

Proof) By induction on derivations. We use subject reduction (Proposition 12), type translation (Proposition 13) and validity (Proposition 14).

As an example, consider the rule

$$\frac{\Gamma \vdash M \Uparrow A' \quad A \searrow\!\swarrow A'}{\Gamma \vdash M \Downarrow A,}$$

where

$$\Gamma \vdash A \Uparrow s \implies \Gamma_e \vdash A_e \in s.$$

Assume A and A' are reduced to B.

By induction hypothesis, we have

$$\Gamma \vdash M \Uparrow A' \implies \Gamma'_e \vdash M_e \in A'_e.$$

By type translation, we have

$$\Gamma \vdash A' \Uparrow s \implies \Gamma'_e \vdash A'_e \in s,$$

or A' is a sort.

In the former case, from $\Gamma \vdash A' \Uparrow s \implies \Gamma'_e \vdash A'_e \in s$, we can derive Γ `valid` $\implies$ Γ'_e `valid`. By validity for Γ, $\Gamma_e \simeq \Gamma'_e$ holds. Therefore, we have

$$\Gamma \vdash A' \Uparrow s \implies \Gamma_e \vdash A'_e \in s.$$

In the latter case, since A' is a sort, A is reduced to A'. By subject reduction, we have the same conclusion.

By subject reduction for the reduction from A to B,

$$\Gamma \vdash B \Uparrow s \implies \Gamma_e \vdash B_e \in s.$$

By subject reduction for the reduction from A' to B,

$$\Gamma \vdash B \Uparrow s \implies \Gamma_e \vdash B'_e \in s.$$

By validity for B, we have $B_e \simeq B'_e$. This implies $A_e \simeq A'_e$. Therefore we have

$$\Gamma \vdash M \Uparrow A \implies \Gamma_e \vdash M_e \in A_e$$

by type convertibility for translation judgements. ■

6 Typechecker

We describe our typechecker in this section. It consists of several procedures. We first explain the procedures that do typechecking, followed by the procedure for equation solving.

In this section, the notation FM may denote an annotated application $F^{\dagger}N$ as well as an ordinary application. When FM denotes an annotated application, we say that FM *is annotated.*

We assume that the set of variables is divided into two subsets: the subset of global constants and that of ordinary variables. Constants cannot be used as bound variables.

Typechecking Procedures

The typechecker consists of several procedures. The procedure `Type0`(M, Γ) typechecks term M under context Γ and returns its type. This procedure corresponds to the judgement $\Gamma \vdash M \Uparrow A$ in the previous section. It is defined as follows.

```
Type0(M, Γ) :
  save old values of unknowns, pendings and equations;
  unknowns := []; pendings := []; equations := [];
  let A = type(M, Γ) in
    patch();
    restore old values of unknowns, pendings and equations;
    return(A);
```

The procedures including `Type0` refer to three global variables; `unknowns` is a list of unknown variables, `pendings` is a list of pending calls of the procedure `Type+` (defined later), and `equations` is a list of equations (more precisely, a list of quadruples as defined later).

The above procedure `Type0`(M, Γ) calls the procedure `type`(M, Γ), which does actual typechecking by recursively traversing term M.

The procedure `patch`() checks that all the equations have been solved and all the unknowns (in `unknowns`) have been instantiated, and then invokes pending procedure calls in `pendings`. It is defined as follows.

```
patch() :
  check that equations is empty;
  check that each unknown in unknowns is instantiated;
  foreach (M, A, Γ) in pendings do
    Type+(M, A, Γ);
```

The procedure `Kind`(A, Γ) typechecks type A and returns its kind. It simply calls `Type0`(A, Γ) and checks that the result is a sort.

```
Kind(A, Γ) :
  let K = Type0(A, Γ) in
    head-normalize K;
    check that K is a sort;
    return(K);
```

The procedure `Type-`(M, A, Γ) typechecks term M and matches type A with the type of M; type A may contain unknown variables.

```
Type-(M, A, Γ) :
  let B = Type0(M, Γ) in
    match(A, B, Γ);
```

The procedure `match`(A, B, Γ) tries to instantiate unknowns in A by matching A with B. Note that unknowns do not occur in B. It is defined as follows.

```
match(A, B, Γ) :
  add_equation(A, B, Γ, true);
  solve();
```

The procedure `add_equation` adds the quadruple $(A, B, \Gamma, \mathbf{true})$ to the list `equations` of quadruples. Quadruples in `equations` are processed by the procedure `solve()` defined later.

The procedure `Type+`(M, A, Γ) typechecks term M and matches the type of M with type A; the type of M may contain unknown variables. This procedure corresponds to the judgement $\Gamma \vdash M \Downarrow A$. It is defined as follows.

```
Type+(M, A, Γ) :
    save old values of unknowns, pendings and equations;
    unknowns := []; pendings := []; equations := [];
    let B = type(M, Γ) in
        match(B, A, Γ);
        patch();
        restore old values of unknowns, pendings and equations;
        return(B);
```

Notice that this procedure is called from `patch()`.

The procedure `type`(M, Γ) recursively traverses term M and returns its type. It corresponds to the typing rules for variables and applications.

```
type(M,Γ) :
  case M of
  s ⇒    % s is a sort
    return(s') where (s,s') ∈ Axiom;
  x ⇒    % x is a variable
    return(coerce(A,Γ)) where A is the type of x under Γ;
  M:A ⇒
    check that M is not an implicit Π-abstraction;
    check that A is not an implicit Π-abstraction;
    Kind(A,Γ);
    Type+(M,A,Γ);
    return(A);
  Πx:A.B or Πx|A.B ⇒
    check that x is not a constant;
    check that A is not an implicit Π-abstraction;
    let s1 = Kind(A,Γ) in
      let s2 = Kind(B,(Γ,x:A)) in
        return(s3) where (s1,s2,s3) ∈ Rule;
  FM ⇒
    check that M is not an implicit Π-abstraction;
    let A = if F is a variable or an application then
              type(F,Γ)
            else
              Type0(F,Γ) in
      head-normalize A;
      let Πx:B.C = A in
        if FM is annotated then
          add_pending(M,B,Γ);
          return(coerce(C[x:=(M:A)],Γ));
        else
          Type-(M,B,Γ);
          return(coerce(C[x:=M],Γ));
  λx:A.M ⇒
    check that x is not a constant;
    check that M is not an implicit Π-abstraction;
    let B = Type0(M,(Γ,x:A)) in
      Kind(Πx:A.B,Γ);
      return(Πx:A.B);
```

The procedure `add_pending`(M,B,Γ) adds (M,B,Γ) to `pendings` so that `Type+`(M,B,Γ) is called in `patch()`.

For variables and applications, `type` calls the procedure `coerce`. It repeatedly introduces a new unknown correpsonding to an implicit argument while type A is an implicit Π-abstraction.

```
coerce(A,Γ) :
  while A is an implicit Π-abstraction do
    let Πx|B.C = A in
      let ? = make_unknown(B,Γ) in
        A := C[x:=?];
  return(A);
```

The procedure `make_unknown`(B, Γ) creates a new unknown whose type and context are B and Γ, respectively. It then adds the new unknown to the list `unknowns` and returns it.

Equation Solving

The procedure `solve`() tries to solve equations in `equations`. It is a nondeterminisitic procedure that repeatedly selects an element from `equations` and processes it. Each element in `equations` is a quadruple of the form (M, N, Γ, b), where M and N are terms that must be equal (i.e., they must have a common reduct), Γ is the context for M and N, and b is the flag telling whether unknowns in M may be instantiated or not. Note that unknowns only occur in M and not in N.

This procedure has two roles: one is to obtain the implicit arguments corresponding to the unknowns while garanteeing the uniqueness conditions, and the other is to reduce both sides of each equation to their common reduct.

In the following, **?** denotes an unknown; its type and context are denoted by `type`(**?**) and `context`(**?**), respectively.

```
solve() :
  while some changes are made do
    take a quadruple e from equations and remove it;
    head-normalize the two terms in e;
    case e of
    (s, s', Γ, b) ⇒
      check that sort s is equal to sort s';
    (x, x', Γ, b) ⇒
      check that variable x is equal to variable x';
    (?, N, Γ, true) ⇒
      normalize N;
      check that N is not an implicit Π-abstraction;
      check that N contains only variables in context(?);
      Type-(N, type(?), context(?));
      instantiate ? with N;
    (FM, GN, Γ, b) or (F:M, G:N, Γ, b) ⇒
      add_equation(F, G, Γ, b);
      if F begins with a constant then
        add_equation(M, N, Γ, b);
      else if F begins with a variable then
        add_equation(M, N, Γ, false);
      else
        put back e to equations;
    (λx:A.M, λx:B.N, Γ, b) or (Πx:A.M, Πx:B.N, Γ, b)
                           or (Πx|A.M, Πx|B.N, Γ, b) ⇒
      add_equation(A, B, Γ, b);
      add_equation(M, N, (Γ, x:A), b);
    otherwise ⇒
      put back e to equations;
```

Note that an unknown is only instantiated when the flag of the quadruple is **true**. Note also that before unknown **?** is instantiated with term N, the term N is normalized and typechecked by `Type-`$(N, \texttt{type}(?), \texttt{context}(?))$.

The most crucial cases of the procedure are those for applications and explicit coercions. When the head of applications FM and GN is a constant, the arguments M and N are processed with the same flag as that of the applications, but when the head is not a constant, the arguments M and N are always processed with flag **false**. This means that unknowns in M are not allowed to be instantiated.

Example

Let us give an example. Consider the following context.

$$
\begin{aligned}
&Q : * \\
&\texttt{Ex} : \Pi X\,|\,*.(X \to *) \to * \\
&\texttt{exI} : \Pi X\,|\,*.\Pi P\,|\,(X \to *).\Pi x{:}X.Px \to \texttt{Ex}\ P
\end{aligned}
$$

$$R : \mathtt{Nat} \to \mathtt{Nat} \to *$$
$$f : \mathtt{Ex}\ (\lambda x{:}\mathtt{Nat}.\mathtt{Ex}\ (\lambda y{:}\mathtt{Nat}.Rxy)) \to Q$$
$$r : R\ 1\ 2$$

The algorithm successfully typechecks the following term.

$$f^{\dagger}(\mathtt{exI}\ 1^{\dagger}(\mathtt{exI}\ 2\ r))$$

Remarks

The following remarks point out the weakness of the algorithm.

We can compare an ordinary application $M_1 M_2$ and an annotated application $N_1^{\dagger} N_2$, because if M_1 and N_1 have a common translation and M_2 and N_2 have a common translation, then $M_1 M_2$ and $N_1^{\dagger} N_2$ also have a common translation.

However, we cannot compare a coercion term with a non-coercion term in general. For example, (`sum` (`nil`:`List Nat`)) and (`sum`†`nil`) cannot be compared. In order to see that the two terms are equal (i.e., have a common translation), we must replace `nil` in the latter term with (`nil`:`List Nat`). This would, however, require the procedure `solve` to do typechecking.

Since untyped terms may not terminate in reduction, the typechecker must check that a term is well-typed before reducing it. Unfortunately, this is in general not possible because a term (and its type) may contain unknowns and it is guaranteed to be well-typed only after the unknowns have been instantiated.

Note that since the typechecker is not complete, the discussion about its termination is somewhat meaningless, because in its actual implementation, it can stop at any place where there is a danger of non-termination.

The typechecker solves equations by first-order matching, checking the occurrence of variables when an unknown is instantiated [7]. The extension of the typechecker with the so-called simple unification [5, 6] is almost straight-forward. However, continuing its formal treatment is not easy. For example, we must assume that the explicit calculus allows η-reduction.

7 Implicit λ-abstraction

So far, we assumed that the calculus did not have implicit λ-abstractions. In this section, implicit λ-abstractions are introduced and discussed.

As we restrict the use of implicit Π-abstractions, implicit λ-abstractions must also be used with restrictions. Remember that an implicit Π-abstraction may only appear in a context (or in the body of a Π-abstraction). Therefore, we allow λ-abstractions only through variables defined in the (global) context.

In addition to a variable declaration of the form $x{:}A$, a context may now contain a variable definition of the form $x{=}M$. For example, the definition

$$\mathtt{andI} \;=\; \lambda X\,|\,*.\lambda Y\,|\,*.\lambda x{:}X.\lambda y{:}Y.\lambda Z{:}*.\lambda z{:}(X{\to}Y{\to}Z).zxy.$$

in the introduction is allowed in the context.

When a variable x defined as above is referenced, it must be given an enough number of implicit arguments, so that the type of the application is not an implicit Π-abstraction. For example, `andI` is always referenced with its two implicit arguments.

When x is applied to arguments in the explicit calculus, it can be reduced by the ordinary β-reduction rule. In the implicit calculus, on the other hand, in order to reduce an application of x containing implicit arguments, we must infer the implicit arguments during reduction.

The simplest approach to cope with this problem is to allow reduction only when successive steps of β-reduction eliminate all the implicit arguments. For example, when `andI` is given only two explicit arguments M and N, then (`andI` M N) is not allowed to be reduced to

$$\lambda Z{:}*.\lambda z{:}(X{\to}Y{\to}Z).zMN,$$

because X and Y are implicit arguments. On the other hand, the term

$$\texttt{andI}\ M\ N\ A\ (\lambda x{:}A.\lambda y{:}B.x)$$

is reduced to M, because all the implicit arguments are eliminated and disappear.

Another approach is to do typechecking during reduction. When an implicit λ-abstraction $\lambda x\,|\,A.M$ is encountered, it is reduced as follows, where **?** is a newly created unknown.

$$\lambda x\,|\,A.M \ \longrightarrow\ M[x{:=}?]$$

Such an unknown is instantiated in a later reduction step. While reducing an ordinary β-redex

$$(\lambda y{:}B.N)L \ \longrightarrow\ N[y{:=}L],$$

if B contains unknowns created in a preceding reduction step of an implicit λ-abstraction, the type of L is computed and then matched with B. This seems to work well, but typechecking makes reduction very inefficient. Moreover, the formal treatment of the entire algorithm becomes complicated.

8 Conclusions

We have presented a typechecking algorithm that directly checks terms in the implicit calculus by restricting instantiation of unknowns introduced by implicit arguments. The algorithm was derived by the analyses of various situations in which the uniqueness of a translation is not preserved by reduction.

The typechecker has actually been implemented and runs with reasonable efficiency. However, users of our typechecker sometimes find it difficult to distinguish between well-typed terms and ill-typed terms. It is therefore important to characterize the terms that can be checked by our typechecker in a style that is more natural and intuitive. With such a different style of formalization, it becomes meaningful to show the completeness and the termination of the typechecker.

Finally, we want to work on the handling of implicit λ-abstractions as a future work.

Acknowledgments

We deeply thank Randy Pollack, Atsushi Ohori, Pierre-Louis Curien and many others for their invaluable advices, and Shin'ya Nishizaki for being the first and impatient user of our typechecker.

References

[1] H. Barendregt. Introduction to generalised type systems. *Journal of Functional Programming,* Vol.1, No.2, pp.124–154, 1991.

[2] V. Breazu-Tannen, T. Coquand, C. A. Gunter and A. Scedrov. Inheritance as Implicit Coercion. *Information and Computation,* Vol.93, No.1, pp.172–221, 1991.

[3] R. Harper and J. C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems,* Vol.15, No.2, pp.211–252, 1993.

[4] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Department, University of Edinburgh, 1992.

[5] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation,* Vol.1, No.4, pp.497–536, 1991.

[6] D. Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, In *Proceedings of the Eighth International Conference on Logic Programming,* MIT Press, 1991.

[7] D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation,* Vol.14, No.4, pp.321–358, 1992.

[8] F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science, Asilomar, California,* 1989.

[9] F. Pfenning. Logic programming in the LF logical Framework, In *Proceedings of the First Workshop on Logical Frameworks,* 1991.

[10] R. Pollack. Implicit syntax. LFCS, Computer Science Department, University of Edinburgh, 1992.

[11] H. Tsuiki. A normalizing calculus with subtyping and overloading. *Theoretical Aspects of Computer Software 94,* Lecture Notes in Computer Science, Vol.789, pp.273–295, 1994.

[12] L. S. van Benthem Jutting. Typing in pure type systems. *Information and Computation,* Vol.105, pp.30–41, 1993.