Publ. RIMS, Kyoto Univ.
30 (1994), 1055–1121

Simply Typed Lambda Calculus with First-Class Environments

By

Shin-ya NISHIZAKI*

Abstract

We propose a lambda calculus $\lambda_{env}^{\rightarrow}$ where it is possible to handle first-class environments. This calculus is based on the idea of *explicit substitution*, that is; $\lambda\sigma$-calculus. Syntax of $\lambda_{env}^{\rightarrow}$ is obtained by merging the class of terms and the one of substitutions. Reduction is made from the weak reduction of $\lambda\sigma$-calculus. Its type system also originates in the one of $\lambda\sigma$-calculus. Confluence of $\lambda_{env}^{\rightarrow}$ is proved by Hardin's *interpretation method* which is originally used for proving confluence of $\lambda\sigma$-calculus. We proved strong normalizability of $\lambda_{env}^{\rightarrow}$ by reducing it to strong normalizability of a simply typed record calculus. Finally, we propose a type inference algorithm which produced a principal typing for each typable term.

§1. Introduction

Environment

In many programming languages and logical systems, we can avoid several repetitions of writing an expression by using *variables*. At each position in programs or sentences, we have an assignment of values to variables, called an *environment*. We use this notion in evaluators or denotational semantics of functional languages.

First-Class Object

In programming languages, an object which can be passed to and returned from procedures, is said to be *first-class*. For example, functions are *not* first-class objects in BASIC and FORTRAN. In contrast, functions are treated as first-class object in functional languages. Programming language Scheme [2] [1], a dialect of Lisp, is equipped with first-class continuations. This

Communicated by S. Takasu, November 25, 1993. Revised March 7, 1994.
1991 Mathematics Subject Classifications: 68N15.
* Department of Information Technology, Faculty of Engineering, Okayama University, 3-1-1 Tsushima-naka, Okayama, 700, Japan.

allows us to use the powerful control mechanism like backtrack and coroutine. In many implementations like MIT-Scheme, Elk or X-Scheme, we can also treat environments as first-class objects. The basic primitives supporting the facility of first-class environment, are

- ⓐ **(the-environment)** which returns the current environment, and
- ⓑ **(eval** ⟨*list*⟩ ⟨*environment*⟩**)** which returns the result of evaluation of the expression represented by ⟨*list*⟩ under ⟨*environment*⟩.

The following is an example of these primitives:

```
(let ((l-inf-space
    (let ((norm (lambda (x) (max (abs(car x)) (abs(cdr x))))))
      (the-environment)))
;;; l^∞ space's norm: sup_{i=1,2} |x_i|
    (l1space,
      (let ((norm (lambda (x) (+ (abs(car x)) (abs(cdr x))))))
        (the-environment)))
;;; l^1 space's norm: Σ_{i=1,2} |x_i|
    (l2space
      (let ((norm
              (lambda (x) (sqrt (+ (expt (car x)2) (expt (cdr x) 2))))))
        (the-environment))))
;;; l^2 space's norm: (Σ_{i=1,2} |x_i|^2)^{1/2}
    (print (eval '(norm '(3.4)) l-inf-space))
    (print (eval '(norm '(3.4)) l1space))
    (print (eval '(norm '(3.4)) l2space)))
```

output:

```
4
7
5
```

Variable **l-inf-space, l1space,** and **l2space** are bound to the reified environments in which each variable **norm** is bound to l^∞-, l^1-, and l^2-norm's procedure, respectively. We can use a desired norm's procedure by evaluating variable **norm** in the environment where the norm's procedure is bound.

We next observe these primitives from the semantic viewpoint. The

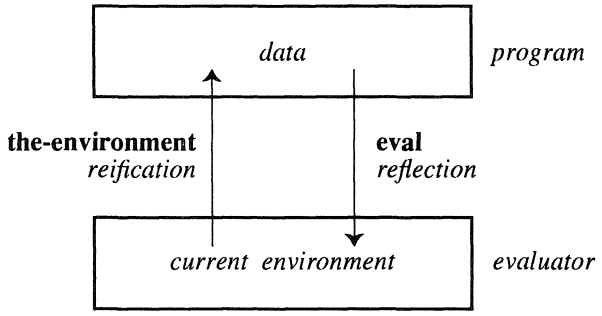
following is well-known as *environment semantics*, where these primitives are naïvely defined:

$$
\begin{aligned}
[\![\mathbf{x}]\!] &= \lambda r.(r.\mathbf{x}) \\
[\![\textbf{(lambda (x) M)}]\!] &= \lambda x'.\lambda r.[\![\mathbf{M}]\!]\langle \mathbf{x} = x' | r\rangle \\
[\![\textbf{(M N)}]\!] &= \lambda r.([\![\mathbf{M}]\!]r)([\![\mathbf{N}]\!]r) \\
[\![\textbf{(the-enviroment)}]\!] &= \lambda r.r \\
[\![\textbf{(eval 'M N)}]\!] &= \lambda r.([\![\mathbf{M}]\!]([\![\mathbf{N}]\!](r)))
\end{aligned}
$$

where $r.\mathbf{x}$ is field selection and $\langle \mathbf{x} = x' | r\rangle$ is record extension of record r by label $\mathbf{x}$ and term x'. In the environment semantics, we observe that

- **(the-environment)** corresponds to an *identity function*,
- **(eval 'M N)** to *function composition*

In terminology of reflective programming paradigm, primitive **the-environment** *reifies* the current environment and **eval** *reflects* the reified environment to the current environment.



We therefore find the following correspondence:

(the-environment) $\Leftrightarrow$ *reification* $\Leftrightarrow$ *identity function*,
(eval 'M N) $\Leftrightarrow$ *reflection* $\Leftrightarrow$ *function composition*.

We next show how to applied the idea of *explicit substitution* to our calculus.

Explicit Substitution

Lambda calculus was proposed as the general theory of functions in mathematical logic and has been widely used for describing, analyzing, and implementing programming languages. In lambda calculus, we consider lambda-terms as programs, their reduction as computation, and normal terms as observable values. We cannot find environments in reduction sequences, though the notion of environment is fundamental in functional programming languages.

$$\begin{aligned}&(\lambda x.x)M\\ &\xrightarrow{\beta} x[x:=M]\\ &\equiv M\end{aligned}$$

The reason why we cannot find environments in reduction, is that the variable reference mechanism, i.e. the substitution operation, is defined at the *meta-level.* Recently, $\lambda\sigma$-calculus is proposed by Curien et al. ([4], [12], [3], [6]) as an improvement of lambda calculus. There substitutions are treated not at *meta*-level but at *object*-level:

$$\begin{aligned}&(\lambda x.x)M\\ &\xrightarrow{\beta} x[(M/x)\cdot id]\\ &\xrightarrow{\sigma} M\end{aligned}$$

Comparing these two reduction sequences, we find that the term $x[(M/x)\cdot id]$ can be read as an application of a *substitution* $[x:=M]$ to a variable x, and also as the result which x is evaluated to under an *environment* $[x\rightarrow M]$.

This is one of principal ideas of $\lambda\sigma$-calculus *identification* of *environment with substitution* and this allows us to investigate *syntactically* the mechanism of environment.

Although it is allowed to treat environments as object-level notions in $\lambda\sigma$-calculus, these are not first-class objects: it is impossible to pass substitutions as parameters. For example, the following term

$$\lambda env.x[env]$$

is not a term of $\lambda\sigma$-calculus since it is inhibited to occur a term in [ ]. The point to be stressed is that the syntactic class of substitutions is precisely

distinguished from the one of terms in $\lambda\sigma$-calculus. Roughly speaking, the calculus defined in this paper is obtained by integration of these two syntactic class, as a result, the above term will be justified in our calculus and the integration enables first-class environment facility.

Motivations

Untyped and typed lambda calculi have achieved great successes in computer science because of their nice mathematical properties. These properties are not always preserved when we extend them by addition of new constructs. For example, lambda calculus with first-class continuations, which may be considered as classical logic from the viewpoint of Curry-Howard isomorphism, is not essentially confluent (cf. page 150 in [7]). On the other hand, strong normalizability holds in simply typed lambda calculus with first-class continuations (cf. [8]). This property justifies the intuition that bugs of non-termination do not exist in the part of continuation-handling, but anywhere else, e.g. the part of recursion. Without such theoretical background, this intuition would be nothing but superstition.

As we know from the above example, it is important to discover fundamental properties preserved in extending lambda calculus by adding first-class environments.

Outline of this paper

In **Section** 2, we introduce a simply typed lambda calculus $\lambda^{\rightarrow}_{env}$ with first-class environments: its syntax, its typing rule, and its computational rule which originates in the weak reduction of $\lambda\sigma$-calculus. Subject reduction property of $\lambda^{\rightarrow}_{env}$ is here proved.

Section 3 is devoted to *confluence* for the reduction given in **Section** 2. We use Hardin's interpretation method in proving this property, similary to the case of $\lambda\sigma$-calculus.

In **Section** 4, we prove *strong normalizability* for $\lambda^{\rightarrow}_{env}$. First, we introduce a simply typed calculus λ_{record} with records and show strong normalizability of λ_{record}. Then, we derive strong normalizability of $\lambda^{\rightarrow}_{env}$ from that of λ_{record}, by using the property that β-steps in a reduction sequence before the translation correspond exactly to the ones after the translation.

In **Section** 5, we study type inference algorithm for $\lambda^{\rightarrow}_{env}$. First, we present unification on types of $\lambda^{\rightarrow}_{env}$, originally developped by Jategaokar and Mitchell for record typing. Then, we propose a type inference algorithm for $\lambda^{\rightarrow}_{env}$ and

show that this algorithm is sound and gives us principal typing.

§2. Simply Typed Lambda Calculus with First-class Environment: $\lambda^{\rightarrow}_{env}$

In this section, we propose a typed lambda calculus with first-class environments, called $\lambda^{\rightarrow}_{env}$. The base of this calculus is a simply typed lambda calculus *à la Curry*. We give syntax, type system, and reduction to this calculus. Then its subject reduction property will be presented.

§2.1 Syntax of $\lambda^{\rightarrow}_{env}$

Definition 2.1 [Types and Raw Terms]. Three countable sets

- a set **Type Var** of *type variables*,
- a set **Term Var** of *term variables*,
- a set **Env Type Var** of *environment type variables*, and
- a mapping $PVar$ from **EnvTypeVar** to a family of finite sets of **Term Var**, called *prohibited variable assignment.*

are given in advance. And we assume that a set $PVar^{-1}(\{x_1, \cdots, x_n\})$ is infinite for an arbitrary finite subset $\{x_1, \cdots, x_n\}$ of **Term Var**.

For each environment type variable ρ, we call its image $PVar(\rho)$ the set of *prohibited variables* of ρ. When we write an environment type variable, we specify its prohibited variables by subscription. For example, we write ρ_X, if we assume $X = PVar(\rho)$.

Environment types and *types* are defined as

$$E ::= \{x_1 : A_1\} \cdots \{x_m : A_m\}\rho,$$

and as

$$A ::= \alpha \mid E \mid A \rightarrow B$$

respectively, where $m \geq 0$, $x_i \in$ **Term Var**, $\rho \in$ **Env Type Var**, $\{x_i, \cdots, x_m\} \subseteq PVar(\rho)$, $\alpha \in$ **Type Var**, $x_1, \cdots, x_m$ are distinct with each other, i.e. $\forall i \forall j((1 \leq i \leq m) \wedge (1 \leq j \leq m) \wedge (i \neq j) \Rightarrow x_i \not\equiv x_j)$.

We do not distinguish environment types by the order of bindings between variables and types. For example, we identify $\{x : A\}\{y : B\}\rho$ with $\{y : B\}\{x : A\}\rho$.

Type is a set of types defined as above.

We define a *domain dom*(E) of an environment type $E \equiv \{x_1 : A_1\} \cdots \{x_n : A_n\}\rho_X$ by $dom(E) = \{x_1, \cdots, x_n\}$

Raw terms are defined as

$$M ::= x \mid \lambda x.M \mid MN \mid id \mid (M/x)\cdot N \mid M \circ N$$

where $x \in$ **Term Var**.

We call raw terms as *terms* simply if there is no danger of confusion. ■

Intuitively, an environment type variable means a "hole" in which an environment type may be put. Its prohibited variables specify the variables which should not be included. For example, it is possible to substitute $\{y:B\}\rho'_{\{y\}}$ for $\rho_{\{z\}}$ in $\{x:A\}\rho_{\{z\}}$, but impossible to substitute $\{y:B\}\rho'_{\{y\}}$ for $\rho_{\{y\}}$ in $\{x:A\}\rho_{\{y\}}$, assuming x, y, and z are different each other.

The definition of environment type is influenced by the studies on record calculus. Environment type variable exactly corresponds to row variables in record typing, introduced by M. Wand [15]. The condition that each variable in an environment type should be distinct with each other, is required for the existence of mgu.

There are two intuitive explanations of primitive *id*: one is that it is the *identity substitution*, which is similar to *id* of $\lambda\sigma$-calculus; the other is that it is a primitive which returns a current environment, like **(the-environment)** of Scheme. In the same way, $M \circ N$ has also two explanations: one is that it is the *composition* of substitution M and substitution N, and the other that it is **(eval** M N**)** of Scheme. These explanations will be justified by the reduction given in **Section** 2.2.

Names of each syntax class and usages of alphabets for meta-variables are summarized in the following table:

$\alpha,\beta,\gamma,\cdots$	type variable
$X,Y,X',Y',\cdots$	prohibited variables
$\rho,\rho',\rho'',\cdots$	environment type variable
$A,B,C,D,\cdots$	type
$E,E',E'',\cdots$	environment type
$A \to B$	*function type*

$x,y,z,\cdots$	(term) variable
$L,M,N,\cdots$	term
$\lambda x.M$	*lambda abstraction*
MN	*application*
id	*identical environment*
$(M/x)\cdot N$	*environment extension*
$M \circ N$	*environment composition*

We may call two kinds of variables–type variables and environment type variables–*type variables* simply, if there is no danger of confusion.

We next introduce basic notions for types and terms:

Definition 2.2 [Free Type Variable]. We call a type variable and an environment type variable occurring in a type a *free type variable* and a *free environment type variables* respectively, and $ftv(A)$ is the set of free type variables and free environment type variables. ■

Definition 2.3 [Length of Type]. We define *length* $length(A)$ of a type A inductively as

$$\begin{aligned} length(\{x_1 : A_1\}\cdots\{x_n : A_n\}\rho) &= length(A_1)+\cdots+length(A_n)+2,\\ length(\alpha) &= 1,\\ length(A \to B) &= length(A)+length(B)+1. \end{aligned}$$

■

Definition 2.4 [Length of Term]. We define *length* $length(M)$ of a term M inductively as

$$\begin{aligned} length(x) &= 1,\\ length(MN) &= length(M)+length(N)+1,\\ length(\lambda x.M) &= length(M)\times 2,\\ length(M\circ N) &= length(M)\times(length(N)+1),\\ length(id) &= 1,\\ length((M/x)\cdot N) &= length(M)+length(N)+1. \end{aligned}$$

(cf. Lemma 3.3) ■

§2.2. Weak Reduction of $\lambda_{env}^{\rightarrow}$

In this section, we present an operational semantics of $\lambda_{env}^{\rightarrow}$ called the *weak reduction* originally given to $\lambda\sigma$-calculus. This reduction is weaker than β-reduction used in usual lambda calculi: the application of a substitution N to a term $\lambda x.M$, $(\lambda x.M)\circ N$ can not be reduced any more in the weak reduction. For example, $(\lambda y.y)\circ((\lambda z.z/x)\cdot id)$ is the normal form of term $(\lambda x.\lambda y.y)(\lambda z.z)$ in $\lambda_{env}^{\rightarrow}$ although we can obtain "more" normal term $\lambda y.y$ in the usual lambda calculi. It is therefore true that the weak reduction is weaker

than the usual β-reduction but this kind of weakness is common to the call-by-name strategy and the call-by-value's one, and the weak reduction is acceptable as the computational semantics in the same reason as the case of these evaluation strategies.

As a result of the trade-off with weakness of the reduction, we get syntactical simplicity of the calculus. Otherwise, we had to develop a more complex theory and gained a comparatively few advantages.

The calculus with name vs The nameless calculus

In $\lambda\sigma$-calculus, *nameless* terms or also called *de Bruijn terms* are investigated mainly rather than ones with name and the complication of renaming is avoided. It is sufficient that we only study the nameless calculus because every term can be translated to nameless terms whether typed or not. Of course, this is true in the usual calculus but not in the calculus *with first-class environments*. Consider the following term:

$$\lambda env.(x \circ env).$$

The de Bruijn index of variable x is determined by the type of variable *env*. In the lambda calculus with first-class environments, it is probably impossible to give a transformation from the calculus with name to the nameless one. Consequently, we study the calculus with name in this paper.

Definition 2.5 [Weak Reduction]. A binary relation $(-)\underset{wr}{\rightarrow}(-)$ called *weak reduction*, is defined inductively as follows:

[Substitution Rules]

$$\frac{}{(L \circ M) \circ N \underset{wr}{\rightarrow} L \circ (M \circ N)}\,Ass$$

$$\frac{}{id \circ M \underset{wr}{\rightarrow} M}\,IdL \qquad \frac{}{M \circ id \underset{wr}{\rightarrow} M}\,IdR$$

$$\frac{}{((L/x) \cdot M) \circ N \underset{wr}{\rightarrow} ((L \circ N)/x) \cdot (M \circ N)}\,DExtn$$

$$\frac{}{x\circ((M/x)\cdot N)\underset{wr}{\to}M}\,VarRef \qquad \frac{x\not\equiv y}{y\circ((M/x)\cdot N)\underset{wr}{\to}y\circ N}\,VarSkip$$

$$\frac{}{(M_1M_2)\circ N\underset{wr}{\to}(M_1\circ N)(M_2\circ N)}\,DApp$$

[Beta Rules]

$$\frac{}{((\lambda x.M)\circ L)N\underset{wr}{\to}M\circ((N/x)\cdot L)}\,Beta1 \qquad \frac{}{(\lambda x.M)N\underset{wr}{\to}M\circ((N/x)\cdot id)}\,Beta2$$

[Compatibility Rules]

$$\frac{M\underset{wr}{\to}N}{(ML)\underset{wr}{\to}(NL)}\,AppL \qquad \frac{M\underset{wr}{\to}N}{(LM)\underset{wr}{\to}(LN)}\,AppR$$

$$\frac{M\underset{wr}{\to}N}{\lambda x.M\underset{wr}{\to}\lambda x.N}\,Lam$$

$$\frac{M\underset{wr}{\to}N}{M\circ L\underset{wr}{\to}N\circ L}\,CompL \qquad \frac{M\underset{wr}{\to}N}{L\circ M\underset{wr}{\to}L\circ N}\,CompR$$

$$\frac{M\underset{wr}{\to}N}{(M/x)\cdot L\underset{wr}{\to}(N/x)\cdot L}\,ExtnL \qquad \frac{M\underset{wr}{\to}N}{(L/x)\cdot M\underset{wr}{\to}(L/x)\cdot N}\,ExtnR$$

Later, we use the following subrelations:

- $(-)\underset{\sigma}{\to}(-)$: defined by substitution rules and compatibility rules.
- $(-)\underset{\beta}{\to}(-)$: defined by beta rules and compatibility rules. ■

Compatibility rules imply the following proposition.

Proposition 2.6. *Let M, M' be terms and $L[\,]$ a context. If $M\underset{wr}{\to}M'$ then*

$L[M] \underset{wr}{\rightarrow} L[M']$.

Here, contexts is defined as follows:

Definition 2.7 [Contexts–Terms with holes–] A context $L[\ ]$ is a term with a "hole" defined as

$$\begin{aligned} L[\ ] ::= & [\ ] \\ & |(L)[\ ]\ N)|(M\ L[\ ]) \\ & |\lambda x.L[\ ] \\ & |L[\ ]\circ N|M\circ L[\ ] \\ & |(L[\ ]/x)\cdot N|(M/x)\cdot L[\ ]. \end{aligned}$$

■

We would like to show examples of the weak reduction defined above. A term $((\lambda y.\lambda x.id)M)N$ corresponds to the expression of Scheme:

(((lambda (y) (lambda (x) (the-environment))) M)N)

The result of this Scheme expression is an environment in which variable x and y are bound to N and M, respectively. This is reduced as follows:

$$\begin{aligned} & ((\lambda y.\lambda x.id)M)N \\ & \underset{wr}{\rightarrow} ((\lambda x.id)\circ((M/y)\cdot id))N \qquad Beta2 \\ & \underset{wr}{\rightarrow} id\circ((N/x)\cdot(M/y)\cdot id) \qquad Beta1 \\ & \underset{wr}{\rightarrow} (N/x)\cdot(M/y)\cdot id \qquad IdL \end{aligned}$$

The next example is more complicated. A term $(\lambda e.(y\circ e))((\lambda y.\lambda x.id)MN)$, which corresponds to the Scheme expression

((lambda (e) (eval 'y e))
(((lambda (y) (lambda (x) (the-environement))) M) N)),

is reduced as follows:

$$(\lambda e.(y\circ e))((\lambda y.\lambda x.id)MN)$$

$\overset{*}{\underset{wr}{\rightarrow}}(\lambda e.(y \circ e))((N/x)\cdot(M/y)\cdot id)$ *similar to the first example*

$\underset{wr}{\rightarrow}$ $(y \circ e) \circ ((((N/x)\cdot(M/y)\cdot id)/e)\cdot id)$ *Beta2*

$\underset{wr}{\rightarrow}$ $y \circ (e \circ ((((N/x)\cdot(M/y)\cdot id)/e)\cdot id))$ *Ass*

$\underset{wr}{\rightarrow}$ $y \circ ((N/x)\cdot(M/y)\cdot id)$ *Var Ref*

$\underset{wr}{\rightarrow}$ $y \circ ((M/y)\cdot id)$ *VarSkip*

$\underset{wr}{\rightarrow}$ M *VarRef*

§2.3. Typing of $\lambda_{env}^{\rightarrow}$

Definition 2.8 [Type Judgement and Typing Rules]. *Type judgement* $E \vdash M : A$ is a ternary relation among an environment E, a term M and a type A defined inductively by the following *typing rules* (sometimes called *type inference rules*):

$$\frac{}{\{x:A\}E \vdash x:A}Var$$

$$\frac{E \vdash M : A \to B \quad E \vdash N : A}{E \vdash MN : B}App \qquad \frac{\{x:A\}E \vdash M : B}{E \vdash \lambda x.M : A \to B}Lam$$

$$\frac{}{E \vdash id : E}Id \qquad \frac{E \vdash N : E' \quad E' \vdash M : A}{E \vdash M \circ N : A}Comp \qquad \frac{E \vdash M : A \quad E \vdash N : E'}{E \vdash (M/x)\cdot N : \{x:A\}E'}Extn$$

It is supposed that every environment type E occurring in the above rules is syntactically valid: if E is $\{x_1:A_1\}\cdots\{x_n:A_n\}\rho_X$, then $x_1, \cdots, x_n$ are distinct with each other and $\{x_1, \cdots, x_n\} \subseteq X$. ■

In the above definition, it is implicitly supposed that every environment type occurring in rules is valid. For instance, x must not be bound in E' in typing rule *Extn*.

Definition 2.9. For an environment type E, a term M and a type A, when a type judgement $E \vdash M : A$ holds, we say that the term M has type A under an environment type E, or simply M is typed. We call a proof tree which

derives a type judgement using the above typing rules, a *type inference tree.* ■

We would like to show an example of a type inference tree. A Scheme expression **(lambda (x) (lambda (y) (the-environment)))** is a function which accepts two arguments through formal parameters **x** and **y** and returns the environment where these variables **x** and **y** are bound to received arguments. This expression corresponds to a term $\lambda x.\lambda y.id$ of $\lambda_{env}^{\rightarrow}$, which is typed by the following type inference tree:

$$\dfrac{\dfrac{\dfrac{}{\{y:\beta, x:\alpha\}\rho \vdash id : \{y:\beta, x:\alpha\}\rho}\,Id}{\{x:\alpha\}\rho \vdash (\lambda y.id) : \beta \to \{y:\beta, x:\alpha\}\rho}\,Lam}{\rho \vdash (\lambda x.\lambda y.id) : \alpha \to \beta \to \{y:\beta, x:\alpha\}\rho}\,Lam .$$

Theorem 2.10 [*Subject Reduction Property*] *The type of each term is preserved during weak reduction: let M_1 be a term, E an environment type, and A a type such that $E \vdash M_1 : A$. If $M_1 \underset{wr}{\rightarrow} M_2$ for some M_2 then $E \vdash M_2 : A$.*

Proof. We prove this theorem by induction on the structure of a derivation tree of reduction $M_1 \underset{wr}{\rightarrow} M_2$.

Since $M_1 \underset{wr}{\rightarrow} M_2$, it is sufficient to apply one of rules of $\underset{wr}{\rightarrow}$ to M_1.

The case of $M_1 = (L \circ M) \circ N$:

M_1 is reduced to $L \circ (M \circ N) (= M_2)$. For some E' and E'', M_1 is typed as

$$\dfrac{\dfrac{\vdots\Sigma_N \qquad \dfrac{\vdots\Sigma_M \qquad \vdots\Sigma_I}{E' \vdash M : E'' \qquad E'' \vdash L : A}}{E \vdash N : E' \qquad E' \vdash L \circ M : A}}{E \vdash (L \circ M) \circ N : A} .$$

Therefore, M_2 is typed as

$$\dfrac{\dfrac{\vdots\Sigma_N \qquad \vdots\Sigma_M}{E \vdash N : E' \qquad E' \vdash M : E''} \qquad \vdots\Sigma_L}{\dfrac{E \vdash M \circ N : E'' \qquad E'' \vdash L : A}{E \vdash L \circ (M \circ N) : A}} .$$

The case of $M_1 = id \circ M$:

M_1 is reduced to M. Since M_1 is actually typed as

$$\dfrac{\dfrac{\vdots \Sigma_M}{E \vdash M : E'} \qquad \dfrac{}{E' \vdash id : E'}}{E \vdash id \circ M : E'} \ ,$$

M_2 is typed as

$$\begin{matrix} \vdots \Sigma_M \\ E \vdash M : E' \end{matrix} .$$

The case of $M_1 = M \circ id$:

M_1 is reduced to M. Since M_1 is actually typed as

$$\dfrac{\dfrac{}{E \vdash id : E} \qquad \dfrac{\vdots \Sigma_M}{E \vdash M : E'}}{E \vdash M \circ id : E'} \ ,$$

M_2 is typed as

$$\begin{matrix} \vdots \Sigma_M \\ E \vdash M : E' \end{matrix} .$$

The case of $M_1 = ((L/x) \cdot M) \circ N$:

$M_1 \underset{wr}{\rightarrow} ((L \circ N)/x) \cdot (M \circ N)$ $(= M_2)$ and since M_1 is typed as

$$\dfrac{\dfrac{\vdots \Sigma_N}{E \vdash N : E''} \quad \dfrac{\dfrac{\vdots \Sigma_L}{E'' \vdash L : A'} \quad \dfrac{\vdots \Sigma_M}{E'' \vdash M : E'}}{E'' \vdash (L/x) \cdot M : \{x : A'\}E'}}{E \vdash ((L/x) \cdot M) \circ N : \{x : A'\}E'} \ ,$$

M_2 is typed as

$$\dfrac{\dfrac{\vdots \Sigma_N}{E \vdash N : E''} \quad \dfrac{\vdots \Sigma_I}{E'' \vdash L : A'}}{E \vdash L \circ N : A'} \qquad \dfrac{\dfrac{\vdots \Sigma_N}{E \vdash N : E''} \quad \dfrac{\vdots \Sigma_M}{E'' \vdash M : E'}}{E \vdash M \circ N : E'}$$

$$E \vdash ((L \circ N)/x) \cdot (M \circ N) : \{x : A'\}E'$$

The case of $M_1 = x \circ ((M/x) \cdot N)$:

$M_1 \underset{wr}{\rightarrow} M$ $(= M_2)$ and since M_1 is typed as

$$\dfrac{\dfrac{\vdots\Sigma_M \quad \vdots\Sigma_N}{E \vdash M : A \quad E \vdash N : E'} \qquad \dfrac{}{\{x{:}A\}\, E' \vdash x : A}}{E \vdash x \circ ((M/x) \cdot N) : A}\ ,$$

M_2 is typed as

$$\vdots\Sigma_M$$
$$E \vdash M : A\ .$$

The case of $M_1 = y \circ ((M/x) \cdot N)$:

$M_1 \underset{wr}{\rightarrow} y \circ N$ $(= M_2)$ and since M_1 is typed as

$$\dfrac{\dfrac{\vdots\Sigma_M \quad \vdots\Sigma_N}{E \vdash M : B \quad E \vdash N : \{y : A\}E'} \qquad \dfrac{}{\{x : B\}\{y : A\}E' \vdash y : B}}{E \vdash y \circ ((M/x) \cdot N) : A}$$

M_2 is typed as

$$\dfrac{\vdots\Sigma_N \qquad \dfrac{}{\{y : A\}E' \vdash y : A}}{E \vdash y \circ N : A}$$

The case of $M_1 = (LM) \circ N$:

$M_1 \underset{wr}{\rightarrow} (L \circ M)(M \circ N)$ $(= M_2)$ and since M_1 is typed as

$$\dfrac{\dfrac{\vdots\Sigma_N \qquad \dfrac{\vdots\Sigma_L \qquad \vdots\Sigma_M}{E' \vdash L : B \rightarrow A \quad E' \vdash M : B}}{E \vdash N : E' \qquad E' \vdash LM : A}}{E \vdash (LM) \circ N : A}\ ,$$

M_2 is typed as

$$\dfrac{\dfrac{\vdots\Sigma_N \quad \vdots\Sigma_L}{E\vdash N:E' \quad E'\vdash L:B\to A} \qquad \dfrac{\vdots\Sigma_N \quad \vdots\Sigma_M}{E\vdash N:E' \quad E'\vdash M:B}}{\dfrac{E\vdash L\circ N:B\to A \qquad E\vdash M\circ N:B}{E\vdash (L\circ M)\,(M\circ N):A}}\ .$$

The case of $M_1=((\lambda x.M)\circ L)N$:

$M_1 \underset{wr}{\to} M\circ((N/x)\cdot L)$ $(=M_2)$ and since M_1 is typed as

$$\dfrac{\dfrac{\dfrac{\vdots\Sigma_L \qquad \dfrac{\vdots\Sigma_M}{\{x:B\}E'\vdash M:A}}{E\vdash L:E' \quad E'\vdash \lambda x.M:B\to A} \qquad \vdots\Sigma_N}{E\vdash(\lambda x.M)\circ L:B\to A \qquad E\vdash N:B}}{E\vdash((\lambda x.M)\circ L)N:A}$$

M_2 is typed as

$$\dfrac{\dfrac{\vdots\Sigma_N \quad \vdots\Sigma_L}{E\vdash N:B \quad E\vdash L:E'} \qquad \dfrac{\vdots\Sigma_M \quad \vdots\Sigma_M}{\{x:B\}\,E'\vdash M:A}}{E\vdash M\circ((N/x)\cdot L):A}\ .$$

The case of $M_1=(\lambda x.M)N$:

$M_1 \underset{wr}{\to} M\circ((N/x)\cdot id)$ $(=M_2)$ and since M_1 is typed as

$$\dfrac{\dfrac{\dfrac{\vdots\Sigma_M}{\{x:B\}E'\vdash M:A} \qquad \vdots\Sigma_N}{E\vdash\lambda x.M:B\to A \quad E\vdash N:B}}{E\vdash(\lambda x.M)\,N:A}\ ,$$

M_2 is typed as

$$\dfrac{\dfrac{\vdots\Sigma_N \qquad \dfrac{}{E\vdash id:E}}{E'\vdash N:B} \qquad \vdots\Sigma_M}{\dfrac{E\vdash(N/x)\cdot id:\{x:B\}E \quad \{x:B\}E\vdash M:A}{E\vdash M\circ((N/x)\cdot id):A}}\ .$$

The case of $M_1=ML$, $M_2=NL$, and $M_1 \underset{wr}{\to} M_2$ by rule *AppL*:

The proof tree of weak reduction from M_1 to M_2 is as

$$\frac{\vdots}{M \underset{wr}{\to} N}$$
$$(ML) \underset{wr}{\to} (NL).$$

By the induction hypothesis for M, if we suppose M is typed as

$$\vdots\Sigma_M$$
$$E \vdash M : B \to A\ ,$$

then N is also type in the same way:

$$\vdots\Sigma_N$$
$$E \vdash N : B \to A\ .$$

Since M_1 is typed as

$$\frac{\vdots\Sigma_M \qquad \vdots\Sigma_L}{E \vdash M : B \to A \quad E \vdash L : B}$$
$$E \vdash ML : A$$

M_2 is typed as

$$\frac{\vdots\Sigma_N \qquad \vdots\Sigma_L}{E \vdash N : B \to A \quad E \vdash L : B}$$
$$E \vdash ML : A \qquad .$$

The remaining cases, *AppR*, *Lam*, *CompR*, *CompL*, *ExtnR*, and *ExtnL*, are proved similarly to the case of *AppL*. **q.e.d.**

§3 Confluence Property

In this section, we prove confluence property of $\lambda_{env}^{\to}$. We show that the proof of confluence of weak $\lambda\sigma$-calculus in [6] can be used in $\lambda_{env}^{\to}$ with almost no change.

Lemma 3.1 [*Interpretation Method* (*cf. page* 4 *in* [6])].
Let $\mathscr{R} = \mathscr{R}_1 \cup \mathscr{R}_2$ be the union of two (*binary*) *relations* (*on terms*), *$\mathscr{R}_1$ being confluent and strongly normalizing. We denote by $\mathscr{R}_1(M)$ the $\mathscr{R}_1$-normal form of M. Suppose there is some relation $\mathscr{R}'$ on $\mathscr{R}_1$-normal forms satisfying*:

$$\mathscr{R}' \subseteq \mathscr{R}^* \quad \text{and} \quad (M \underset{\mathscr{R}_2}{\to} N \Rightarrow \mathscr{R}_1(M) \overset{*}{\underset{\mathscr{R}'}{\to}} \mathscr{R}_1(N)).$$

Then $\mathscr{R}'$ is confluent if and only if $\mathscr{R}$ is confluent.

Proof. Suppose first that $\mathscr{R}'$ is confluent. Let $M \overset{*}{\underset{\mathscr{R}}{\to}} M'$ and $M \overset{*}{\underset{\mathscr{R}}{\to}} M''$. Then

by assumption, $\mathscr{R}_1(M)\overset{*}{\underset{\mathscr{R}'}{\to}}\mathscr{R}_1(M')$ and $\mathscr{R}_1(M)\overset{*}{\underset{\mathscr{R}'}{\to}}\mathscr{R}_1(M'')$. By confluence of $\mathscr{R}'$, there exists N such that $\mathscr{R}_1(M')\overset{*}{\underset{\mathscr{R}'}{\to}}N$ and $\mathscr{R}_1(M'')\overset{*}{\underset{\mathscr{R}'}{\to}}N$. Since $\mathscr{R}' \subseteq \mathscr{R}^*$, we have $M'\overset{*}{\underset{\mathscr{R}}{\to}}\mathscr{R}_1(M')\overset{*}{\underset{\mathscr{R}}{\to}}N$ and similarly $M''\overset{*}{\underset{\mathscr{R}}{\to}}N$. Hence, $\mathscr{R}$ is confluent.

Suppose conversely that $\mathscr{R}$ is confluent. Let $M\overset{*}{\underset{\mathscr{R}'}{\to}}M'$ and $M\overset{*}{\underset{\mathscr{R}'}{\to}}M''$. By confluence of $\mathscr{R}$, these exists N such that $M'\overset{*}{\underset{\mathscr{R}}{\to}}N$ and $M''\overset{*}{\underset{\mathscr{R}}{\to}}N$. Then, by assumption, we have

$$M'=\mathscr{R}_1(M')\overset{*}{\underset{\mathscr{R}'}{\to}}\mathscr{R}_1(N) \text{ and } M''\overset{*}{\underset{\mathscr{R}'}{\to}}\mathscr{R}_1(N),$$

and the diagram is closed. (This proof is cited from [6].) **q.e.d.**

Later, we will prove the confluence of $\lambda^{\rightarrow}_{env}$ by setting as

$$\mathscr{R}=(-)\underset{wr}{\to}(-),$$

$$\mathscr{R}'=(-)\underset{\beta/\sigma}{\to}(-),$$

$$\mathscr{R}_1=(-)\underset{\sigma}{\to}(-),$$

and $$\mathscr{R}_2=(-)\underset{\beta}{\to}(-).$$

where $\underset{\sigma}{\to}$ and $\underset{\beta}{\to}$ are already defined in Definition 2.5 and $\underset{\beta/\sigma}{\to}$ is as follows:

Definition 3.2 [Relation $(-)\underset{\beta/\sigma}{\to}(-)$].

For σ-normal terms U and V, $U\underset{\beta/\sigma}{\to}V$ if and only if there exists a term N such that $U\underset{\beta}{\to}N$ and V is the σ-normal term of N:

$$\begin{array}{ccccc} U & \underset{\beta}{\to} & N & \overset{*}{\underset{\sigma}{\to}} & V \twoheadrightarrow_{\sigma} \\ \| & & & & \| \\ U & & \underset{\beta/\sigma}{\to} & & V \ . \end{array}$$

■

We conclude the confluence of $\lambda^{\rightarrow}_{env}$ if the following propositions are proved:

- Reduction σ is confluent and strongly normalizing; (Lemma 3.3 and Lemma 3.5)
- If $M \underset{\beta}{\rightarrow} M'$, then $\sigma(M) \overset{*}{\underset{\beta/\sigma}{\rightarrow}} \sigma(N)$; (Lemma 3.6)
- Reduction β/σ is confluent on a set of σ-normal terms. (Lemma 3.15)

First, we show the confluence and the strong normalizability of σ.

Lemma 3.3 [*Termination of* σ]. *Reduction* σ *is terminating.*

Proof. It is easily checked that $length(M) > length(N)$ if $M \underset{\sigma}{\rightarrow} N$. Therefore, $(-)\underset{\sigma}{\rightarrow}(-)$ is terminating. **q.e.d.**

Lemma 3.4 [*Local Confluence of* σ]. *Reduction* σ *is locally confluent.*

Proof. By checking the overlapping cases. **q.e.d.**

By Newman's lemma (confluence = termination and locally confluence),

Lemma 3.5 [*Confluence of* σ]. *Reduction* σ *is confluent.*

Lemma 3.6. *If* $M \underset{\beta}{\rightarrow} N$, *then* $\sigma(M) \overset{*}{\underset{\beta/\sigma}{\rightarrow}} \sigma(N)$.

Proof. We prove this lemma by the induction on the lexical ordering $(depth(M), length(M))$ where $depth(M)$ is the maximum length of σ-normalization's sequences.

$M = x$ or $M = id$: This case does not happen because $M \underset{\beta}{\rightarrow} N$ and x and id cannot be reduced.

$M = \lambda x.M_1$: N has a form $\lambda x.N_1$ such that $M_1 \underset{\beta}{\rightarrow} N_1$ since the last rule for deriving $M \underset{\beta}{\rightarrow} N$ is *Lam*. By the induction hypothesis,

$$\sigma(M_1) \overset{*}{\underset{\beta/\sigma}{\rightarrow}} \sigma(N_1).$$

Therefore,

$$\sigma(M)=\lambda x.\sigma(M_1)\overset{*}{\underset{\beta/\sigma}{\rightarrow}}\lambda x.\sigma(N_1)=\sigma(N).$$

$M=M_1M_2$:

There are several cases according to the last rule used in $M\underset{\beta}{\rightarrow}N$.

[The last rule of $M\underset{\beta}{\rightarrow}N$ is *AppL*]

N has a form (M'_1M_2) such that $M_1\underset{\beta}{\rightarrow}M'_1$. By the induction hypothesis, $\sigma(M_1)\overset{*}{\underset{\beta/\sigma}{\rightarrow}}\sigma(M)'_1$. Therefore,

$$\sigma(M)=\sigma(M_1M_2)=\sigma(M_1)\sigma(M_2)\overset{*}{\underset{\beta/\sigma}{\rightarrow}}\sigma(M'_1)\sigma(M_2)=\sigma(M'_1M_2).$$

[The last rule of $M\underset{\beta}{\rightarrow}N$ is *AppR*]

This case is proved similarly to the above case.

[The last rule of $M\underset{\beta}{\rightarrow}N$ is *Beta*1]

We can assume that

$$M_1=(\lambda x.M_{11})\circ M_{12}$$
$$N=M_{11}\circ((M_2/x)\cdot M_{12}).$$

for some M_{11} and M_{12}. If M_{12} is not *id*, then

$$\sigma(M)=(((\lambda x.\sigma(M_{11}))\circ\sigma(M_{12}))\sigma(M_2)$$
$$\sigma(N)=\sigma(\sigma(M_{11})\circ((\sigma(M_2)/x)\cdot\sigma(M_{12}))).$$

Therefore,

$$\sigma(M)=((\lambda x.\sigma(M_{11}))\circ\sigma(M_{12}))\sigma(M_2)$$
$$\underset{\beta}{\rightarrow}\sigma(M_{11})\circ((\sigma(M_2)/x)\cdot\sigma(M_{12}))$$

$$\xrightarrow[\sigma]{*}\sigma(\sigma(M_{11})\circ((\sigma(M_2)/x)\cdot\sigma(M_{12})))$$

$$=\sigma(N).$$

$\sigma(M)\ \underset{Beta1}{\rightarrow}\ \sigma(M_{11})\circ(\sigma(M_2)/x)\cdot\sigma(M_{12})\ \xrightarrow[\sigma]{*}\ \sigma(\sigma(M_{11})\circ(\sigma(M_{12})/x)\cdot\sigma(L_2))$, that is, $\sigma(M)\underset{\beta/\sigma}{\rightarrow}\sigma(N)$.

If M_{12} is *id*, then

$$\sigma(M)=(\lambda x.\sigma(M_{11}))\sigma(M_2)$$
$$\sigma(N)=\sigma((\sigma(M_{11})\circ((\sigma(M_2)/x)\cdot id)))$$

Hence,

$$\sigma(M)=(\lambda x.\sigma(M_{11}))\sigma(M_2)$$
$$\underset{\beta}{\rightarrow}\sigma(M_{11})\circ((\sigma(M_2)/x)\cdot id))$$
$$\xrightarrow[\sigma]{*}\sigma((\sigma(M_{11})\circ((\sigma(M_2)/x)\cdot\sigma(M_{12}))))$$
$$=\sigma(N)$$

[The last rule of $M\underset{\beta}{\rightarrow}N$ is *Beta*2]

This case is almost same as the one of *Beta*1.

M is a composition:

We distinguish the following subcases:

Subcase 1: $M=(M_1\circ M_2)\circ M_3$
Subcase 2: $M=id\circ M_1$
Subcase 3: $M=M_1\circ id$
Subcase 4: $M=((M_1/x)\cdot M_2)\circ M_3$
Subcase 5: $M=x\circ((M_1/x)\cdot M_2)$
Subcase 6: $M=y\circ((M_1/x)\cdot M_2)$
Subcase 7: $M=(M_1M_2)\circ N$
Subcase 8: *Otherwise*

We here prove only **Subcase** 1 and 7 because the other cases are similar or simpler.

[Subcase 1]: A β-redex exists in M_1, M_2 or M_3. Here, we consider the case that the redex occurs in M_2 or M_3. M_1, that is, $M_1 \underset{\beta}{\rightarrow} M'_1$ and $N = (M'_1 \circ M_2) \circ M_3$. (The cases of M_2 or M_3 are almost similar, therefore, we prove only this case.)

We can apply the induction hypothesis to $M_1 \circ (M_2 \circ M_3) \underset{\beta}{\rightarrow} M'_1 \circ (M_2 \circ M_3)$, since $depth(M_1 \circ (M_2 \circ M_3)) < depth((M_1 \circ M_2) \circ M_3)$. Then, we know that

$$\sigma(M_1 \circ (M_2 \circ M_3)) \underset{\beta/\sigma}{\overset{*}{\rightarrow}} \sigma(M'_1 \circ (M_2 \circ M_3)).$$

Hence,

$$\begin{aligned}
\sigma(M) &= \sigma((M_1 \circ M_2) \circ M_3) \\
&= \sigma(M_1 \circ (M_2 \circ M_3)) \\
&\underset{\beta/\sigma}{\overset{*}{\rightarrow}} \sigma(M'_1 \circ (M_2 \circ M_3)) \\
&= \sigma((M'_1 \circ M_2) \circ M_3) \\
&= \sigma(N).
\end{aligned}$$

[Subcase 7]: Here, we may suppose that the β-redex is in M_3. The other cases are similar and simpler. Suppose that $N = (M_1 M_2) \circ M'_3$ for some M'_3 such that $M_3 \underset{\beta}{\rightarrow} M'_3$.

By the induction hypothesis,

$$\sigma(M_1 \circ M_3) \underset{\beta/\sigma}{\overset{*}{\rightarrow}} \sigma(M_1 \circ M'_3) \text{ and } \sigma(M_2 \circ M_3) \underset{\beta/\sigma}{\overset{*}{\rightarrow}} \sigma(M_2 \circ M'_3).$$

Therefore,

$$\begin{aligned}
\sigma(M) &= \sigma((M_1 M_2) \circ M_3) \\
&= \sigma((M_1 \circ M_3)(M_2 \circ M_3)) \\
&= \sigma((M_1 \circ M_3))\sigma((M_2 \circ M_3)) \\
&\underset{\beta/\sigma}{\overset{*}{\rightarrow}} \sigma((M_1 \circ M'_3))\sigma((M_2 \circ M_3)) \\
&\underset{\beta/\sigma}{\overset{*}{\rightarrow}} \sigma((M_1 \circ M'_3))\sigma((M_2 \circ M'_3)) \\
&= \sigma((M_1 \circ M'_3))\sigma((M_2 \circ M'_3))
\end{aligned}$$

$$= \sigma((M_1 \circ M'_3)(M_2 \circ M'_3))$$
$$= \sigma((M_1 M_2) \circ M_3)$$
$$= \sigma(N).$$

$M = (M_1/x) \cdot M_2$:

[The case that the last rule of $M \xrightarrow{}_{\beta} N$ is *ExtnL*]

It holds that $N = (M'_1/x) \cdot M_2$ where $M_1 \xrightarrow{}_{\beta} M'_1$. By the induction hypothesis, $\sigma(M_1) \xrightarrow[\beta/\sigma]{} \sigma(M'_1)$. Hence,

$$\sigma(M) = (\sigma(M_1)/x) \cdot \sigma(M_2)$$
$$\xrightarrow[\beta/\sigma]{} (\sigma(M'_1)/x) \cdot \sigma(M_2)$$
$$\xrightarrow[\sigma]{*} \sigma((\sigma(M'_1)/x) \cdot \sigma(M_2))$$
$$= \sigma((M_1/x) \cdot M_2)$$
$$= \sigma(N).$$

Therefore, $\sigma(M) \xrightarrow[\beta/\sigma]{} \sigma(N)$.

[The case that the last rule of $M \xrightarrow{}_{\beta} N$ is *ExtnR*]

This case is similar to the above case of *ExtnL*. **q.e.d.**

Lemma 3.7 [*Syntactical Characterization of σ-normal term*].
The set of σ-normal terms are generated by the following grammar:

$$U ::= id \mid (U_1/x) \cdot U_2 \mid x \mid (U_1 U_2) \mid \lambda x.U \mid (\lambda x.U_1) \circ U_2 \mid x \circ W$$

where U, U_1, U_2, V, W are σ-normal and U_2 is not id and W is not an environment extension.

We next introduce parallel reduction for proving confluence of $(-) \xrightarrow[\beta/\sigma]{} (-)$:

Definition 3.8 [Parallel Reduction on σ-normal Terms].
We define a binary relation $(-) \Rightarrow_{par} (-)$, called *parallel* (*weak*) *reduction*, on terms inductively as follows:

$$\frac{}{x \Rightarrow_{par} x}\ ParVar$$

$$\frac{U \underset{par}{\Rightarrow} U'}{\lambda x.U \underset{\lambda x.}{\Rightarrow} \lambda x.U'}\ ParLam \qquad \frac{U \underset{pal}{\Rightarrow} U' \quad V \underset{pal}{\Rightarrow} V'}{(UV) \underset{pal}{\Rightarrow} (U'\,V')}\ ParApp$$

$$\frac{}{id \underset{pal}{\Rightarrow} id}\ ParId \qquad \frac{U \underset{pal}{\Rightarrow} U' \quad V \underset{pal}{\Rightarrow} V'}{(U/x) \cdot V \underset{pal}{\Rightarrow} (U'/x) \cdot V'}\ ParExtn$$

$$\frac{U \underset{pal}{\Rightarrow} U' \quad V \underset{pal}{\Rightarrow} V' \quad V \neq id}{(\lambda x.U) \circ V \underset{par}{\Rightarrow} (\lambda x.U') \circ V'}\ ParLamComp$$

$$\frac{W \underset{pal}{\Rightarrow} W' \quad W \text{ is not an environment extension}}{x \circ W \underset{pal}{\Rightarrow} x \circ W'}\ ParVarComp$$

$$\frac{U \underset{pal}{\Rightarrow} U' \quad V \underset{pal}{\Rightarrow} V'}{(\lambda x.U)V \underset{pal}{\Rightarrow} \sigma\,(U' \circ ((V'/x) \cdot id))}\ ParBeta1$$

$$\frac{W \neq id \quad U \underset{pal}{\Rightarrow} U' \quad V \underset{pal}{\Rightarrow} V' \quad W \underset{pal}{\Rightarrow} W'}{((\lambda x.U) \circ W)V \underset{pal}{\Rightarrow} \sigma\,(U' \circ ((V'/x) \cdot W'))}\ ParBeta2$$

The following lemma is proved by induction on the length of a term:

Lemma 3.9. *If $U \underset{par}{\Rightarrow} V$, then U and V are σ-normal terms.*

Lemma 3.10 [*Reflexivity of Parallel Reduction*] $(-)\underset{par}{\Rightarrow}(-)$ *is reflexive, that is, $U \underset{par}{\Rightarrow} U$ for every σ-normal term U.*

The following lemma is used in Lemma 3.13

Lemma3.11. *If $U \underset{par}{\Rightarrow} U'$ and $V \underset{par}{\Rightarrow} V'$ then $\sigma(U \circ V) \underset{par}{\Rightarrow} \sigma(U' \circ V')$.*

Proof. We prove this lemma by induction on *length*$(U \circ V)$. Suppose that $U \underset{par}{\Rightarrow} U'$ and $V \underset{par}{\Rightarrow} V'$. (Note that this supposition implies that U, U', V, *and* V' are σ-normal.)

The case of $U = id$: $\sigma(U \circ V) = \sigma(V) = V$. U' is *id* since *id* is reduced uniquely to *id*. Thus,

$$\sigma(U' \circ V') = \sigma(id \circ V') = \sigma(V') = V'$$

By the assumption, $V \underset{par}{\Rightarrow} V'$. Therefore, $\sigma(U \circ V) \underset{par}{\Rightarrow} \sigma(U' \circ V')$.

The case of $U = (U_1/x) \cdot U_2$: it holds that $U' = (U_1'/x) \cdot U_2'$ for some U_1 and U_2 such that $U_1 \underset{par}{\Rightarrow} U_1'$ and $U_2 \underset{par}{\Rightarrow} U_2'$ since $U \underset{par}{\Rightarrow} U'$. And,

$$\sigma(U \circ V) = \sigma(((U_1/x) \cdot U_2) \circ V) = (\sigma(U_1 \circ V)/x) \cdot \sigma(U_2 \circ V).$$

By the induction hypothesis,

$$\sigma(U_1 \circ V) \underset{par}{\Rightarrow} \sigma(U_1' \circ V') \text{ and } \sigma(U_2 \circ V) \underset{par}{\Rightarrow} \sigma(U_2' \circ V').$$

Therefore, $(\sigma(U_1 \circ V)/x) \cdot \sigma(U_2 \circ V) \underset{par}{\Rightarrow} (\sigma(U_1' \circ V')/x) \cdot \sigma(U_2' \circ V')$, that is, $\sigma(U \circ V) \underset{par}{\Rightarrow} \sigma(U' \circ V')$.

The case of $U = x$ and $V = (V_1/x) \cdot V_2$: it holds that $V' = (V_1'/x) \cdot V_2'$ for some V_1' and V_2' such that $V_1 \underset{par}{\Rightarrow} V_1'$ and $V_2 \underset{par}{\Rightarrow} V_2'$, since $V \underset{par}{\Rightarrow} V'$. Therefore,

$$\begin{aligned}
\sigma(U \circ V) &= \sigma(x \circ ((V_1/x) \cdot V_2)) \\
&= \sigma(V_1) \\
&= V_1 \text{ (since } V_1 \text{ is } \sigma\text{-normal)} \\
&\underset{par}{\Rightarrow} V_1' \text{ (by the assumption)} \\
&= \sigma(V_1') \text{ (since } V_1' \text{ is } \sigma\text{-normal)} \\
&= \sigma(x \circ ((V_1'/x) \cdot V_2')) \\
&= \sigma(U' \circ V') \text{ (since } x(=U) \text{ is uniquely } \sigma\text{-reduced to } x(=U')).
\end{aligned}$$

The case that $U = x$ and $V = (V_1/y) \cdot V_2 (x \neq y)$.
It holds that $V' = (V_1'/x) \cdot V_2'$ for some V_1' and V_2' such that $V_1 \underset{par}{\Rightarrow} V_1'$ and $V_2 \underset{par}{\Rightarrow} V_2'$, since $V \underset{par}{\Rightarrow} V'$.

$$\begin{aligned}
\sigma(U \circ V) &= \sigma(x \circ ((V_1/y) \cdot V_2)) \\
&= \sigma(x \circ V_2) \\
&\underset{par}{\Rightarrow} \sigma(x \circ V_2') \text{ (by the induction hypothesis)}
\end{aligned}$$

$$=\sigma(x\circ((V_1'/y)\cdot V_2'))$$
$$=\sigma(U'\circ V')$$

The case that $U=x$ and V is not an environment extension:

$$\sigma(U\circ V)=\sigma(x\circ V)$$
$$=\sigma(x)\circ\sigma(V) \text{ (since } V \text{ is not an environment extension)}$$
$$=x\circ V$$
$$\underset{par}{\Rightarrow} x\circ V' \text{ (by the assumption } V\underset{par}{\Rightarrow}V')$$
$$=\sigma(x)\circ\sigma(V')$$
$$=\sigma(x\circ V')$$
$$=\sigma(U'\circ V').$$

The case that U is an application:

We analyze the cases of the last rule for deriving $U\underset{par}{\Rightarrow}U'$:

(Case 1: *ParApp*) $U=U_1U_2$ for some U_1 and U_2 such that $U_1\underset{par}{\Rightarrow}U_1'$ and $U_2\underset{par}{\Rightarrow}U_2'$;

(Case 2: *ParBeta*1) $U=(\lambda x.U_1)U_2$ and $U'=\sigma(U_1'\circ((U_2'/x)\cdot id))$ such that $U_1\underset{par}{\Rightarrow}U_1'$ and $U_2\underset{par}{\Rightarrow}U_2'$;

(Case 3: *ParBeta*2) $U=((\lambda x.U_1)\circ U_2)U_3$ and $U'=\sigma(U_1'\circ((U_3'/x)\cdot U_2'))$ such that $U_1\underset{par}{\Rightarrow}U_1'$, $U_2\underset{par}{\Rightarrow}U_2'$, and $U_3\underset{par}{\Rightarrow}U_3'$.

[Case 1]:

$$\sigma(U\circ V)=\sigma((U_1U_2)\circ V)$$
$$=\sigma((U_1\circ V)(U_2\circ V)$$
$$=\sigma((U_1\circ V)\sigma(U_2\circ V).$$

By the induction hypothesis, $\sigma(U_1\circ V)\underset{par}{\Rightarrow}\sigma(U_1'\circ V')$ and $\sigma(U_2\circ V)\underset{par}{\Rightarrow}\sigma(U_2'\circ V')$. Therefore,

$$\sigma(U\circ V)\underset{par}{\Rightarrow}\sigma(U_1'\circ V')\sigma(U_2'\circ V')$$
$$=\sigma((U_1'\circ U_2')\circ V')$$
$$=\sigma(U'\circ V')$$

[Case 2]:

$$\sigma(U\circ V)=\sigma(((\lambda x.U_1)U_2)\circ V)$$
$$=\sigma((\lambda x.U_1)\circ V)\sigma(U_2\circ V)$$

By the induction hypothesis, $\sigma((\lambda x.U_1)\circ V)\underset{par}{\Rightarrow}\sigma((\lambda x.U_1')\circ V')$ and $\sigma(U_2\circ V)\underset{par}{\Rightarrow}\sigma(U_2'\circ V')$. Therefore,

$$\sigma(U\circ V)\underset{par}{\Rightarrow}\sigma((\lambda x.U_1')\circ V')\sigma(U_2'\circ V')$$
$$=\cdots$$
$$=\sigma(U'\circ V')$$

[Case 3]:

$$\sigma(U\circ V)=\sigma((((\lambda x.U_1)\circ U_2)U_3)\circ V)$$
$$=\sigma(((\lambda x.U_1)\circ(U_2\circ V))(U_3\circ V))$$
$$=\sigma((\lambda x.U_1)\circ(U_2\circ V))\sigma(U_3\circ V)$$
$$=((\lambda x.\sigma(U_1))\circ\sigma(U_2\circ V))\sigma(U_3\circ V).$$

$\sigma(U_1)=U_1\underset{par}{\Rightarrow}U_1'=\sigma(U_1')$. By the induction hypothesis, $\sigma(U_2\circ V)\underset{par}{\Rightarrow}\sigma(U_2'\circ V')$ and $\sigma(U_3\circ V)\underset{par}{\Rightarrow}\sigma(U_3'\circ V')$ Thus,

$$\sigma(U\circ V)\underset{par}{\Rightarrow}\sigma(\sigma(U_1')\circ((\sigma(U_3'\circ V')/x)\cdot\sigma(U_2'\circ V'))).$$

On the other hands,

$$\sigma(U'\circ V')=\sigma((U_1'\circ((U_3'/x)\cdot U_2'))\circ V')$$
$$=\sigma(U_1'\circ((U_3'\circ V'/x)\cdot(U_2'\circ V')))$$
$$=\sigma(\sigma(U_1')\circ((\sigma(U_3'\circ V')/x)\cdot\sigma(U_2'\circ V'))).$$

Therefore, $\sigma(U\circ V)\underset{par}{\Rightarrow}\sigma(U'\circ V')$. (We finished the case that $U=U_1U_2$).

$U=\lambda x.U_1$: If $V=id$ then this is evident by the induction hypothesis $\sigma(U)\underset{par}{\Rightarrow}\sigma(U')$. Thus, we suppose that $V\neq id$.

$$\begin{aligned}\sigma(U\circ V)&=\sigma(U)\circ\sigma(V)\\&=U\circ V \text{ (since } U \text{ and } V \text{ are } \sigma\text{-normal)}\\&=U'\circ V' \text{ (by the assumption)}\\&=\sigma(U')\circ\sigma(V')\\&=\sigma(U'\circ V').\end{aligned}$$

$U=(\lambda x.U_1)\circ U_2$ (where U_2 is not id):
$U'=(\lambda x.U_1')\circ U_2'$ (for some U_1' and U_2' such that $U_1\underset{par}{\Rightarrow}U_1'$ and $U_2\underset{par}{\Rightarrow}U_2'$ because $U\underset{par}{\Rightarrow}U'$.

$$\begin{aligned}\sigma(U\circ V)&=\sigma((\lambda x.U_1)\circ(U_2\circ V))\\&=(\lambda x.\sigma(U_1))\circ\sigma(U_2\circ V)\\&=(\lambda x.U_1)\circ\sigma(U_2\circ V) \text{ (since } U_1 \text{ and } U_2 \text{ are } \sigma\text{-normal)}\\&\underset{par}{\Rightarrow}(\lambda x.U_1')\circ\sigma(U_2'\circ V) \text{ (since } U_1\underset{par}{\Rightarrow}U_1' \text{ and } \sigma(U_2\circ V)\underset{par}{\Rightarrow}\sigma(U_2'\circ V'))\\&=((\lambda x.\sigma(U_1'))\circ\sigma(U_2'\circ V')\\&=\sigma((\lambda x.U_1')\circ(U_2'\circ V'))\\&=\sigma(U'\circ V').\end{aligned}$$

$U=x\circ W$ where W is not an environment extension:
It holds that $U'=x\circ W'$ for some W' such that $W\underset{par}{\Rightarrow}W'$. Then,

$$\begin{aligned}\sigma(U\circ V)&=\sigma((x\circ W)\circ V)\\&=\sigma(x\circ\sigma(W\circ V))\\&\underset{par}{\Rightarrow}\sigma(x\circ\sigma(W'\circ V'))\\&=\sigma((x\circ W')\circ V')\\&=\sigma(U'\circ V').\end{aligned}$$

q.e.d.

Definition 3.12 [Transformation U^*]. We define a transformation $(-)^*$ from σ-normal terms to σ-normal terms inductively as:

$$
\begin{aligned}
id^* &= id \\
((U_1/x)\cdot U_2)^* &= (U_1^*/x)\cdot U_2^* \\
x^* &= x \\
(xU)^* &= xU^* \\
((\lambda x.\, U_1)U_2)^* &= \sigma(U_1^* \circ ((U_2^*/x)\cdot id)) \\
(((\lambda x.\, U_1)\circ U_2)U_3)^* &= \sigma(U_1^* \circ ((U_3^*/x)\cdot U_2^*)) \\
(\lambda x.\, U)^* &= \lambda x.\, U^* \\
((\lambda x.\, U)\circ V)^* &= (\lambda x.\, U^*)\circ V^* \\
(x\circ W)^* &= x\circ W^*,
\end{aligned}
$$

where U, U_1, U_2, V, and W *are* σ-normal and V is not id and W is not an environment extension. ■

U^* of a σ-normal term U is a term where all beta-redexes in U are reduced but the newborn redexes are left.

Lemma 3.13. *If* $U \underset{par}{\Rightarrow} V$ *then* $V \underset{par}{\Rightarrow} U^*$. *Therefore,* $(-)\underset{par}{\Rightarrow}(-)$ *is strongly confluent.*

Proof. By the structural induction on derivation tree of $U \underset{par}{\Rightarrow} V$. We would like to show only two cases of $U=((\lambda x.\, U_1)\circ U_2)U_3$ and $U=(U_1/x)\cdot U_2$ since the other cases are similar to them.

The case of $U=(U_1/x)\cdot U_2$

The derivation tree of $U \underset{par}{\Rightarrow} V$ has a form

$$
\frac{\begin{array}{cc} \vdots & \vdots \\ U_1 \underset{par}{\Rightarrow} V_1 & U_2 \underset{par}{\Rightarrow} V_2 \end{array}}{(U_1/x)\cdot U_2 \underset{par}{\Rightarrow} (V_1/x)\cdot V_2.}
$$

By the induction hypothesis, we know that

$$
V_1 \underset{par}{\Rightarrow} U_1^* \text{ and } V_2 \underset{par}{\Rightarrow} U_2^*.
$$

Thus, $(V_1/x)\cdot V_2 \underset{par}{\Rightarrow} (U_1^*/x)\cdot U_2^*$ holds, that is, $V \underset{par}{\Rightarrow} U^*$

The case of $U=((\lambda x.U_1)\circ U_2)U_3$:

The derivation tree of $U \underset{par}{\Rightarrow} V$ has a form

$$\frac{\overset{\vdots}{U_1 \underset{par}{\Rightarrow} V_1} \quad \overset{\vdots}{U_2 \underset{par}{\Rightarrow} V_2} \quad \overset{\vdots}{U_3 \underset{par}{\Rightarrow} V_3}.}{((\lambda x.U_1)\circ U_2)U_3 \underset{par}{\Rightarrow} (\sigma(V_1\circ(V_3/x)\cdot V_2))}$$

By the induction hypothesis, we know that

$$V_1 \underset{par}{\Rightarrow} U_1^*, \quad V_2 \underset{par}{\Rightarrow} U_2^*, \text{ and } V_3 \underset{par}{\Rightarrow} U_3^*.$$

Thus, $(V_3/x)\cdot V_2 \underset{par}{\Rightarrow} (U_3^*/x)\cdot U_2^*$. By **Lemma** 3.11,

$$\sigma(V_1\circ(V_3/x)\cdot V_2) \underset{par}{\Rightarrow} \sigma(U_1^*\circ((U_3^*/x)\cdot U_2^*)),$$

that is, $V \underset{par}{\Rightarrow} U^*$ by the definition of $(-)^*$. **q.e.d.**

Lemma 3.14. *$\underset{\beta/\sigma}{\rightarrow} \subseteq \underset{par}{\Rightarrow} \subseteq \overset{*}{\underset{\beta/\sigma}{\rightarrow}}$ holds.*

Proof. $\underset{\beta/\sigma}{\rightarrow} \subseteq \underset{par}{\Rightarrow}$, i.e. if $M \underset{\beta/\sigma}{\rightarrow} M'$ then $M \underset{par}{\Rightarrow} M'$ for any M and M':
This is straight-forwardly proved by induction on *length*(M).
$\underset{par}{\Rightarrow} \subseteq \overset{*}{\underset{\beta/\sigma}{\rightarrow}}$, i.e. if $M \underset{par}{\Rightarrow} M'$ then $M \overset{*}{\underset{\beta/\sigma}{\rightarrow}} M'$ for any M and M':
This is also straight-forwardly proved by induction on *length*(M). **q.e.d.**

Lemma 3.15. *$\underset{\beta/\sigma}{\rightarrow}$ is confluent.*

Proof. Let

$$\begin{array}{ccccc} M & \underset{\beta/\sigma}{\rightarrow} & \cdots & \underset{\beta/\sigma}{\rightarrow} & M_1 \\ \downarrow & & & & \\ \vdots & & & & \\ \downarrow & & & & \\ M_2 & & & & \end{array}$$

By Lemma 3.14 and Lemma 3.13,

$$\begin{array}{ccccc} M & \underset{par}{\Rightarrow} & \cdots & \underset{par}{\Rightarrow} & M_1 \\ \vdots & & & & \vdots \\ M_2 & \underset{par}{\Rightarrow} & \cdots & \underset{par}{\Rightarrow} & M'. \end{array}$$

By Lemma 3.14,

$$\begin{array}{ccccc} M & \overset{*}{\underset{\beta/\sigma}{\rightarrow}} & \cdots & \overset{*}{\underset{\beta/\sigma}{\rightarrow}} & M_1 \\ {\scriptstyle *}\downarrow & & & & {\scriptstyle *}\downarrow \\ \vdots & & & & \vdots \\ {\scriptstyle *}\downarrow & & & & {\scriptstyle *}\downarrow \\ M_2 & \overset{*}{\underset{\beta/\sigma}{\rightarrow}} & \cdots & \overset{*}{\underset{\beta/\sigma}{\rightarrow}} & M'. \end{array}$$

Therefore, $M_1 \overset{*}{\underset{\beta/\sigma}{\rightarrow}} M'$ and $M_2 \overset{*}{\underset{\beta/\sigma}{\rightarrow}} M'$. **q.e.d.**

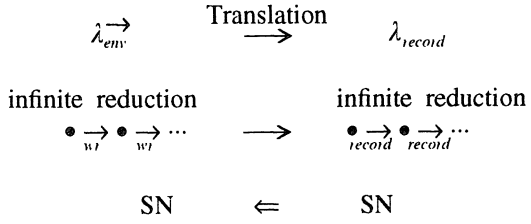
By Lemma 3.1, Lemma 3.3, Lemma 3.5, Lemma 3.6, and Lemma 3.15, we obtain the following theorem:

Theorem 3.16. $\underset{wr}{\rightarrow}$ *is confluent.*

§4. Strong Normalizability

In this section, we would like to show the strong normalizability of $\lambda^{\rightarrow}_{env}$. We will not prove this property directly for $\lambda^{\rightarrow}_{env}$: First, we prove strong normalizability of a (Church-style) simply typed lambda calculus λ_{record} with *records*. Second, we give a translation from type inference trees of $\lambda^{\rightarrow}_{env}$ to

(explicitly) typed terms of λ_{record}. Then, we know that $\lambda_{env}^{\rightarrow}$ has no infinite reduction sequence since neither has λ_{record}.



Let us begin with the definition of simply typed lambda calculus λ_{record} with records.

§4.1. Simple Record Calculus λ_{record}

Definition 4.1 [Type of λ_{record}]. Given a countable set **Type Var** of type variables, *types* of λ_{record} is defined inductively as

$$A ::= \alpha \mid A \to B \mid \{l_1 : A_1, \cdots, l_n : A_n\}$$

where $n \geq 0$.

Type Var may differ from the one of $\lambda_{env}^{\rightarrow}$, however, we assume that **Type Var** is same as the one of $\lambda_{env}^{\rightarrow}$ since it will be more convenient when we interpret λ_{record} to $\lambda_{env}^{\rightarrow}$. ■

Definition 4.2 [Raw Terms of λ_{record}]. Given a countable set **Term Var**$_{\text{record}}$ of term variables and a countable set **Label** of *labels*, we define *terms* of λ_{record} inductively as

$$M ::= x^A \mid \lambda x : A.M \mid MN \mid \langle\rangle \mid \langle l = M \mid N \rangle \mid M.l \mid M \backslash l,$$

where $x \in$ **Term Var**$_{\text{record}}$ and $l \in$ **Label.**

In the sequel, we do *not* interpret each variable in $\lambda_{env}^{\rightarrow}$ as a variable in λ_{record}, but a *label.* Therefore, we assume that **Term Var** of $\lambda_{env}^{\rightarrow}$ and **Term Var**$_{\text{record}}$ are disjoint and suppose that the following injection is given in order to interpret variables as labels:

$$l_{[\![-]\!]} : \textbf{Term Var} \text{ (of } \lambda_{env}^{\rightarrow}) \subsetneq \textbf{Label.}$$

We assume that a superfix A is unique for each variable x^A. When there is no danger of confusion, we will omit variable's superfixes. ■

Definition 4.3 [Typing Rules of λ_{record}]. A sequence $x_1:A_1\cdots x_n:A_n$ is called a *type assignment* when $n\geq 0$, each x_i is a variable of λ_{record}, and each A_i a type of λ_{record}. We use Γ, $\Gamma'\cdots$ as metavariables on type assignments.

Type judgement of λ_{record} is a ternary relation whose domain are type assignment, terms, and types, defined inductively by the following *typing rules*. This relation is written $\Gamma\vdash M:A$ for a type assignment Γ, a term M, and a type A.

$$\frac{0\leq i\leq n}{x_0:A_0\cdots x_n:A_n\vdash x_i^{A_i}:A_i}Var;$$

$$\frac{x:A\ \Gamma\vdash M:B}{\Gamma\vdash\lambda x:A.M:A\to B}Lam; \qquad \frac{\Gamma\vdash M:A\to B\ \Gamma\vdash N:A}{\Gamma\vdash MN:B}App;$$

$$\frac{}{\Gamma\vdash\langle\rangle:\{\}}Nil;$$

$$\frac{\Gamma\vdash M:A\ \Gamma\vdash N:\{l_1:A_1,\cdots,l_n:A_n\}}{\Gamma\vdash\langle l=M|N\rangle:\{l:A,l_1:A_1,\cdots,l_n:A_n\}}Extn;$$

$$\frac{\Gamma\vdash M:\{l_1:A_1,\cdots,l_n:A_n\}}{\Gamma\vdash M.l_i:A_i}RefHit;$$

$$\frac{\Gamma\vdash M:\{l_1:A_1,\cdots,l_n:A_n\}}{\Gamma\vdash M\backslash l_i:\{l_1:A_1,\overset{i}{\cdots},l_n:A_n\}}Rest.$$

■

Notation 4.4 [$\overset{i}{\cdots}$] We abbreviate a record type $\{l_1:A_1,\cdots,l_{i-1}:A_{i-1},l_{i+1}:A_{i+1},\cdots,l_n:A_n\}$ to $\{l_1:A_1,\overset{i}{\cdots},l_n:A_n\}$. ■

Definition 4.5 [Reduction of λ_{record}]. Reduction $(-)_{\overrightarrow{record}}(-)$ is defined inductively as follows:

$$\frac{}{(\lambda x{:}A.M)N \underset{record}{\rightarrow} [N/x]M}\ Beta$$

$$\frac{}{\langle l = M \mid N\rangle.l \underset{record}{\rightarrow} M}\ RefHit \qquad \frac{l \neq l'}{\langle l' = M \mid N\rangle.l \underset{record}{\rightarrow} N.l}\ RefSkip$$

$$\frac{}{\langle l = M \mid N\rangle \backslash l \underset{record}{\rightarrow} N}\ RestHit \qquad \frac{l \neq l'}{\langle l' = M \mid N\rangle \backslash l \underset{record}{\rightarrow} N \backslash l}\ RestSkip$$

$$\frac{M \underset{record}{\rightarrow} M'}{M\,N \underset{record}{\rightarrow} M'N}\ AppL \qquad \frac{N \underset{record}{\rightarrow} N'}{M\,N \underset{record}{\rightarrow} M\,N'}\ AppR$$

$$\frac{M \underset{record}{\rightarrow} M'}{\lambda x : A.M \underset{record}{\rightarrow} \lambda x : A.M'}\ Lam$$

$$\frac{M \underset{record}{\rightarrow} M'}{\langle l = M \mid N\rangle \underset{record}{\rightarrow} \langle l = M' \mid N\rangle}\ ExtnL \qquad \frac{N \underset{record}{\rightarrow} N'}{\langle l = M \mid N\rangle \underset{record}{\rightarrow} \langle l = M \mid N'\rangle}\ ExtnR$$

$$\frac{M \underset{record}{\rightarrow} M'}{M.l \underset{record}{\rightarrow} M'.l}\ RefHit \qquad \frac{M \underset{record}{\rightarrow} M'}{M \backslash l \underset{record}{\rightarrow} M' \backslash l}\ Rest.$$

This reduction is extended by adding rules for records, i.e. *RefHit* and *Rest*. ■

We can derive easily the following proposition from the compatibility rules:

Proposition 4.6. *Let M,M' be terms and $L[\ ]$ a context. If $M \underset{record}{\rightarrow} M'$ then $L[M] \underset{record}{\rightarrow} L[M']$. As a corollary, subterms of any strongly normalizable term are strongly normalizable.*

Proposition 4.7 [*Subject Reduction Property of λ_{record}*]. *$(-) \underset{record}{\rightarrow} (-)$ is subject reduction, that is, for every term M typed as $\Gamma \vdash M : A$, if $M \underset{record}{\rightarrow} M'$ then it holds that $\Gamma \vdash M' : A$.*

Proof. The subject reduction property of λ_{record} is proved in the same way as $\lambda_{env}^{\rightarrow}$. **q.e.d.**

We will show the strong normalizability of λ_{record}. This calculus seems to be a variant of the simply typed lambda calculus with (non-surjective) pairing. The difference between λ_{record} and the calculus with pairing is the method of reference to fields: the former is by *name* and the latter by

integer. The reader who thinks that the normalizability obviously holds, may skip the following definitions and proofs.

We follow the outline of Girard's proof in [7] from page 42 to 46.

Definition 4.8 [Reducible terms $\mathbf{Red}(A)$]. For each type A, we define a set $\mathbf{Red}(A)$ of *reducible* terms of type A by induction of type A:

$$\mathbf{Red}(\alpha)=\mathbf{SN}(\alpha);$$
$$\mathbf{Red}(A\to B)=\{M\in\mathbf{Term}(A\to B)\mid\forall N\in\mathbf{Red}(A).(MN)\in\mathbf{Red}(B)\};$$
$$\mathbf{Red}(\{\})=\mathbf{SN}(\{\})$$
$$\mathbf{Red}(\{l_1:A_1,\cdots,l_n:A_n\})=\{M\in\mathbf{Term}(\{l_1:A_1,\cdots,l_n:A_n\})\mid M.l_i\in\mathbf{Red}(A_i)$$
$$\text{and } M\backslash l_i\in\mathbf{Red}(\{l_1:A_1,\overset{i}{\cdots},l_n:A_n\}) \text{ for } i=1,\cdots,n\}$$

■

Definition 4.9 [Neutral Term]. A term is called *neutral* if and only if it is of the form x, MN, $\langle\rangle$, $M.l$, or $M\backslash l$. ■

Definition 4.10 [Measure ν]. For each normalizable term M, $\nu(M)$ is the maximal length of normalization sequence of M. ■

Lemma 4.11. *M is strongly normalizable if and only if $\nu(M)<\infty$.*

Proof. $\Leftarrow$) Immediate.
$\Rightarrow$) By König's lemma. **q.e.d.**

Lemma 4.12. **(CR1)** *If $M\in\mathbf{Red}(A)$ then $M\in\mathbf{SN}(A)$.*

(CR2) *If $M\in\mathbf{Red}(A)$ and $M\xrightarrow[record]{}N$, then $N\in\mathbf{Red}(A)$.*

(CR3) *If M is neutral and it holds that $N\in\mathbf{Red}(A)$ whenever $M\xrightarrow[record]{}N$, then $M\in\mathbf{Red}(A)$.*

As a special case of the last clause:

(CR4) *If M is neutral and normal then $M\in\mathbf{Red}(A)$.*

Proof. We prove this lemma by induction on *length*(A).
$A=\alpha$:

(CR1)(CR2): Evident by the definition of **Red**(α).
(CR3): The same way as usual: we prove by induction on $\nu(M)$ that all terms M' such that $M_{\overrightarrow{record}}M'$ and $M' \neq N$, are strongly normalizable:

$$\begin{array}{ccc} M & \to & N \\ \downarrow & & \downarrow \\ M' & \to & \bullet. \end{array}$$

We can use the induction hypothesis on $M'_{\overrightarrow{record}}\bullet$.

$A=\{\}$:
(CR1) (CR2) (CR3): Similar to the above case.

$A=\{l_1:A_1,\cdots,l_n:A_n\}$ $(n\geq 1)$:
(CR1): Let $M\in$**Red**$(\{l_1:A_1,\cdots,l_n:A_n\})$. Then, $M.l_1\in$**Red**(A_1). By the induction hypothesis on A_1, $M.l_1\in$**SN**(A_1). Hence, $M\in$**SN**$(\{l_1:A_1,\cdots,l_n:A_n\})$ by Lemma 4.6.
(CR2): Let $M\in$**Red**$(\{l_1:A_1,\cdots,l_n:A_n\})$ and $M_{\overrightarrow{record}}N$. Then, for any i, $M.l_i\in$ **Red**(A_i) and $M\backslash l_i\in$**Red**$(\{l_1:A_1,\overset{i}{\cdots},l_n:A_n\})$. $M.l_{i\overrightarrow{record}}N.l_i$ and $M\backslash l_{i\overrightarrow{record}}N\backslash l_i$ by reduction rules *RefHit* and *Rest*. By the induction hypothesis on $M.l_i$ and $M\backslash l_i$, $N.l_i\in$ **Red**(A_i) and $N\backslash l_i\in$**Red**$(\{l_1:A_1,\overset{i}{\cdots},l_n:A_n\})$. Therefore, N is reducible.

(CR3): Let M be neutral and suppose that all the M' such that $M_{\overrightarrow{record}}M'$ are reducible. Then, $M'.l_i$ are also reducible. Since M is neutral, no redex occurs at the root of $M.l_i$, hence, results of one-step reduction of term $M.l_i$ corresponds to these reducible terms $M'.l_i$. Therefore, by the induction hypothesis, $M.l_i$ are reducible. Similarly, $M\backslash l_i$ are reducible. Thus, M is reducible.
$A=B\to C$: We omit this case because the proof for this case is same as [7].

q.e.d.

Lemma 4.13. *If* $M\in$**Red**(A) *and* $N\in$**Red**$(\{l_1:A_1,\cdots,l_n:A_n\})$, *then*

$$\langle l_0=M|N\rangle\in\textbf{Red}(\{l_0:A,l_1:A_1,\cdots,l_n:A_n\}).$$

Proof. First, we show that $\langle l_0=M|N\rangle.l_0\in$**Red**$(A)$. By **(CR1)**, M and N are strongly normalizable. Thus, we can use the induction on $\nu(M)+\nu(N)$.

$\langle l_0 = M | N \rangle . l_0$ is reduced to

- M, which is reducible by the assumption.
- $\langle l_0 = M' | N \rangle . l_0$, with $M \underset{record}{\rightarrow} M'$. From **(CR2)** and $M \underset{record}{\rightarrow} M'$, it is derived that M' is reducible. By $v(M') + v(N) < v(M) + v(N)$ and the induction hypothesis on $v(M') + v(N)$, $\langle l_0 = M' | N \rangle . l_0$ is reducible.
- $\langle l_0 = M | N' \rangle . l_0$, with $N \underset{record}{\rightarrow} N'$. Similar to the above case.

Next, we show that $\langle l_0 = M | N \rangle . l_j \in \mathbf{Red}(A)$ $(j \neq 0)$. We use the induction on $v(M) + v(N)$ similarly. $\langle l_0 = M | N \rangle . l_j$ is reduced to

- $N.l_j$. $N \in \mathbf{Red}(\{l_1 : A_1, \cdots, l_n : A_n\})$, therefore, $N.l_j \in \mathbf{Red}(A_j)$ by the definition of reducible terms.
- $\langle l_0 = M' | N \rangle . l_j$ with $M \underset{record}{\rightarrow} M'$. Similar to the second case in the former.
- $\langle l_0 = M | N' \rangle . l_j$ with $N \underset{record}{\rightarrow} N'$. Similar to the third case in the former.

The rest of this proof is on $\langle l_0 = M | N \rangle \backslash l_0 \in \mathbf{Red}(\{l_1 : A_1, \cdots, l_n : A_n\})$ and $\langle l_0 = M | N \rangle \backslash l_j \in \mathbf{Red}(\{l_0 : A_0, \overset{j}{\cdots}, l_n : A_n\})$ $(j \neq 0)$. We omit its proof since the way of proving is almost same. **q.e.d.**

Lemma 4.14. $\langle\rangle \in \mathbf{Red}(\{\})$.

Proof. $\langle\rangle \in \mathbf{SN}(\{\}) = \mathbf{Red}(\{\})$. **q.e.d.**

Lemma 4.15. *If $[N/x]M$ is reducible for all reducible N, then $\lambda x : A.M$ is also reducible.*

Proof. The proof is similar to [7]. **q.e.d.**

Lemma 4.16. *Let M be any term (not assumed to be reducible), and suppose all free variables in M are among $x_1, \cdots, x_n$ of type $A_1, \cdots, A_n$. If $N_1, \cdots, N_n$ are reducible terms of types $A_1, \cdots, A_n$ then $[N_1/x_1, \cdots N_n/x_n]M$ is reducible.*

Proof. We prove this lemma by induction on M. We abbreviate $\overline{[N_n/x_n]}$ for $[N_1/x_1, \cdots N_n/x_n]$.

$M=\langle\rangle$: $\overline{[N_n/x_n]}\langle\rangle=\langle\rangle\in\mathbf{Red}(\{\})$.

$M=\langle l=M_1|M_2\rangle$: By the induction hypothesis, $\overline{[N_n/x_n]}M_1$, $\overline{[N_n/x_n]}M_2$ are reducible.
$\overline{[N_n/x_n]}(\langle l=M_1|M_2\rangle)=\langle l=\overline{[N_n/x_n]}M_1|\overline{[N_n/x_n]}M_2\rangle$ By Lemma 4.13, this is reducible.

$M=M_1.l$: By the induction hypothesis, $\overline{[N_n/x_n]}M_1$ is reducible. By the definition of reducible terms, $\overline{[N_n/x_n]}M_1.l$ is reducible. And, this is equivalent to $\overline{[N_n/x_n]}(M_1.l)$.

$M=M_1\backslash l$: Similar to the above.

the other cases: Similar to [7]. **q.e.d.**

Theorem 4.17 [*Strong Normalizability of* λ_{record}]. λ_{record} *has strong normalizability with respect to reduction* $(-)_{\overrightarrow{record}}(-)$.

Proof. Variables $x_1,\cdots,x_n$ *are reducible. Therefore, this theorem is the special case of the above proposition with* $N_1=x_1,\cdots,N_n=x_n$. **q.e.d.**

§4.2. Translations of $\lambda_{env}^{\rightarrow}$ into λ_{record}

Definition 4.18 [Translation $[\![A]\!]$]. We define a mapping $[\![-]\!]$ from types of $\lambda_{env}^{\rightarrow}$ to types of λ_{record} as:

$$[\![\{x_1:A_1\}\cdots\{x_n:A_n\}\rho]\!]=\{l_{[\![x_1]\!]}:[\![A_1]\!],\cdots,l_{[\![x_n]\!]}:[\![A_n]\!]\}$$
$$[\![\alpha]\!]=\alpha$$
$$[\![A\to B]\!]=[\![A]\!]\to[\![B]\!]$$

■

Definition 4.19 [Translation $[\![M]\!]^*(L)$]. We define a mapping from each pair of a type inference free $\Gamma\vdash M:A$ of $\lambda_{env}^{\rightarrow}$ and a term L of λ_{record} to a term of λ_{record} as:

$$[\![\{x:A\}E\vdash x:A]\!]^*(L)=L.l_{[\![x]\!]}$$
$$[\![E\vdash(\lambda x.M):(A\to B)]\!]^*(L)=\lambda x':[\![A]\!].[\![M]\!]^*(\langle l_{[\![x]\!]}=x'^{[\![A]\!]}|L\rangle),$$

where x' is a fresh variable in L and M.

$$[\![E \vdash MN : B]\!]^*(L) = [\![M]\!]^*(L)[\![N]\!]^*(L)$$
$$[\![E \vdash id : E]\!]^*(L) = L$$
$$[\![E \vdash M \circ N : A]\!]^*(L) = [\![M]\!]^*([\![N]\!]^*(L))$$
$$[\![E \vdash (M/x) \cdot N : \{x : A\}E]\!]^*(L) = \langle l_{[\![x]\!]} = [\![M]\!]^*(L) \mid [\![N]\!]^*(L) \rangle$$

We will abbreviate $[\![\Gamma \vdash M : A]\!]L$ *to* $[\![M]\!]L$, when there is no danger of confusion. ■

The following lemma is immediately checked by straightfoward observation:

Lemma 4.20. *If* $E \vdash M : A$ *in* $\lambda_{env}^{\rightarrow}$ *and* $\vdash L : [\![E]\!]$ *in* λ_{record}, *then* $\vdash [\![M]\!]^*(L) : [\![A]\!]$ *in* λ_{record}.

Lemma 4.21. *If* $M \underset{wr}{\rightarrow} M'$ *then* $[\![M]\!]^*(e) \underset{record}{\rightarrow} [\![M']\!]^*(e)$ *or* $[\![M]\!]^*(e) \equiv [\![M']\!]^*(e)$ *for a variable* e *of* λ_{record}. *Moreover, reduction rules of* $\lambda_{env}^{\rightarrow}$ *in the proof tree for* $M \underset{wr}{\rightarrow} M'$, *are respectively translated into the ones of* λ_{record}, *which derive* $[\![M]\!]^*(e) \underset{record}{\rightarrow} [\![M']\!]^*(e)$, *as follows*:

$M \underset{wr}{\rightarrow} M'$	$[\![M]\!]^*(e) \overset{*}{\underset{record}{\rightarrow}} [\![M']\!]^*(e)$
Ass	$[\![M]\!]^*(e) \equiv [\![M']\!]^*(e)$
IdL	$[\![M]\!]^*(e) \equiv [\![M']\!]^*(e)$
IdR	$[\![M]\!]^*(e) \equiv [\![M']\!]^*(e)$
DExtn	$[\![M]\!]^*(e) \equiv [\![M']\!]^*(e)$
DApp	$[\![M]\!]^*(e) \equiv [\![M']\!]^*(e)$
VarRef	*RefHit*
VarSkip	*RefSkip*
Beta1	*Beta*
Beta2	*Beta*
AppL	*AppL*
AppR	*AppR*
Lam	*Lam*
CompL	*AppL*
CompR	*AppR*

Lemma 4.22. *If there exists an infinite reduction sequence of reduction* $(-)\underset{wr}{\rightarrow}(-)$, *then it includes infinitely many Beta*1 *or Beta*2.

Proof. It is evident from Lemma 3.3. **q.e.d.**

Theorem 4.23 [*Strong Normalizability of* $\lambda^{\rightarrow}_{env}$]. *Reduction* $(-)\underset{wr}{\rightarrow}(-)$ *is strongly normalizable.*

Proof. Suppose that there exists an infinite reduction of $\lambda^{\rightarrow}_{env}$:

$$M_1 \underset{wr}{\rightarrow} M_2 \underset{wr}{\rightarrow} M_3 \underset{wr}{\rightarrow} M_4 \underset{wr}{\rightarrow} M_5 \underset{wr}{\rightarrow} \cdots$$

Let e be a variable in λ_{record}. The above reduction sequence is mapped into λ_{record}:

$$[\![M_1]\!]^*(e) \overset{*}{\underset{record}{\rightarrow}} [\![M_2]\!]^*(e) \overset{*}{\underset{record}{\rightarrow}} [\![M_3]\!]^*(e) \overset{*}{\underset{record}{\rightarrow}} \cdots.$$

By Lemma 4.22, the former reduction sequence includes infinitely many *Beta*1 or *Beta*2. Hence, the latter sequence includes infinitely many *Beta*. Therefore, this is an infinite reduction sequence in λ_{record}. This contradicts the strong normalizability of λ_{record}. **q.e.d.**

§5. Type Inference Algorithm and Principal Typing

§5.1. An Overview: Type Inference Algorithm as Automatic Proving

In this section, we will develop an algorithm which verifies typability of a given lambda-term and gives us its type if it exists, which is called *type inference algorithm*. First of all, we would like to present an overview of the type inference algorithm.

The type of a term is determined by some type inference tree which built of typing rules, and the type inference algorithm decides whether the type judgement about the given term is provable from the typing rules, or not. We therefore consider type inference algorithm as a *prover* for "logic" of typing rules, which is actually *Prolog*.

Prolog is a high-level programming language but also considered as

automatic prover for a subsystem of first-order logic. Programs are sets of restricted formulas which are called *Horn clauses*. A Horn clause is a logical formula whose shape is restricted to $p \leftarrow q_1,\cdots,q_n$ or $p \leftarrow$ where $p,q_1,\cdots,q_n$ are *atomic* predicates where free variables may occur like $p(x_1,\cdots,x_l)$.

An input for a program is given as a formula called *goal* whose shape is $\leftarrow g$. In Prolog, computation is proving process under SLD-resolution and an output is an assignment which refutes formulas given as a program and an input. If a goal $\leftarrow g$ is given and Prolog interpreter terminates successfully then we can reconstruct the proof tree for its negation $g \leftarrow$. Therefore, we may think that

- a program is a collection of non-logical axioms,
- an input is a formula to be examined its provability, and
- an output is a signal of unprovability or the proof tree proving the input's formula.

Many notions in mathematical logic and computer science, can be formalized by Horn clauses. Inference rules of Gentzen's sequent calculus, type inference rules of typed lambda calculi, Plotkin's structural operational semantics and Kahn's natural semantics of functional programming language all of these rules are defined as

$$\frac{\langle \textit{upper expression}\rangle_1,\cdots\langle \textit{upper expression}\rangle_n}{\langle \textit{lower expression}\rangle}$$

or

$$\frac{}{\langle \textit{lower expression}\rangle}.$$

It is evident that these are Horn clause if we write them as

$$\langle \textit{lower expression}\rangle \leftarrow \langle \textit{upper expression}\rangle_1,\cdots,\langle \textit{upper expression}\rangle_n.$$

and

$$\langle \textit{lower expression}\rangle \leftarrow.$$

respectively.

The typing rules given in the former section are also Horn closes. For example, typing rule *Extn* is written as

$$(E\vdash MN:B) \leftarrow (E\vdash M:A\rightarrow B),(E\vdash N:A)$$

where $((-)\vdash(-):(-))$ is a ternary predicate on environment types, terms, and types.

We obtain a type inference algorithm by giving a (Prolog's) variable to the first argument of a goal $\leftarrow((-)\vdash(-):(-))$, a term which should be tested typability and searched its type to the second, and a (Prolog's) variable to the third. After execution of this goal, the first and third argument are unified to environment type and type respectively, which satisfy a type judgement if the goal succeeds under the Horn clauses derived from typing rules. In this situation, we can build up a type inference tree from the SLD-resolution tree, i.e. the trace of the computation of this goal in prolog. The soundness and completeness of the type inference algorithm are merely corollaries of the ones of SLD-resolution. And existence of principal type is also obtained by the most generality of SLD-resolution tree.

This explanation gives us a clear view. However, there hides a trick. Terms are defined exactly by first-order terms with equality usually used in prolog and the unification on the second argument is usual first-order unification which finds the most general unifier if it exists. On the other hand, environment types are *not* a first-order term in the strict sense: An environment type

$$\{x:A\}\{y:B\}\rho$$

represents a collection of functions which correspond a variable x to a type A, y to B, and z to C. Hence, an environment type

$$\{x:A\}\{y:B\}\{z:C\}\rho'$$

is included in the environment type $\{x:A\}\{y:B\}\rho$ if we identify environment types to collections of functions.

This situation is very similar to type inference in record calculus since a record type (with a row variable) also represents a collection function from labels to types. A unification algorithm of record types with raw variables was proposed by Mitchell Wand [15] and corrected by Jategaonkar and Mitchell [11]. This algorithm gives us the most general unifier if some unifier exists. Based on this result, we present the existence of principal type of $\lambda^{\rightarrow}_{env}$.

§5.2. Preliminary

In the following several lines, we introduce a few basic notions.

Definition 5.1 [Substitution]. A *substitution on type variables and environment type variables* is a function which maps each type variable α to a type A and each environment type variable ρ to an environment type $\overline{\{x_n:A_n\}}\rho'$, such that

$$PVar(\rho)\cap\{x_1,\cdots,x_n\}=\emptyset,$$

$$PVar(\rho)\subseteq PVar(\rho'),$$

$$\text{and}\quad \{x_1,\cdots,x_n\}\subseteq PVar(\rho').$$

and the *domain* of substitution θ

$$dom(\theta)=\{\alpha\in\mathbf{TypeVar}|\alpha^\theta\neq\alpha\}\cup\{\rho\in\mathbf{EnvTypeVar}|\rho^\theta\neq\rho\}$$

is only finitely many.

A substitution θ on (environment) type variables is extended uniquely to a function $\hat{\theta}:\mathbf{Type}\to\mathbf{Type}$ as:

$$(\{x_1:A_1\}\cdots\{x_n:A_n\}\rho)^{\hat{\theta}}=\{x_1:A_1^{\hat{\theta}}\}\cdots\{x_n:A_n^{\hat{\theta}}\}\rho^\theta$$

$$\alpha^{\hat{\theta}}=\alpha^\theta$$

$$(A\to B)^{\hat{\theta}}=A^{\hat{\theta}}\to B^{\hat{\theta}}$$

In the following part of this paper, we will identify a substitution θ and its extension $\hat{\theta}$ since θ is extended *conservatively* to $\hat{\theta}$ and will call it a *substitution* simply. ■

The side condition on environment type variables which occurs in a substitution, is necessary for the following proposition:

Proposition 5.2. *Let E be an environment type and θ a substitution. Then, E^θ is an environment type.* ■

Without the side condition of the definition of substitution, for example, a substitution

$$[\rho\mapsto\{x:\alpha\}\rho'']$$

is allowed, and consequently, this causes an illegal environment type

$$(\{x:\beta\}\rho)^{[\rho\mapsto\{x:\alpha\}\rho'']}=\{x:\beta\}\{x:\alpha\}\rho''.$$

A substitution preserves the validity of a type judgement:

Proposition 5.3. *Let θ be a substitution. If $E\vdash M:A$, then $E^\theta\vdash M:A^\theta$.*

Proof. This proposition is proved straight-forwardly by structural induction on $E\vdash M:A$. **q.e.d.**

Definition 5.4 [Principal Typing]. Let E be an environment type, M a term, and A a type such that $E\vdash M:A$. The typing $E\vdash M:A$ is called *principal* if there exists a substitution θ such that $E^\theta=E'$ and $A^\theta=A'$ for any environment type E' and any type A' such that $E'\vdash M:A'$. ■

In the definition of the usual principal typing, we compare a typing $E\vdash M:A$ with $E'\vdash M:A'$, after restricting E' on $dom(E)$ since we cannot change the number of the entries in E by any substitution. However, this is not necessary in our calculus, because we can add entries by substitution. For example, consider two typings $\rho\vdash\lambda x.x:\alpha\to\alpha$ and $\{y:\beta\}\rho'\vdash\lambda x.x:\alpha\to\alpha$. Here, it holds that $\rho^\theta=\{y:\beta\}\rho'$ by $\theta=[\rho\mapsto\{y:\beta\}\rho']$.

Definition 5.5 [Restriction and Extension]. Let $\mathscr{V}$ be a set of (environment) type variables and θ a substitution.

A *restriction* $\theta|_{\mathscr{V}}$ on $\mathscr{V}$ of θ is a substitution which is same as θ except that it acts identically on $\mathscr{V}$.

τ is an *extension* of θ, or θ is *extended* to τ, if $\theta=\tau|_{dom(\theta)}$. ■

§5.3. Unification on types

In this section, we introduce a unification algorithm which computes a most general unifier for a set of type equations. This algorithm is originally developed by Mitchell Wand for record types. This algorithm has an error, however, Jategaonkar and Mitchell corrected it later.

We start with definitions of basic notions:

Definition 5.6 [Unifier]. A *unifier* θ of type A and B is a substitution such that $A^\theta=B^\theta$.

A set $\mathscr{E}$ of *unordered* pairs of types will be called a set of *type equations*, often written $A\stackrel{?}{=}B$ for types A and B. A set of substitutions which unify every pair in $\mathscr{E}$ is written as $\mathscr{U}(\mathscr{E})$. ■

Definition 5.7 [Generality of Unifier and Most General Unifier]. Let θ and τ be unifiers of type A and B. We say that θ is *more general* than τ if there exists a substitution τ' such that $\tau = \theta\tau'$.

A unifier θ of type A and B is called the *most general unifier*, or *mgu*, if θ is more general than any unifier of A and B. ■

Notation 5.8. We abbreviate $\{x_1 : A_1\} \cdots \{x_p : A_p\}\rho$ to $\overline{\{x_p : A_p\}}\rho$, $A_1 \stackrel{?}{=} A'_1$, $\cdots, A'_p \stackrel{?}{=} A'_p$ to $\overline{A_p \stackrel{?}{=} A'_p}$, and $\{x_1, \cdots, x_p\}$ to $\{\overline{x_p}\}$. ■

In advance of definition of the unification transformation, we prepare a fundamental notion for it, which specifies a successful case of normal forms of this transformation.

Definition 5.9 [Solved Type-equation]. A type equation $\alpha \stackrel{?}{=} A$ (or $\rho \stackrel{?}{=} E$) is called *solved* in a set of $\mathscr{E}$ of type equations, if α (or ρ, resp.) does not occur anywhere else in $\mathscr{E}$. This α (or ρ resp.) is called *solved variable*. $\mathscr{E}$ is called *solved* if all its pairs are solved.

If $\mathscr{E}$ is solved, it is clear that $\mathscr{E}$ has a form:

$$\{\alpha_1 \stackrel{?}{=} A_1, \cdots, \alpha_m \stackrel{?}{=} A_m, \rho_1 \stackrel{?}{=} E_1, \cdots, \rho_n \stackrel{?}{=} E_n\},$$

where $\alpha_1, \cdots, \alpha_m, \rho_1, \cdots, \rho_m$ are distinct with each other and any α_i and ρ_j does not occur anywhere else except themselves.

Therefore, we make the following substitution from $\mathscr{E}$:

$$[\alpha_1 \mapsto A_1, \cdots, \alpha_m \mapsto A_m, \rho_1 \mapsto E_1, \cdots, \rho_n \mapsto E_n].$$

We identify this substitution to $\mathscr{E}$ itself. ■

Definition 5.10 [Unification Transformer]. Next, we define a transformation $(\mathscr{E}, \mathscr{V}) \Rightarrow (\mathscr{E}', \mathscr{V}')$ which maps each pair of a set $\mathscr{E}$ of type equations and a set $\mathscr{V}$ of variables occurring in $\mathscr{E}$, to a pair of $\mathscr{E}'$ and $\mathscr{V}'$ of the same kind respectively, as follows:

Tr Tvar Tvar $\quad (\mathscr{E} \cup \{\alpha \stackrel{?}{=} \alpha\},\ \mathscr{V})$

$$\Rightarrow (\mathscr{E}, \mathscr{V})$$

Tr Tvar Type $(\mathscr{E} \cup \{\alpha \stackrel{?}{=} A\},\ \mathscr{V})$

$\Rightarrow (\mathscr{E}^{[\alpha \mapsto A]} \cup \{\alpha \stackrel{?}{=} A\},\ \mathscr{V})$

where $\alpha \stackrel{?}{=} A$ is not solved in $\mathscr{E} \cup \{\alpha \stackrel{?}{=} A\}$, and $\alpha \notin ftv(A)$.

Tr Fun Fun $(\mathscr{E} \cup \{A \to B \stackrel{?}{=} C \to D\},\ \mathscr{V})$

$\Rightarrow (\mathscr{E} \cup \{A \stackrel{?}{=} C, B \stackrel{?}{=} D\},\ \mathscr{V})$

Tr Evar Evar $(\mathscr{E} \cup \{\rho \stackrel{?}{=} \rho\},\ \mathscr{V})$

$\Rightarrow (\mathscr{E}, \mathscr{V})$

Tr Evar Etype $(\mathscr{E} \cup \{\rho \stackrel{?}{=} E\},\ \mathscr{V})$

$\Rightarrow (\mathscr{E}^{[\rho \mapsto E]} \cup \{\rho \stackrel{?}{=} E\},\ \mathscr{V})$

where $\rho \stackrel{?}{=} E$ is not solved in $\mathscr{E} \cup \{\rho \stackrel{?}{=} E\}$,
$[\rho \mapsto E]$ is a valid substitution, and $\rho \notin ftv(E)$

Tr Etype Etype1 $(\mathscr{E} \cup \{\overline{\{x_p : A_p\}}\rho \stackrel{?}{=} \overline{\{x_p : A'_p\}}\rho\},\ \mathscr{V})$

$\Rightarrow (\mathscr{E} \cup \{\overline{A_p \stackrel{?}{=} A'_p}\},\ \mathscr{V})$

Tr Etype Etype2 $(\mathscr{E} \cup \{\overline{\{x_p : A_p\}}\ \overline{\{y_q : B_q\}}\rho \stackrel{?}{=} \overline{\{x_p : A'_p\}}\ \overline{\{z_r : C_r\}}\rho'\},\ \mathscr{V})$

$\Rightarrow (\mathscr{E}^\theta \cup \{\overline{A_p^\theta \stackrel{?}{=} A'^\theta_p}, \rho \stackrel{?}{=} \overline{\{z_r : C_r^\theta\}}\rho'', \rho' \stackrel{?}{=} \overline{\{y_q : B_q^\theta\}}\rho''\},\ \mathscr{V} \cup \{\rho''\})$

where $\rho \neq \rho'$, $\rho'' \notin \mathscr{V}$,
$PVar(\rho'') = PVar(\rho) \cup PVar(\rho')$,
$PVar(\rho) \cap \{\overline{z_r}\} = \emptyset$, $PVar(\rho') \cap \{\overline{y_q}\} = \emptyset$,
$\forall k(\rho \notin ftv(C_k))$, $\forall j(\rho' \notin ftv(B_j))$,
$\forall k(\rho \notin ftv(B_k)) \vee \forall j(\rho' \notin ftv(C_j))$, and
$\theta = [\rho \mapsto \overline{\{z_r : C_r^{[\rho' \mapsto \overline{\{y_q : B_q\}}\rho'']}\}}\rho'', \rho' \mapsto \overline{\{y_q : B_q^{[\rho \mapsto \overline{\{z_r : C_r\}}\rho'']}\}}\rho'']$

It is evident that $\mathscr{V}'$ includes every variable occurring in $\mathscr{E}'$ in any $(\mathscr{E}, \mathscr{V}) \Rightarrow (\mathscr{E}', \mathscr{V}')$ ■

The following rules are special instances of *Tr Tvar Type* and *Tr Evar Etype*, respectively:

Tr Tvar Tvar Coalesce $(\mathscr{E} \cup \{\alpha \stackrel{?}{=} \beta\}, \mathscr{V})$

$\Rightarrow (\mathscr{E}^{[\alpha \mapsto \beta]} \cup \{\alpha \stackrel{?}{=} \beta\}, \mathscr{V})$

where $\alpha, \beta \in ftv(\mathscr{E})$ and $\alpha \neq \beta$,

Tr Evar Evar Coalesce $(\mathscr{E} \cup \{\rho \stackrel{?}{=} \rho'\}, \mathscr{V})$

$\Rightarrow (\mathscr{E}^{[\rho \to \rho']} \cup \{\rho \stackrel{?}{=} \rho'\}, \mathscr{V})$

where $\rho, \rho' \in ftv(\mathscr{E})$ and $\rho \neq \rho'$.

In transformation *Tr Etype Etype2*, the readers should be note that the existence of ρ'' is guaranteed by the condition that a set $PVar^{-1}(\{x_1, \cdots, x_n\})$ is infinitely many for a given $\{x_1, \cdots, x_n\}$, in other words, we can get a fresh ρ''.

The last rule *Tr Etype Etype2* seems to be complicated and unnatural, however the following lemma reveals the secret of its complication.

Lemma 5.11. *Let a rule Tr Etype Etype0 be*

$$(\mathscr{E} \cup \{\overline{\{x_p : A_p\}}\ \overline{\{y_q : B_q\}}\rho \stackrel{?}{=} \overline{\{x_p : A'_p\}}\ \overline{\{z_r : C_r\}}\rho'\}, \mathscr{V})$$

$$\Rightarrow (\mathscr{E} \cup \{\overline{A_p \stackrel{?}{=} A'_p}, \rho \stackrel{?}{=} \overline{\{z_r : C_r\}}\rho'', \rho' \stackrel{?}{=} \overline{\{y_q : B_q\}}\rho''\}, \mathscr{V} \cup \{\rho''\})$$

if $\rho'' \notin \mathscr{V}$, *and* $PVar(\rho'') = PVar(\rho) \cup PVar(\rho')$.

If

$$(\mathscr{E}, \mathscr{V}) \quad \Rightarrow \quad (\mathscr{E}', \mathscr{V}')$$
$$Tr\ Etype\ Etype2$$

then

$$(\mathscr{E}, \mathscr{V}) \quad \Rightarrow \quad \stackrel{*}{\Rightarrow} \quad (\mathscr{E}', \mathscr{V}').$$
$$Tr\ Etype\ Etype0\ \ Tr\ Evar\ Etype$$

Proof. Let $(\mathscr{E}_1, \mathscr{V}_1)$ be

$$(\mathscr{E} \cup \{\overline{\{x_p : A_p\}}\ \overline{\{y_q : B_q\}}\rho \stackrel{?}{=} \overline{\{x_p : A'_p\}}\ \overline{\{z_r : C_r\}}\rho'\}, \mathscr{V})$$

satisfying the side condition of *Tr Etype Etype*2.

We can apply *Tr Etype Etype*0 to $(\mathscr{E}_1, \mathscr{V}_1)$ because the side condition of *Tr Etype Etype*2 includes the one of *Tr Etype Etype*0. The result is as follows:

$$(\mathscr{E}_2, \mathscr{V}_2) = (\mathscr{E} \cup \{\overline{A_p \stackrel{?}{=} A'_p}, \rho \stackrel{?}{=} \overline{\{z_r : C_r\}}\rho', \rho' \stackrel{?}{=} \overline{\{y_q : B_q\}}\rho''\}, \ \mathscr{V} \cup \{\rho''\}).$$

If $\rho \stackrel{?}{=} \overline{\{z_r : C_r\}}\rho''$ is not solved in $\mathscr{E}_2$, then this pair is transformed to

$$(\mathscr{E}_3, \mathscr{V}_3) = (\mathscr{E}^{\theta_1} \cup \{\overline{A_p^{\theta_1} \stackrel{?}{=} A_p'^{\theta_1}}, \rho \stackrel{?}{=} \overline{\{z_r : C_r\}}\rho'', \rho' \stackrel{?}{=} \overline{\{y_q : B_q^{\theta_1}\}}\rho''\}, \ \mathscr{V} \cup \{\rho''\}),$$

where $\theta_1 = [\rho \mapsto \overline{\{z_r : C_r\}}\rho'']$.

Otherwise, $\rho \stackrel{?}{=} \overline{\{z_r : C_r\}}\rho$ is solved, then ρ does not occur anywhere else except in this pair. Therefore, $\mathscr{E}_2 = \mathscr{E}_3$.

Similarly, $(\mathscr{E}_3, \mathscr{V}_3)$ is transformed to or equivalent to

$$(\mathscr{E}_4, \mathscr{V}_4) = (\mathscr{E}^{\theta_1\theta_2} \cup \{\overline{A_p^{\theta_1\theta_2} \stackrel{?}{=} A_p'^{\theta_1\theta_2}}, \rho \stackrel{?}{=} \overline{\{z_r : C_r^{\theta_2}\}}\rho'', \rho' \stackrel{?}{=} \overline{\{y_q : B_q^{\theta_1}\}}\rho''\}, \ \mathscr{V} \cup \{\rho''\})$$

where $\theta_2 = [\rho' \mapsto \overline{\{y_q : B_q^{\theta_1}\}}\rho'']$.

Noting that $\forall k(\rho \notin ftv(C_k)), \forall j(\rho' \notin ftv(B_j))$, and $\forall k(\rho \notin ftv(B_k)) \vee \forall j(\rho' \notin ftv(C_j))$, we know that

$$\theta_1\theta_2 = \theta,$$
$$C_r^{\theta_2} = C_r^{\theta}, \text{ and}$$
$$B_q^{\theta_1} = B_q^{\theta}.$$

Therefore,

$$(\mathscr{E}_1, \mathscr{V}_1) \underset{Tr\ Etype\ Etype0}{\Rightarrow} \overset{*}{\underset{Tr\ Evar\ Etype}{\Rightarrow}} (\mathscr{E}_4, \mathscr{V}_4)$$

and $$(\mathscr{E}_1, \mathscr{V}_1) \underset{Tr\ Etype\ Etype2}{\Rightarrow} (\mathscr{E}_4, \mathscr{V}_4).$$ **q.e.d.**

By this lemma, we know that the rule *Tr Etype Etype* is derived from *Tr Etype Etype*0. The reason why we introduce *Tr Etype Etype* rather than more intuitive *Tr Etype Etype*0 is that length of $\mathscr{E}$ does not decreases strictly in *Tr Etype Etype*0 but do in *Tr Etype Etype*. This is shown in Lemma 5.18.

Lemma 5.12. *If* $(\mathscr{E}_1, \mathscr{V}_1) \Rightarrow (\mathscr{E}_2, \mathscr{V}_2)$, *then* $\mathscr{U}(\mathscr{E}_1) \supseteq \mathscr{U}(\mathscr{E}_2)$.

Proof. We prove this lemma by the case analysis on $\Rightarrow$.

The cases of *Tr Tvar Tvar*, *Tr Tvar Type*, *Tr Fun Fun*, *Tr Evar Evar*, and *Tr Etype Etype*1 are easy.

The case of *Tr Etype Etype*2:

Suppose that

$$(\mathscr{E}_1, \mathscr{V}_1) \underset{Tr\ Etype\ Etype2}{\Rightarrow} (\mathscr{E}_2, \mathscr{V}_2).$$

By Lemma 5.11, this sequence is decomposed to

$$(\mathscr{E}_1, \mathscr{V}_1) \underset{Tr\ Etype\ Etype0}{\Rightarrow} (\mathscr{E}_1', \mathscr{V}_1') \underset{Tr\ Evar\ Evar}{\overset{*}{\Rightarrow}} (\mathscr{E}_2, \mathscr{V}_2).$$

$\mathscr{U}(\mathscr{E}_1') \supseteq \mathscr{U}(\mathscr{E}_2)$ holds by the result of the case already proved. $\mathscr{U}(\mathscr{E}_1) \supseteq \mathscr{U}(\mathscr{E}_1')$ is clear. **q.e.d.**

The following lemma shows us syntactical characterization of normal forms in this transformation

Lemma 5.13. *If* $\mathscr{E}$ *is solved or consists of the following type equations, then any rule cannot be applied:*

- $\alpha \stackrel{?}{=} A$ *where* $\alpha \in ftv(A)$;
- $\rho \stackrel{?}{=} A \to B$;
- $\rho \stackrel{?}{=} E$, *not satisfying the side condition of Tr Evar Etype*;
- $\overline{\{x_p : A_p\}}\,\overline{\{y_q : B_q\}}\rho \stackrel{?}{=} \overline{\{x_p : A_p'\}}\,\overline{\{z_r : C_r\}}\rho'$

*where Tr Etype Etype*1 *is not applicable and the side condition of Tr Etype Etype*2 *is not satisfied.*

Proof. It is clear by observation of the definition of unification transformation. **q.e.d.**

Corollary 5.14. Suppose that no rule is applicable to $(\mathscr{E}_0, \mathscr{V}_0)$. If $\mathscr{E}_0$ is unifiable then $\mathscr{E}_0$ is solved.

Proof. This is clear, noting that the four cases in the above lemma are not unifiable. **q.e.d.**

Lemma 5.15. *Let $\mathscr{E}_0$ be solved. Then, $\mathscr{E}_0$, which is regarded as a substitution, is the most general unifier of $\mathscr{E}_0$.*

Proof. It is evident that a substitution $\mathscr{E}_0$ unifies a set $\mathscr{E}_0$ of type equations. Next, we prove that $\mathscr{E}_0$ is the most general.

Let θ be a unifier of $\mathscr{E}_0$. Then, it holds that $\theta = \mathscr{E}_0\theta$ by the following reason: for each type variable α, if α is a solved variable of $\mathscr{E}_0$, then

$$\begin{aligned}\alpha^\theta &= A^\theta \quad (\text{because } \theta \in \mathscr{U}(\mathscr{E}_0)) \\ &= (\alpha^{\mathscr{E}_0})^\theta \\ &= \alpha^{(\mathscr{E}_0\theta)},\end{aligned}$$

otherwise, i.e. α is not a solved variable,

$$\begin{aligned}\alpha^\theta &= (\alpha^{\mathscr{E}_0})^\theta \quad (\text{because the substitution } \mathscr{E}_0 \text{ maps the } \alpha \text{ to itself.}) \\ &= \alpha^{(\mathscr{E}_0\theta)}.\end{aligned}$$

Therefore, for every type variable α, $\alpha^\theta = \alpha^{(\mathscr{E}_0\theta)}$. Similarly, for every environment type variable ρ_X, $\rho_X^\theta = \rho_X^{(\mathscr{E}_0\theta)}$.

Thus, $\theta = \mathscr{E}_0\theta$. This shows that $\mathscr{E}_0$ is more general than θ and that $\mathscr{E}_0$ is the most general. **q.e.d.**

Lemma 5.16 [*Soundness*]. *If $(\mathscr{E}, \mathscr{V}) \overset{*}{\Rightarrow} (\mathscr{E}_0, \mathscr{V}_0)$ and $\mathscr{E}_0$ is solved, then substitution $\mathscr{E}_0$ is a unifier of $\mathscr{E}$.*

Proof. By Lemma 5.12, it holds that $\mathscr{U}(\mathscr{E}) \supseteq \mathscr{U}(\mathscr{E}_0)$. And by Lemma 5.15, $\mathscr{E}_0 \in \mathscr{U}(\mathscr{E}_0)$. Hence, $\mathscr{E}_0 \in \mathscr{U}(\mathscr{E})$. **q.e.d.**

Definition 5.17 [Length of a Set of Type Equations]. A *length* for a set $\{A_1 \overset{?}{=} B_1 \cdots A_n \overset{?}{=} B_n\}$ of type equations is defined as

$$length(\{A_1 \stackrel{?}{=} B_1 \cdots A_n \stackrel{?}{=} B_n\}) = length(A_1) \cdot length(B_1) + \cdots + length(A_n) \cdot length(B_n),$$

especially,

$$length(\emptyset) = 0. \qquad \blacksquare$$

Lemma 5.18 [*Terminating Property of Unification Transformation*]. *The transformation is noetherian: there is no infinite sequence like*

$$(\mathscr{E}, \mathscr{V}) \Rightarrow (\mathscr{E}_1, \mathscr{V}_1) \Rightarrow (\mathscr{E}_2, \mathscr{V}_2) \Rightarrow \cdots$$

for any $\mathscr{E}$ *and any* $\mathscr{V}$.

Proof. Consider the dictionary ordering on pairs like

$$\langle \text{number of unsolved variables}, length(\mathscr{E}) \rangle.$$

Then, in each rule of the transformation: $(\mathscr{E}, \mathscr{V}) \Rightarrow (\mathscr{E}', \mathscr{V}')$, $\mathscr{E}$ is strictly smaller than $\mathscr{E}'$, therefore, the sequence of the transformation must terminate. **q.e.d.**

We should note that the length does not decrease for *Tr Etype Etype*0 while it decrease for *Tr Etype Etype*2. This is the reason why we adopt *Tr Etype Etype*2 as a rule instead of *Tr Etype Etype*0.

Lemma 5.19. *Suppose that* $(\mathscr{E}_1, \mathscr{V}_1) \Rightarrow (\mathscr{E}_2, \mathscr{V}_2)$. *If* $\theta_1 \in \mathscr{U}(\mathscr{E}_1)$ *satisfies* $dom(\theta_1) \subseteq \mathscr{V}_1$, *then there exists an extension* θ_2 of θ_1 *which unifies* $\mathscr{E}_2$ *and satisfies* $dom(\theta_2) \subseteq \mathscr{V}_2$.

Before starting the proof, we make a comment to this lemma: for the usual unification transformation for first-order terms, more simple statement holds:

$$\text{if} \quad \theta \in \mathscr{U}(\mathscr{E}_1) \text{ then } \theta \in \mathscr{U}(\mathscr{E}_2), \text{ that is, } \mathscr{U}(\mathscr{E}_1) \subseteq \mathscr{U}(\mathscr{E}_2).$$

The reason why this statement does not hold is that the unification transformation studied here may introduce fresh variables. The difference of this lemma to the above statement is due to this point. The reason why $\theta_1 \in \mathscr{U}(\mathscr{E}_2)$ does not hold is that $\mathscr{E}_2$ may include fresh type variables during the transformation. For example, consider

$$(\{\{x:\alpha\}\rho_{\{x\}} \stackrel{?}{=} \{y:\beta\}\rho'_{\{y\}}\},\ \{\alpha,\beta,\rho_{\{x\}},\rho'_{\{y\}}\})$$

$$\Rightarrow (\{\rho_{\{x\}} \stackrel{?}{=} \{y:\beta\}\rho''_{\{x,y\}}, \rho_{\{y\}} \stackrel{?}{=} \{x:\alpha\}\rho''_{\{x,y\}}\},\ \{\alpha,\beta,\rho_{\{x\}},\rho'_{\{y\}},\rho''_{\{x,y\}}\})$$

A substitution

$$\theta_1 = \begin{bmatrix} \rho_{\{x\}} \mapsto \{y:\beta\}\{z:\gamma\to\gamma\}\rho'''_{\{x,y,z\}}, \\ \rho'_{\{y\}} \mapsto \{x:\alpha\}\{z:\gamma\to\gamma\}\rho'''_{\{x,y,z\}}, \\ \rho''_{\{x,y\}} \mapsto \{z:\gamma\}\rho'''_{\{x,y,z\}} \end{bmatrix}$$

unifies $\mathscr{E}_1$ but does not $\mathscr{E}_2$.

Proof. We will prove this lemma by case analysis on the transformation.

In the cases of *Tr Tvar Tvar*, *Tr Tvar Type*, *Tr Fun Fun*, *Tr Evar Evar*, and *Tr Evar Etype*, θ_2 is actually θ_1 itself and the proof is easy.

The case of *Tr Etype Etype* is more complicated than the other cases since a fresh variable is introduced in this step:

Suppose that

$$\mathscr{E}_1 = \mathscr{E}'_1 \cup \{\overline{\{x_p:A_p\}}\,\overline{\{y_q:B_q\}}\rho \stackrel{?}{=} \overline{\{x_p:A'_p\}}\,\overline{\{z_r:C_r\}}\rho'\}$$

$$(\mathscr{E}_1,\mathscr{V}_1) \quad \Rightarrow \quad (\mathscr{E}_2,\mathscr{V}_2),$$
Tr Etype Etype2

$\theta_1 \in \mathscr{U}(\mathscr{E}_1)$ and

$dom(\theta_1) \subseteq \mathscr{V}_1$.

We next try to construct $\theta_2 \in \mathscr{U}(\mathscr{E}_2)$ satisfying $\theta_1 = \theta_2|_{dom(\theta_1)}$, from θ_1.

Since $\theta_1 \in \mathscr{U}(\mathscr{E}_1)$,

$$A_i^{\theta_1} = A_i'^{\theta_1}\ (i=1,\cdots,p);$$
$$\overline{\{y_q:B_q^{\theta_1}\}}(\rho^{\theta_1}) = \overline{\{z_r:C_r^{\theta_1}\}}(\rho'^{\theta_1}).$$

From the second equation, both of $\overline{\{y_q:B_q^{\theta_1}\}}(\rho^{\theta_1})$ and $\overline{\{z_r:C_r^{\theta_1}\}}(\rho'^{\theta_1})$ have the same form

$$\overline{\{y_q:B_q^{\theta_1}\}\{z_r:C_r^{\theta_1}\}}E$$

where E is an environment type.

Let $\theta_2 = \theta_1[\rho''_{X\cup X'} \mapsto E]$, noticing that $\rho''_{X\cup X'} \notin dom(\mathscr{E}_1) \subseteq \mathscr{V}_1$.

Here, remind that

$$(\mathscr{E}_1, \mathscr{V}_1) \overset{Tr\ Etype\ Etype}{\Rightarrow} (\mathscr{E}_2, \mathscr{V}_2)$$

is decomposed to

$$(\mathscr{E}_1, \mathscr{V}_1) \overset{Tr\ Etype\ Etype0}{\Rightarrow} (\mathscr{E}_3, \mathscr{V}_2) \overset{Tr\ Evar\ Etype}{\Rightarrow} (\mathscr{E}_4, \mathscr{V}_2) \overset{Tr\ Evar\ Etype}{\Rightarrow} (\mathscr{E}_2, \mathscr{V}_2)$$

by Lemma 5.11. It is now clear that $\theta_2 \in \mathscr{U}(\mathscr{E}_3)$ and $\theta_1 = \theta_2|_{dom(\theta_1)}$ by observation of the definition of *Tr Etype Etype*0. $\theta_2 \in \mathscr{U}(\mathscr{E}_4)$ and therefore $\theta_2 \in \mathscr{U}(\mathscr{E}_2)$ are derived from the former case of this proof.

Hence, we know that θ_2 is an extension of θ_1 (from $\theta_2 \in \mathscr{U}(\mathscr{E}_2)$ and $\theta_1 = \theta_2|_{dom(\theta_1)}$).

The rest of the proof, $dom(\theta_2) \subseteq \mathscr{V}_2$, is evident since $dom(\theta_2) = dom(\theta_1) \cup \{\rho''_{X \cup X''}\}$ and $\mathscr{V}_2 = \mathscr{V}_1 \cup \{\rho''_{X \cup X'}\}$. **q.e.d.**

Theorem 5.20 [*Completeness of Unification*]. *If there exists a unifier* θ *satisfying* $dom(\theta) \subseteq \mathscr{V}$, *then any sequence of unification transformation,*

$$(\mathscr{E}, \mathscr{V}) \Rightarrow \cdots$$

eventually terminates in $(\mathscr{E}_0, \mathscr{V}_0)$ *where* $\mathscr{E}_0$ *is solved. And in this case,* $\mathscr{E}_0|_{\mathscr{V}}$ *is more general than any unifier* θ *of* $\mathscr{E}$ *satisfying* $dom(\theta) \subseteq \mathscr{V}$.

Proof. Termination of the transformation is proved in Lemma 5.18.

Next, let us prove that $\mathscr{E}_0$ is solved: By Lemma 5.19, we can extend θ to a unifier θ' of $\mathscr{E}$ such that $\theta = \theta'|_{dom(\theta)}$. $\mathscr{E}_0$ can not be transformed any more. By Corollary 5.14, $\mathscr{E}_0$ is solved.

Finally, we will show that a unifier made from $\mathscr{E}_0|_{\mathscr{V}}$ is more general than any θ of $\mathscr{E}$ satisfying $dom(\theta) \subseteq \mathscr{V}$: By Lemma 5.19, there exists a unifier θ' such that $\theta = \theta'|_{dom(\theta)}$.

$$\begin{array}{ccccccc} (\mathscr{E}, \mathscr{V}) & & \Rightarrow & \cdots & \Rightarrow & & (\mathscr{E}_0, \mathscr{V}_0) \\ \updownarrow \text{unify} & & & & & & \updownarrow \text{unify} \\ \theta & & \overset{\text{Lemma 5.19}}{\rightarrow} & \cdots & \overset{\text{Lemma 5.19}}{\rightarrow} & & \theta' \end{array}$$

Since substitution $\mathscr{E}_0$ is the most general unifier of $\mathscr{E}_0$ by Lemma 5.13, there exists a unifier τ such that $\theta' = \mathscr{E}_0\tau$. By the definition of θ',

$$\theta=(\theta'|_{dom(\theta)})|_{\mathscr{V}}=((\mathscr{E}_0\tau)|_{dom(\theta)})|_{\mathscr{V}}=(\mathscr{E}_0\tau)|_{\mathscr{V}}=(\mathscr{E}_0|_{\mathscr{V}})\tau.$$

q.e.d.

§5.4. Type Inference Algorithm

In this section, we present a type inference algorithm which finds the principal type for a given term. The algorithm consists of two parts: one is the unification algorithm and the other the transformation which generates a set of equations from an ordered sequence of "type judgements" which should hold after some suitable substitution. More accurately, the algorithm receives an ordered sequence $\mathscr{J}$ of triples of an environment type, a term, and a type, and returns a set $\mathscr{E}$ of type equations. And if there exists the most general unifier θ for $\mathscr{E}$, then each entry (E,M,A) in the sequence $\mathscr{J}$ becomes a valid type judgement $E^{\theta}\vdash M:A^{\theta}$ by applying the substitution θ to the set $\mathscr{J}$. In the following of this paper, we will call the triple (E,M,A) a typing candidate:

Definition 5.21 [Typing Candidate: $E\rhd M:A$]. Let E be an environment type, M a term, and A a type. A *typing candidate* $E\rhd M:A$ is a triple of E, M, and A. ■

Definition 5.22 [Equation Extractor: $(\mathscr{J},\mathscr{V})\Downarrow(\mathscr{E},\mathscr{V}')$]. We define inductively a transformation, called an *equation extractor*,

$$(\mathscr{J},\mathscr{V})\Downarrow(\mathscr{E},\mathscr{V}')$$

which maps each pair $(\mathscr{J},\mathscr{V})$ of an ordered sequence $\mathscr{J}$ of typing candidates and a set $\mathscr{V}$ of (environment) type variables, to a pair $(\mathscr{E},\mathscr{V}')$ of a set $\mathscr{E}$ of type equations and a set $\mathscr{V}'$ of (environment) type variables, by the following rules:

$$\frac{}{(\textit{Empty Sequence},\mathscr{V})\Downarrow(\emptyset,\mathscr{V})}\textit{Eq Ex Empty}$$

where *Empty Sequence* is the empty sequence.

$$\frac{(\mathscr{J},\mathscr{V})\Downarrow(\mathscr{E},\mathscr{V}')\quad \rho_{\{x\}}\notin\mathscr{V}'}{((E\rhd x:A)::\mathscr{J},\mathscr{V})\Downarrow(\{E\stackrel{?}{=}\{x:A\}\rho\}\cup\mathscr{E},\mathscr{V}'\cup\{\rho\})}\textit{Eq Ex Var}$$

$$\frac{\alpha,\beta\notin\mathscr{V}\quad \alpha\neq\beta\quad ((\{x:\alpha\}E\rhd M:\beta)::\mathscr{J},\mathscr{V}\cup\{\alpha,\beta\})\Downarrow(\mathscr{E},\mathscr{V}')}{((E\rhd\lambda x.M:C)::\mathscr{J},\mathscr{V})\Downarrow(\{C\stackrel{?}{=}\alpha\to\beta\}\cup\mathscr{E},\mathscr{V}')}\textit{Eq Ex Lam}$$

$$\frac{\alpha \notin \mathscr{V} \quad ((E \rhd M : \alpha \to B) :: (E \rhd N : \alpha) :: \mathscr{J}, \mathscr{V} \cup \{\alpha\}) \Downarrow (\mathscr{E}, \mathscr{V}')}{((E \rhd (MN) : B) :: \mathscr{J}, \mathscr{V}) \Downarrow (\mathscr{E}, \mathscr{V}')} \textit{Eq Ex App}$$

$$\frac{(\mathscr{J}, \mathscr{V}) \Downarrow (\mathscr{E}, \mathscr{V}')}{((E \rhd id : A) :: \mathscr{J}, \mathscr{V}) \Downarrow (\{E \stackrel{?}{=} A\} \cup \mathscr{E}, \mathscr{V}')} \textit{Eq Ex Id}$$

$$\frac{\alpha, \rho_{\{x\}} \notin \mathscr{V} \quad ((E \rhd M : \alpha) :: (E \rhd N : \rho) :: \mathscr{J}, \mathscr{V} \cup \{\alpha, \rho\}) \Downarrow (\mathscr{E}, \mathscr{V}')}{((E \rhd (M/x) \cdot N : A) :: \mathscr{J}, \mathscr{V}) \Downarrow (\{A \stackrel{?}{=} \{x : \alpha\}\rho\} \cup \mathscr{E}, \mathscr{V}')} \textit{Eq Ex Extn}$$

$$\frac{\rho_0 \notin \mathscr{V} \quad ((E \rhd N : \rho) :: (\rho \rhd M : A) :: \mathscr{J}, \mathscr{V} \cup \{\rho\}) \Downarrow (\mathscr{E}, \mathscr{V}')}{((E \rhd M \circ N : A) :: \mathscr{J}, \mathscr{V}) \Downarrow (\mathscr{E}, \mathscr{V}')} \textit{Eq Ex Comp} \qquad \blacksquare$$

Lemma 5.23 [*Decidability of Equation Extractor*]. *Given an ordered sequence $\mathscr{J}$ of typing candidates and a set $\mathscr{V}$ of (environment) type variables occurring in $\mathscr{J}$, then we can find a finite set $\mathscr{E}$ of type equations and a set $\mathscr{V}'$ of type variables and environment type variables such that $(\mathscr{J}, \mathscr{V}) \Downarrow (\mathscr{E}, \mathscr{V}')$.*

Proof. It is proved straight-forwardly by induction on *length*($\mathscr{J}$) of a sequence $\mathscr{J}$ of type candidates, noting that the following assertion holds:

$((E \rhd M : A) :: \mathscr{J}, \mathscr{V}) \Downarrow (\mathscr{E}', \mathscr{V}')$ is determined *only* by the shape of M.

q.e.d.

length($\mathscr{J}$) is defined as follows:

Definition 5.24 [Length of $\mathscr{J}$: *length*($\mathscr{J}$)]. For each finite ordered-sequence $\mathscr{J}$ of typing candidates, *length*($\mathscr{J}$) is defined inductively as

$$\textit{length}(\textit{Empty Sequence}) = 0;$$

$$\textit{length}((E \rhd M : A) :: \mathscr{J}) = \textit{length}(M) + \textit{length}(\mathscr{J}). \qquad \blacksquare$$

Lemma 5.25. *Let $\mathscr{J}$ be a finite ordered sequence of typing candidates, $\mathscr{V}$ and $\mathscr{V}'$ sets of type variables and environment type variables, $\mathscr{E}$ a finite set of type equations, satisfying $(\mathscr{J}, \mathscr{V}) \Downarrow (\mathscr{E}, \mathscr{V}')$.*

If there exists a unifier θ for $\mathscr{E}$, then $E^\theta \vdash M : A^\theta$ holds for each $E \rhd M : A \in \mathscr{J}$.

Proof. The proof is straight-forward if we use structural induction on derivation tree of equation extractor $(-,-)\Downarrow(-,-)$. **q.e.d.**

We next obtain a type inference algorithm from the preceding lemmas:

Definition 5.26 [Type Inference Algorithm: *TypeInfer*].

Input: a term M.

Step 1: introduce an environment type variable $\rho_\emptyset$ and a type variable α;

Step 2: find $\mathscr{E}$ and $\mathscr{V}$ satisfying

$$((\rho_\emptyset \rhd M:\alpha)::\mathit{Empty\ Sequence},\{\rho_\emptyset,\alpha\})\quad(\mathscr{E},\mathscr{V});$$

Step 3: apply the unification transformation to $(\mathscr{E},\mathscr{V})$ as many as possible:

$$(\mathscr{E},\mathscr{V})\overset{*}{\Rightarrow}(\mathscr{E}_0,\mathscr{V}')\Rightarrow;$$

Step 4: if $\mathscr{E}_0$ is solved then output $\rho_\emptyset^{\mathscr{E}_0}$ and $\alpha^{\mathscr{E}_0}$ else halt with failure. ■

Theorem 5.27 [*Soundness of Type Infer*]. *Type inference algorithm Type Infer terminates, and if it succeeds, then* $\rho^{\mathscr{E}_0}\vdash M:\alpha^{\mathscr{E}_0}$ *holds for outputs* $\rho^{\mathscr{E}_0}$ *and* $\alpha^{\mathscr{E}_0}$.

Proof. The former assertion, the termination of this algorithm, is derived by the decidability of equation extractor $(-,-)\Downarrow(-,-)$ (Lemma 5.23) and the termination of unification transformation (Lemma 5.18).

And, the second assertion, $\rho_\emptyset^\theta\vdash M:\alpha^\theta$, is derived from Lemma 5.25 and 5.16. **q.e.d.**

Lemma 5.28. *Suppose that* $(\mathscr{J},\mathscr{V})\Downarrow(\mathscr{E},\mathscr{V}')$ *and* $\mathscr{V}$ *includes every type variable and every environment type variable occurring in a sequence* $\mathscr{J}$ *of typing candidates.*

If we are given a substitution θ *such that,* $dom(\theta)\subseteq\mathscr{V}$ *and* $E^\theta\vdash M:A^\theta$ *for each typing candidate* $E\rhd M:A$ *in* $\mathscr{J}$, *then we can find an extension* θ' *of* θ *which unifies* $\mathscr{E}$ *and satisfies* $dom(\theta')\subseteq\mathscr{V}'$.

Proof. This is proved by structural induction on a derivation tree of type extractor. We here show the case of rule *Eq Ex Comp*: Suppose that

$$\frac{\begin{array}{c}\vdots\\ \rho_0 \notin \mathscr{V} \quad ((E \rhd N : \rho) :: (\rho \rhd M : A) :: \mathscr{J}, \mathscr{V} \cup \{\rho\}) \Downarrow (\mathscr{E}, \mathscr{V}')\end{array}}{((E \rhd M \circ N : A) :: \mathscr{J}, \mathscr{V}) \Downarrow (\mathscr{E}, \mathscr{V}')} \textit{Eq Ex Comp},$$

and there exists a substitution θ such that all typing candidates in $E \rhd M \circ N : A :: \mathscr{J}$ holds by θ and $dom(\theta) \subseteq \mathscr{V}$.

$E^\theta \vdash N : E'$ and $E' \vdash M : A^\theta$ hold since $E^\theta \vdash M \circ N : A^\theta$. We can extend θ to θ' by appending a new correspondence $[\rho \mapsto E']$. Then, $(E \rhd N : \rho) :: (\rho \rhd M : A) :: \mathscr{J}$ holds by θ'. And, $dom(\theta') = \mathscr{V} \cup \{\rho\}$. Therefore, by the induction hypothesis, there exists an extension θ'' of θ' which unifies $\mathscr{E}$ and $dom(\theta'') \subseteq \mathscr{V}'$ **q.e.d.**

Theorem 5.29 [*Completeness of Type Infer and Principal Type*]. *If a term is typed then the type inference algorithm always succeeds and gives a principal type.*

Proof. First, we will prove that the type inference algorithm terminates successfully.

Suppose that a term M is typed with $E \vdash M : A$ *and* $((\rho_0 \rhd M : \alpha) :: \mathit{Empty\ Sequence}, \{\rho_0, \alpha\}) \Downarrow (\mathscr{E}, \mathscr{V})$. Let a substitution θ be $[\rho \mapsto E, \alpha \mapsto A]$. Then by Lemma 5.28, there exists an extension θ' of θ which unifies $\mathscr{E}$:

$$\begin{array}{ccc} ((\rho_0 \rhd M : \alpha) :: \mathit{Empty\ Sequence}, & \{\rho_0, \alpha\}) \Downarrow (\mathscr{E}, \mathscr{V}) \\ \uparrow \text{unify} & \uparrow \text{unify} \\ \theta & \to \quad \theta' \end{array}$$

extend
by Lemma 5.28

By Lemma 5.19, the successive application of the unification transformation terminates in a solved form, hence, the type inference algorithm succeeds.

Next, we will present the principal typing property. Suppose that M is typed as $E \vdash M : A$. Let $\theta = [\rho_0 \mapsto E, \alpha \mapsto A]$, then by Lemma 5.28, we know that θ is extended to θ' which unifies $\mathscr{E}$ and satisfies $dom(\theta') \subseteq \mathscr{V}$. Note the former part of this proof:

$$(\{\rho_0 \rhd M : \alpha\}, \{\rho_0, \alpha\}) \Downarrow (\mathscr{E}, \mathscr{V}) \overset{*}{\Rightarrow} (\mathscr{E}_0, \mathscr{V}_0)$$

existing a solved $\mathscr{E}_0$.

By Lemma 5.20, $\mathscr{E}_0|_{dom(\theta')}$ is more general than θ', i.e. there exists θ'' such that

$$\theta' = (\mathscr{E}_0|_{dom(\theta')})\theta''.$$

Since θ' is an extension of θ,

$$\theta = \theta'|_{dom(\theta)}.$$

From these two, it follows that

$$\theta = (\mathscr{E}_0|_{dom(\theta)})\theta'''.$$

for some substitution θ'''. Therefore,

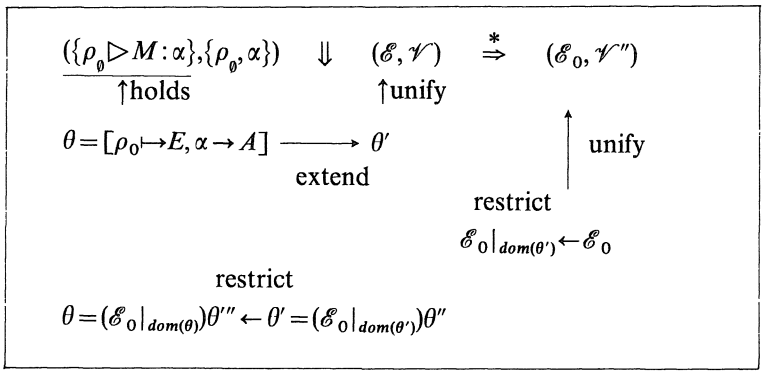
$$E = \rho_\emptyset^\theta = \rho_\emptyset^{(\mathscr{E}_0|_{dom(\theta)})\theta'''} = (\rho_\emptyset^{\mathscr{E}_0})^{\theta'''}$$

and

$$A = \alpha^\theta = \alpha^{(\mathscr{E}_0|_{dom(\theta)})\theta'''} = (\alpha^{\mathscr{E}_0})^{\theta'''}.$$

This shows us that $\rho_\emptyset^{\mathscr{E}_0} \vdash M : \alpha^{\mathscr{E}_0}$ is principal. **q.e.d.**

Last, we present an overview of the above proof as follows:



§5.5. Examples

We present examples of the type inference algorithm introduced in the former section. The examples showing in this section are generated by the

prototype implemented on programming language Prolog.

We start with the simplest example, the typing of term *id*. The following type equation is extracted from typing candidate $\rho_p \triangleright id : \alpha_a$:

$$\rho_p \stackrel{?}{=} \alpha_a.$$

This unification problem is solvable and we get the solution as

$$\alpha_a \mapsto \rho_p.$$

As a result, we obtain typing $\rho_p \vdash id : \rho_p$.

Next, we try the more complex case of term $f(x \circ env)(y \circ env)$, which corresponds to Scheme's program (***f*(*eval* 'x *env*)(*eval* 'y *env*)**). The type equations extracted from typing candidate $\rho_p \triangleright f(x \circ env)(y \circ env). : \alpha_a$ is

$$\rho_p \stackrel{?}{=} \{f : \alpha_2 \to \alpha_1 \to \alpha_a\}\rho_9$$

$$\rho_p \stackrel{?}{=} \{env : \rho_3\}\rho_8$$

$$\rho_3 \stackrel{?}{=} \{x : \alpha_2\}\rho_7$$

$$\rho_p \stackrel{?}{=} \{env : \rho_4\}\rho_6$$

$$\rho_4 \stackrel{?}{=} \{y : \alpha_1\}\rho_5.$$

This set of type equations is solved by the following unifier:

$$\rho_5 \mapsto \{x : \alpha_2\}\rho_{12}$$
$$\rho_p \mapsto \{f : \alpha_2 \to \alpha_1 \to \alpha_a\}\{env : \{x : \alpha_2\}\{y : \alpha_1\}\rho_{12}\}\rho_{11}$$
$$\rho_9 \mapsto \{env : \{x : \alpha_2\}\{y : \alpha_1\}\rho_{12}\}\rho_{11}$$
$$\rho_8 \mapsto \{f : \alpha_2 \to \alpha_1 \to \alpha_a\}\rho_{11}$$
$$\rho_3 \mapsto \{x : \alpha_2\}\{y : \alpha_1\}\rho_{12}$$
$$\rho_{10} \mapsto \rho_{11}$$
$$\rho_6 \mapsto \{f : \alpha_2 \to \alpha_1 \to \alpha_a\}\rho_{11}$$
$$\rho_4 \mapsto \{x : \alpha_2\}\{y : \alpha_1\}\rho_{12}$$
$$\rho_7 \mapsto \{y : \alpha_1\}\rho_{12}$$

Consequently, we get the principal typing

$$\{f:\alpha_2 \to \alpha_1 \to \alpha_a\}\{env:\{x:\alpha_2\}\{y:\alpha_1\}\rho_{12}\}\rho_{11} \vdash f(x \circ env)(y \circ env):\alpha_a$$

Last, the following example is a variant of the Scheme program in the introduction.

$(\lambda l\text{-}inf\text{-}space.\lambda l1space.\lambda l2space.id)$
$((\lambda norm.id)(\lambda x.max(abs(car\ x))(abs(cdr\ x))))$
$((\lambda norm.id)(\lambda x.plus(abs(car\ x))(abs(car\ x))))$
$((\lambda norm.id)(\lambda x.sqrt(plus(expt(car\ x)two)(expt(cdr\ x)two))))$

Its corresponding Scheme's program:

```
(let ((l-inf-space
       (let ((norm (lambda (x)(max (abs(car x))(abs(cdr x))))))
         (the-environment)))
      (l1space
       (let ((norm (lambda (x)(+ (abs(car x)) (abs(cdr x))))))
         (the-environment)))
      (l2space
       (let ((norm
              (lambda (x)(sqrt (+ (expt (car x) 2)(expt (cdr x) 2))))))
         (the-environment))))
  (the-environment))
```

We can use here primitives (e.g. *max* or *cdr*) by replacing them with variables. Then, we obtain the typing in the followings.

$\{max:\alpha_{38} \to \alpha_{38} \to \alpha_{14}\}\{abs:\alpha_{44} \to \alpha_{38}\}$
$\{car:\alpha_{45} \to \alpha_{44}\}\{cdr:\alpha_{45} \to \alpha_{44}\}$
$\{plus:\alpha_{38} \to \alpha_{38} \to \alpha_{37}\}\{sqrt:\alpha_{37} \to \alpha_{36}\}\{expt:\alpha_{44} \to \alpha_{43} \to \alpha_{38}\}$
$\{two:\alpha_{43}\}\rho_{93}$
$\vdash$
$(\lambda l\text{-}inf\text{-}space.\lambda l1space.\lambda l2space.id)$

$((\lambda norm.id)(\lambda x.max(abs(car\ x))(abs(cdr\ x))))$

$((\lambda norm.id)(\lambda x.plus(abs(car\ x))(abs(cdr\ x))))$

$((\lambda norm.id)(\lambda x.sqrt(plus(expt(car\ x)two)(expt(cdr\ x)two))))$

$: \{l2space: \{norm: \alpha_{45} \to \alpha_{36}\}$

$\{max: \alpha_{38} \to \alpha_{38} \to \alpha_{14}\}\{abs: \alpha_{44} \to \alpha_{38}\}$

$\{car: \alpha_{45} \to \alpha_{44}\}\{cdr: \alpha_{45} \to \alpha_{44}\}$

$\{plus: \alpha_{38} \to \alpha_{38} \to \alpha_{37}\}\{sqrt: \alpha_{37} \to \alpha_{36}\}\{expt: \alpha_{44} \to \alpha_{43} \to \alpha_{38}\}$

$\{two: \alpha_{43}\}$

$\rho_{93}\}$

$\{l1space: \{norm: \alpha_{45} \to \alpha_{37}\}$

$\{max: \alpha_{38} \to \alpha_{38} \to \alpha_{14}\}\{abs: \alpha_{44} \to \alpha_{38}\}$

$\{car: \alpha_{45} \to \alpha_{44}\}\{cdr: \alpha_{45} \to \alpha_{44}\}$

$\{plus: \alpha_{38} \to \alpha_{38} \to \alpha_{37}\}\{sqrt: \alpha_{37} \to \alpha_{36}\}\{expt: \alpha_{44} \to \alpha_{43} \to \alpha_{38}\}$

$\{two: \alpha_{43}\}$

$\rho_{93}\}$

$\{l\text{-}inf\text{-}space: \{norm: \alpha_{45} \to \alpha_{14}\}$

$\{max: \alpha_{38} \to \alpha_{38} \to \alpha_{14}\}\{abs: \alpha_{44} \to \alpha_{38}\}$

$\{car: \alpha_{45} \to \alpha_{44}\}\{cdr: \alpha_{45} \to \alpha_{44}\}$

$\{plus: \alpha_{38} \to \alpha_{38} \to \alpha_{37}\}\{sqrt: \alpha_{37} \to \alpha_{36}\}\{expt: \alpha_{44} \to \alpha_{43} \to \alpha_{38}\}$

$\{two: \alpha_{43}\}\rho_{93}\}$

$\{max: \alpha_{38} \to \alpha_{38} \to \alpha_{14}\}\{abs: \alpha_{44} \to \alpha_{38}\}$

$\{car: \alpha_{45} \to \alpha_{44}\}\{cdr: \alpha_{45} \to \alpha_{44}\}$

$\{plus: \alpha_{38} \to \alpha_{38} \to \alpha_{37}\}\{sqrt: \alpha_{37} \to \alpha_{36}\}\{expt: \alpha_{44} \to \alpha_{43} \to \alpha_{38}\}$

If we assume that

$$max: int \to int \to int$$

$$abs: int \to int$$

$$car: int\ List \to int$$

$$cdr: int\ List \to int$$

$$plus: int \to int \to int$$

$$sqrt : int \to real$$
$$expt : int \to int \to int$$
$$two : int,$$

then the former typing can be read as

$$\{max : int \to int \to int\}\{abs : int \to int\}$$
$$\{car : int\ List \to int\}\{cdr : int\ List \to int\}$$
$$\{plus : int \to int \to int\}\{sqrt : int \to real\}\{expt : int \to int \to int\}$$
$$\{two : int\}\rho_{93}$$
$$\vdash$$
$$(\lambda l\text{-}inf\text{-}space.\lambda l1space.\lambda l2space.id)$$
$$((\lambda norm.id)(\lambda x.max(abs(car\ x))(abs(cdr\ x))))$$
$$((\lambda norm.id)(\lambda x.plus(abs(car\ x))(abs(cdr\ x))))$$
$$((\lambda norm.id)(\lambda x.sqrt(plus(expt(car\ x)two)(expt(cdr\ x)two))))$$
$$: \{l2space : \{norm : int\ List \to real\}$$
$$\{max : int \to int \to int\}\{abs : int \to int\}$$
$$\{car : int\ List \to int\}\{cdr : int\ List \to int\}$$
$$\{plus : int \to int \to int\}\{sqrt : int \to real\}\{expt : int \to int \to int\}$$
$$\{two : int\}$$
$$\rho_{93}\}$$
$$\{l1space : \{norm : int\ List \to int\}$$
$$\{max : int \to int \to int\}\{abs : int \to int\}$$
$$\{car : int\ List \to int\}\{cdr : int\ List \to int\}$$
$$\{plus : int \to int \to int\}\{sqrt : int \to real\}\{expt : int \to int \to int\}$$
$$\{two : int\}$$
$$\rho_{93}\}$$
$$\{l\text{-}inf\text{-}space : \{norm : int\ List \to int\}$$
$$\{max : int \to int \to int\}\{abs : int \to int\}$$
$$\{car : int\ List \to int\}\{cdr : int\ List \to int\}$$
$$\{plus : int \to int \to int\}\{sqrt : int \to real\}\{expt : int \to int \to int\}$$

$$\{two : int\}\rho_{93}\}$$
$$\{max : int \to int \to int\}\{abs : int \to int\}$$
$$\{car : int\ List \to int\}\{cdr : int\ List \to int\}$$
$$\{plus : int \to int \to int\}\{sqrt : int \to real\}\{expt : int \to int \to int\}$$
$$\{two : int\}\rho_{93}$$

§6. Conclusion

We proposed a simply typed lambda calculus $\lambda^{\rightarrow}_{env}$ with first-class environments, where we adopt the idea of *explicit substitution*: the formal treatment of environments with substitutions. The integration of the terms and the environments gives us first-class environments. We then showed several fundamental properties of this calculus: subject reduction property, confluence, and strong normalizability with respect to weak reduction. Moreover, we developed a type inference algorithm which gives us principal typing.

§7. Related Works and Future Studies

§7.1. $\lambda\sigma$-calculus and Categorical Combinator

$\lambda^{\rightarrow}_{env}$ and $\lambda\sigma$-calculus seem to be similar to each other, since $\lambda\sigma$-calculus is the origin of $\lambda^{\rightarrow}_{env}$. On the other hand, the integration of terms and substitutions distinguishes $\lambda^{\rightarrow}_{env}$ from $\lambda\sigma$-calculus. Although we obtain first-class environments due to such an integration, alternatively, we lose the nameless transformation: it is possible to transform a term with names to a nameless one in lambda calculus and in $\lambda\sigma$-calculus even if the term is untyped, but in our calculus, it is impossible. Consider a term $\lambda env.(x \circ env)$: An environment where variable x is bound, is passes to this term as an argument and then the binded value to the variable x in the passed environment, is returned. The *de Bruijn index* is the "address" of the binding of x in the environment given as the argument. We can know what bindings are located in the environment, *only at execution time*, and therefore, it is impossible to know statically the de Bruijn index of x. Our $\lambda^{\rightarrow}_{env}$ is *neither superior nor inferior* to $\lambda\sigma$-calculus: these are only different.

In spite of such a difference, some properties in $\lambda\sigma$-calculus also hold in $\lambda^{\rightarrow}_{env}$. Readers can find the similarity of the confluence proof between $\lambda^{\rightarrow}_{env}$ and $\lambda\sigma$-calculus, if they read [6]. The reason may be that $\lambda\sigma$-calculus has its origin in categorical combinatorial logic [5], where environments and terms

are uniformly treated and as a result, there implicitly exists first-class environments. Therefore, some properties of categorical combinatorial logic still remain in $\lambda\sigma$-calculus and also in $\lambda^{\rightarrow}_{env}$.

§7.2. Record Calculi

Lambda calculi with records, so-called *record calculi*, is studied by many researchers in order to enable us object-oriented programming in functional programming paradigm ([15], [10], [14], [13], [11]). Records and environments resemble each other in their structure: both of them are assignments which associate each name with a value. However, we should note the difference between them. Names used in records, called *labels*, are only for records. In contrast, names in environments, i.e. variables, do not only occur in environments but also in lambda abstraction.

The proof of strong normalizability gives a view of the relation between them. We proved the strong normalizability by interpretation of $\lambda^{\rightarrow}_{env}$ to λ_{record}. Variables are *not* translated to variables but to labels of records, and environments to records. Primitive *id* receives a record which is an interpretation of an environment, and returns it without any change:

$$[\![id]\!]^*(Env) = Env.$$

In the computation of primitive $M \circ N$, term N is first computed and a record is obtained as a result, then M is evaluated under the record which denoted an environment:

$$[\![M \circ N]\!]^*(Env) = [\![M]\!]^*([\![N]\!]^*(Env)).$$

Therefore, we can say that

- records are *reified* data of environments, and
- primitives *id* and $(-)\circ(-)$ accomplish the *reification* and *reflection* of environments, respectively.

Many fruitful results in record calculus will be applicable to our calculus.

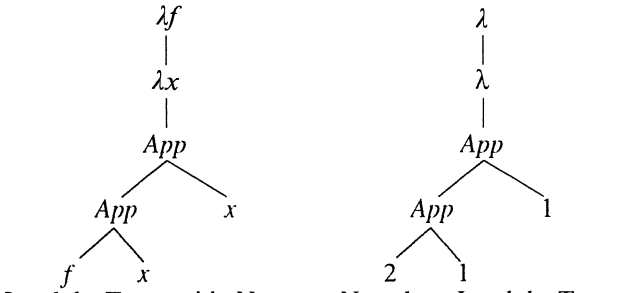
The unification on environment type is also a topic related to this correspondence. It is shown in the former section that the unification algorithm developed in record calculi play an important role also in our type inference algorithm. And, to tell the truth, side conditions in the definition of environment

types are appended in order to make the unification algorithm work. The former version of $\lambda_{env}^{\rightarrow}$ does not have any environment type variable. These notions are borrowed from Wand-Jategaonkar-Mitchell's record calculus. Although we imposed a strong restriction on environment types, we think that this restriction will be indispensable to ensure the existence of the most general unifier since the mutual distinction of field's labels is very essential in Wand-Jategaonkar-Mitchell's unification. Therefore, if we had a unification algorithm without this restriction on labels, we would obtain the type inference algorithm without such a restriction.

§7.3. De Bruijn Indexing

We have already made a comment on the nameless transformation, i.e. de Bruijn indexing, in the previous section. In this section, we would like to continue discussing de Bruijn indexing of $\lambda_{env}^{\rightarrow}$.

The method of de Bruijn indexing is convenient for formalizing lambda calculus on a proof checker because we can avoid the treatment of α-equivalence by using this indexing. In Categorical Abstract Machine (CAM), this method plays an important role in compiling variable reference operation. In the usual lambda calculi (and $\lambda\sigma$-calculus), we can transform terms with names to nameless ones, even if they are untyped. (Note that all untyped terms are "typable" under universal typing: all terms have unique one type U and $U \cong U \rightarrow U$.) As widely known, a type inference tree includes information on de Bruijn indexing (cf. Section 3 in [9]):



Lambda Term with Names: Nameless Lambda Term (with de Bruijn index):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\alpha,f:\alpha\to\alpha\to\alpha\vdash f:\alpha\to\alpha\to\alpha}2 \quad \dfrac{}{x:\alpha,f:\alpha\to\alpha\to\alpha\vdash x:\alpha}1}{x:\alpha,f:\alpha\to\alpha\to\alpha\vdash fx:\alpha\to\alpha}App \quad \dfrac{}{x:\alpha,f:\alpha\to\alpha\vdash x:\alpha}1}{x:\alpha,f:\alpha\to\alpha\to\alpha\vdash fxx:\alpha}App}{f:\alpha\to\alpha\to\alpha\vdash\lambda x.fxx:\alpha\to\alpha}\lambda}{\vdash\lambda f.\lambda x.fxx:(\alpha\to\alpha\to\alpha)\to\alpha\to\alpha}\lambda$$

Type Inference Tree

This relation holds if we suppose that variable-type pairs in the left side of $\vdash$ cannot not be permuted each other. However, variable-type pairs in environment types in $\lambda^{\to}_{env}$ are commutative, therefore, we cannot extract a nameless term from a type inference tree as usual: consider the following example

$$f(g(\lambda x.\lambda y.id))(g(\lambda y.\lambda x.id)).$$

Subterm $(\lambda x.\lambda y.id)$ is typed as $\alpha\to\beta\to\underline{\{y:\beta\}\{x:\alpha\}\rho}$ and subterm $(\lambda y.\lambda x.id)$ is typed as $\beta'\to\alpha'\to\underline{\{x:\alpha'\}\{y:\beta'\}\rho}$. Without the commutativity between $y:\beta$ and $x:\alpha$ or between $x:\alpha'$ and $y:\beta'$, we cannot unify these two types and therefore, this term is not typable.

Our problem seems to be related to the record compilation method [13] from the viewpoint in Section 7.2.

Acknowledgements

The author wishes to thank his friend, his colleagues, his parents, and his co-supervisors: Prof. Satoru Takasu and Prof. Masami Hagiya. Thanks are due also to Prof. Pierre-Louis Curien, Prof. Atsushi Ohori, and the referee for discussions, comments, and pointing out of errors in the draft.

References

[1] *MIT Scheme Reference Manual*, MIT.
[2] Abelson, H. and Sussman, G.J., *Structure and Interpretation of Computer Programs*, The MIT Press, 1985.
[3] Curien, P-L., An abstract framework for environment machines, *Theoretical Computer Science*, **82** (1991), 389–402.
[4] ———, Categorical combinators, *Information and Control*, **69** (1986), 188–254.
[5] ———, *Categorical combinators, sequential algorithms, and functional programming*, Birkhäuser, second edition, 1993.

[6] Curien, P-L., Hardin, T. and Lévy, J-J., *Confluence Properties of Weak and Strong Calculi of Explicit Substitutions*, Rapports de Recherche 1617, INRIA, February 1992.

[7] Girard, J-Y., Taylor, P. and Lafont, Y., *Proofs and Types*, *Cambridge Tracts in Compu. Sci.*, **7**, Cambridge University Press, 1989.

[8] Griffin, T.G., A formulae-as-types notion of control, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

[9] Gunter, C.A., *Semantics of programming languages: structures and techniques*, The MIT Press, 1992.

[10] Jategaonkar, L.A. and Mitchell, J.C., ML with extended pattern matching and subtypes, *Proceedings of the* 1988 *Conference on LISP and Functional Programming*, (1988), 198–211.

[11] ———, Type inference with extended pattern matching and subtypes, *Fundamenta Informaticae*, **19** (1993), 127–166.

[12] Abadi, M., Cardelli, L., Curien, P-L. and Lévy, J-J., Explicit substitutions, *Proceedings of the Seventeenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Francisco, California, January 1990.

[13] Ohori, A., A compilation method for ML-style polymorphic record calculi, *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, (1992), 154–165.

[14] Rémy, D., Typechecking records and variants in a natural extention of ML, *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, (1989), 60–67.

[15] Wand, M., Complete type inference for simple objects, *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, (1987), 37–44.