A Simple Theory of Expressions, Judgments and Derivations

Masahiko Sato

Graduate School of Informatics, Kyoto University

Abstract. We propose a simple theory of expressions which is intended to be used as a foundational syntactic structure for the Natural Framework (NF). We define expression formally and give a simple proof of the decidability of α-equivalence. We use this new theory of expressions to define judgments and derivations formally, and we give concrete examples of derivation games to show a flavor of NF.

1 Introduction

Abstraction and *instantiation* are two basic operations on expressions. Although the intuitive meanings of an abstract like $B \equiv (x)A(x)$ and its instance $A(t)$ (which is obtained from $A(x)$ by substituting t for x) are clear, the actual process of computing the expression $A(t)$ from B is very subtle as it sometimes requires the renaming of bound variables in B to avoid the capturing of free variables in t during the process of instantiation.

In this paper, we will present a simple theory of expressions in which expressions form a structure equipped with the operations of abstraction and instantiation. The novelty of the structure is that the instantiation operation can be carried out without renaming bound variables. Because of this property, we can define α-equivalence of expressions without relying on renaming of variables, and the proof of the decidability of α-equivalence becomes very simple. We remark that, as pointed out by Vestergaard [7], to give a rigorous proof of it is a delicate problem. We will give an informal account of the theory of expressions in 2 and give a formal presentation of the theory in 3.

We will then use the theory of expressions to reformulate the theory of judgments and derivations introduced in [5]. This theory forms the basis of the Natural Framework (NF), and in NF we can define various mathematical systems in a uniform and convenient way using *derivation games*. In 4, we will define the concepts of judgments, derivations and derivations, and will show that these concepts can be rigorously defined by using the higher-order abstract syntax provided by the simple theory of expressions.

2 Informal theory of expressions

In this section we present our theory of expression in an informal way. The presentation is informal only because we will take the notion of α-equivalence

for granted. As we usually take this notion for granted, the reader should be able to grasp the essence of the theory by reading this section.

2.1 Expressions

For each n ($n = 0, 1, 2, \cdots$), we assume a countably infinite set V_n of *variables* (x, y, z). For each n ($n = 0, 1, 2, \cdots$), we assume a countably infinite set C_n of *constants* (c, d). We assume that all these sets are mutually disjoint, so that given any variable x (or constant c) we can uniquely determine a natural number such that $x \in \mathsf{V}_n$ ($c \in \mathsf{C}_n$, resp.).

We will say that a variable (constant) is of *arity* n if it is in V_n (C_n, resp.). A variable is *higher-order* if its arity is positive and it is *first-order* if its arity is 0, and similarly for a constant.

We define expressions as follows, where $e : \mathsf{exp}$ will mean that e is an *expression*.

We identify α-equivalent expressions.

$$\frac{x \in \mathsf{V}_n \quad a_1 : \mathsf{exp} \quad \cdots \quad a_n : \mathsf{exp}}{x[a_1, \ldots, a_n] : \mathsf{exp}}\ \mathsf{var}$$

$$\frac{c \in \mathsf{C}_n \quad a_1 : \mathsf{exp} \quad \cdots \quad a_n : \mathsf{exp}}{c[a_1, \ldots, a_n] : \mathsf{exp}}\ \mathsf{const}$$

$$\frac{x \in \mathsf{V}_n \quad a : \mathsf{exp}}{(x)a : \mathsf{exp}}\ \mathsf{abs}$$

We will understand that $x[a_1, \ldots, a_n]$ ($c[a_1, \ldots, a_n]$) stands for x (c, resp.) when $n = 0$. We will write $(x_1, \cdots, x_n)a$ for $(x_1) \cdots (x_n)a$, and when $n = 0$, this stands for a. We will also write $(x)\,[a]$ for $(x)a$ when we wish to emphasize that x is the binding variable and its scope is a.

Note that a variable standing by itself is not an expression if its arity is positive. For each variable $x \in \mathsf{V}_n$, we associate an expression $x^* : \mathsf{exp}$ as follows.

1. $x^* :\equiv x$ if $n = 0$.
2. $x^* :\equiv (x_1, \ldots, x_n)x[x_1, \ldots, x_n]$ if $n > 0$, where $x_1, \ldots, x_n$ are all of arity 0.

We will sometimes simply write x as a short hand for x^*.

2.2 Environments

We define environments which are used to instantiate abstract expressions and also to define substitution. Let x be an n-ary variable. We say that an expression e is *substitutable for* x if e is of the form $(x_1, \ldots, x_n)a$ where $x_1, \ldots, x_n$ are all 0-ary variables. So, any expression is substitutable for a 0-ary variable, but, only expressions of the form $(x, y)e$ (x, y are 0-ary) are substitutable for 2-ary variables.

If x is a variable of arity n and e is substitutable for x, then $x = e$ is a *definition*, and a set of definitions $\rho = \{x_1 = e_1, \ldots, x_k = e_k\}$ is an *environment* if $x_1, \ldots, x_k$ are distinct variables, and its *domain* $|\rho|$ is $\{x_1, \ldots, x_k\}$.

2.3 Instantiation

Given an expression e and an environment ρ, we define an expression $[e]_\rho$ as follows. We choose fresh local variables as necessary.

1. $[x]_\rho :\equiv e$ if x is of arity 0 and $x = e \in \rho$.
2. $[x[a_1, \ldots, a_n]]_\rho :\equiv [e]_{\{x_1=[a_1]_\rho, \ldots, x_n=[a_n]_\rho\}}$ if $n > 0$ and $x = (x_1, \ldots, x_n)e \in \rho$.
3. $[x[a_1, \ldots, a_n]]_\rho :\equiv x[[a_1]_\rho, \ldots, [a_n]_\rho]$ if $x \notin |\rho|$.
4. $[c[a_1, \ldots, a_n]]_\rho :\equiv c[[a_1]_\rho, \ldots, [a_n]_\rho]$.
5. $[(x)\,[a]]_\rho :\equiv (x)\,[[a]_\rho]$.

We can check the well-definedness of the above definition as follows.

An environment ρ is *first-order* if all the variables in $|\rho|$ are first-order, and it is *higher-order* if $|\rho|$ contains at least one higher-order variable. If the given environment is first-order, then the above definition is an ordinary inductive definition. Now, since we know that the above definition is well-defined for first-order environments, we can carry out the above definition for higher-order environments.

It is essential to distinguish first-order variables and higher-order variables. Without the distinction, evaluation of expressions may fail to terminate as can be seen by the following example.

$$[x[x]]_{\{x=(y)y[y]\}} \equiv [y[y]]_{\{y=[x]_{\{x=(y)y[y]\}}\}} \equiv [y[y]]_{\{y=(y)y[y]\}} \equiv \cdots$$

However, since we do have the distinction of first-order and higher-order variables, the above computation is not possible. Namely, by our definition of environment, y must be of arity 0, since y occurs as a binder in $(y)y[y]$. But y must be also of arity 1 because of the first occurrence of y in $y[y]$. This is a contradiction.

3 Formal theory of expressions

We now present our theory of expressions formally. To do this, we first extend the notion of variables as follows.

3.1 Variable references

If $x \in \mathsf{V}_n$ and $k \in \mathsf{N}$, where N is the set of natural numbers, then $\sharp^k x$ is a *variable reference* and k is called the *level* of the variable reference. In particular, if $k = 0$, then $\sharp^k x$ is the variable x. We use the letter r as a meta variable ranging over variable references.

For each arity n we choose and fix an n-ary variable and write it as v_n.

3.2 Expressions

We define expressions inductively as follows. If the judgment e:exp can be derived by the following rules, e is said to be a *expression*.

$$\frac{k \in \mathsf{N} \quad x \in \mathsf{V}_n \quad a_1 : \mathsf{exp} \quad \cdots \quad a_n : \mathsf{exp}}{\sharp^k x[a_1, \ldots, a_n] : \mathsf{exp}}\ \mathsf{varref}$$

$$\frac{c \in \mathsf{C}_n \quad a_1 : \mathsf{exp} \quad \cdots \quad a_n : \mathsf{exp}}{c[a_1, \ldots, a_n] : \mathsf{exp}}\ \mathsf{const}$$

$$\frac{x \in \mathsf{V}_n \quad a : \mathsf{exp}}{(x)a : \mathsf{exp}}\ \mathsf{abs}$$

For a variable reference $r \equiv \sharp^k x$ $(x \in \mathsf{V}_n)$ we define an expression r^* as follows.

$$(\sharp^k x)^* :\equiv \begin{cases} \sharp^k x & \text{if } n = 0, \\ (x_1, \ldots, x_n)\sharp^k x[x_1, \ldots, x_n] & \text{if } n > 0, \text{ where } x_i \equiv \mathsf{v}_0\ (1 \leq i \leq n). \end{cases}$$

For each expression a we assign its *size*, $|a|$, as follows.

1. $|\sharp^k x[a_1, \ldots, a_n]| :\equiv |a_1| + \cdots + |a_n| + 1$.
2. $|c[a_1, \ldots, a_n]| :\equiv |a_1| + \cdots + |a_n| + 1$.
3. $|(x)a| :\equiv |a| + 1$.

For each variable reference r and expression a we associate a set $\mathrm{occ}(r, a)$ of *free occurrences of* r *in* a as follows. An occurrence is represented by a string in N^*. If $S \subseteq \mathsf{N}^*$ and $n \in \mathsf{N}$ we put $n \cdot S :\equiv \{n \cdot \sigma \mid \sigma \in S\}$, where $n \cdot \sigma$ denotes the concatenation of the natural number n (regarded as a character) with the string σ.

1. $\mathrm{occ}(\sharp^k x, \sharp^\ell y[a_1, \ldots, a_n])$
$$:\equiv \begin{cases} \{0\} \cup 1 \cdot \mathrm{occ}(\sharp^k x, a_1) \cup \cdots \cup n \cdot \mathrm{occ}(\sharp^k x, a_n) & \text{if } k = \ell \text{ and } x \equiv y, \\ 1 \cdot \mathrm{occ}(\sharp^k x, a_1) \cup \cdots \cup n \cdot \mathrm{occ}(\sharp^k x, a_n) & \text{otherwise.} \end{cases}$$
2. $\mathrm{occ}(\sharp^k x, c[a_1, \ldots, a_n]) :\equiv 1 \cdot \mathrm{occ}(\sharp^k x, a_1) \cup \cdots \cup n \cdot \mathrm{occ}(\sharp^k x, a_n)$.
3. $\mathrm{occ}(\sharp^k x, (y)a) :\equiv \begin{cases} \mathrm{occ}(\sharp^{k+1} x, a) & \text{if } x \equiv y, \\ \mathrm{occ}(\sharp^k x, a) & \text{if } x \not\equiv y. \end{cases}$

For example, assuming that c is a binary constant and d is a 4-ary constant, we have

$$\begin{aligned} \mathrm{occ}(x, c[x, (x)d[x, \sharp^1 x, \sharp^2 x, y]]) &= \{1, 2 \cdot 2\}, \\ \mathrm{occ}(\sharp^1 x, c[x, (x)d[x, \sharp^1 x, \sharp^2 x, y]]) &= \{2 \cdot 3\}. \\ \mathrm{occ}(\sharp^2 x, c[x, (x)d[x, \sharp^1 x, \sharp^2 x, y]]) &= \emptyset \text{ and} \\ \mathrm{occ}(y, c[x, (x)d[x, \sharp^1 x, \sharp^2 x, y]]) &= \{2 \cdot 4\}. \end{aligned}$$

We put $\mathrm{FV}(a) :\equiv \{r \mid \mathrm{occ}(r, a) \neq \emptyset\}$ and call it the set of *free variable occurrences* in a.

An expression a is *closed* if $\mathrm{FV}(a) = \emptyset$. For example, $(x)\sharp^2 x$ is an expression but is not closed since $\mathrm{FV}((x)\sharp^2 x) = \{\sharp^1 x\}$, and $(x)x$ is a closed expression since $\mathrm{FV}((x)x) = \emptyset$.

3.3 Environments

We modify the notion of environments and define it as follows. Firstly, a *definition* is defined in the same way as before. Namely, a definition is an expression of the form $x = (x_1, \ldots, x_n)a$ where $x \in \mathsf{V}_n$ and $x_1, \ldots, x_n$ are all arity 0 variables. Secondly, we will call any variable x a *declaration*. Then an *environment* is a list of definitions and declarations of the form:

$$\rho = (x_1 = a_1, \ldots, x_k = a_k, y_1, \ldots, y_\ell) \quad (k, \ell \in \mathsf{N}).$$

If $k = l = 0$, then $\rho = ()$ is called the *empty environment*. An environment is said to be *first-order* if x is of arity 0 for any definition $x = a$ in the environment, and it is said to be *higher-order* otherwise.

We will also modify the process of instantiation, and for that we first define the process of *pushing* an expression *through* a variable reference. Namely, for any expression a and any variable reference $\sharp^k z$, we define an expression $a \uparrow \sharp^k z$ as follows and call it the result of *pushing* a *through* $\sharp^k z$.

1. $\sharp^m x[a_1, \ldots, a_n] \uparrow \sharp^k z :\equiv \begin{cases} \sharp^m x[a_1 \uparrow \sharp^k z, \ldots, a_n \uparrow \sharp^k z] & \text{if } m < k \text{ and } x \equiv z, \\ \sharp^{m+1} x[a_1 \uparrow \sharp^k z, \ldots, a_n \uparrow \sharp^k z] & \text{if } m \geq k \text{ and } x \equiv z, \\ \sharp^m x[a_1 \uparrow \sharp^k z, \ldots, a_n \uparrow \sharp^k z] & \text{if } x \not\equiv z. \end{cases}$
2. $c[a_1, \ldots, a_n] \uparrow \sharp^k z :\equiv c[a_1 \uparrow \sharp^k z, \ldots, a_n \uparrow \sharp^k z]$.
3. $(x)\,[a] \uparrow \sharp^k z :\equiv \begin{cases} (x)\,[a \uparrow \sharp^{k+1} z] & \text{if } x \equiv z, \\ (x)\,[a \uparrow \sharp^k z] & \text{if } x \not\equiv z. \end{cases}$

Next, given an environment ρ and a variable reference $\sharp^k x$ we define $\rho(\sharp^k x)$ as follows and call it the *value* of $\sharp^k x$ in ρ.

1. $\rho(\sharp^k x) :\equiv \sharp^k x$, if $\rho = ()$.
2. $(\rho, z)(\sharp^k x) :\equiv \begin{cases} \sharp^k x & \text{if } k = 0 \text{ and } x \equiv z, \\ \rho(\sharp^{k-1} x) \uparrow z & \text{if } k > 0 \text{ and } x \equiv z, \\ \rho(\sharp^k x) \uparrow z & \text{if } x \not\equiv z. \end{cases}$
3. $(\rho, z = a)(\sharp^k x) :\equiv \begin{cases} a & \text{if } k = 0 \text{ and } x \equiv z, \\ \rho(\sharp^{k-1} x) & \text{if } k > 0 \text{ and } x \equiv z, \\ \rho(\sharp^k x) & \text{if } x \not\equiv z. \end{cases}$

We note that the actual value is computed by (1) the first item of the above definition, or by (2) the first subcase of the second item, or by (3) the first subcase of the third item. If it is computed by (1), then $\sharp^k x$ is said to be *free* in ρ, and if computed by (2), then it is said to be *declared* in ρ, and if computed by (3), then it is said to be *defined* in ρ,

3.4 Instantiation and substitution

Now, given an expression a and an environment ρ, we can define the *instantiation* $[a]_\rho$ of a in the environment ρ as follows. $[a]_\rho$ is also called the ρ-*instance* of a.

1. $[\sharp^k x[a_1,\ldots,a_n]]_\rho$
$:\equiv \begin{cases} [b]_{(x_1=[a_1]_\rho,\ldots,x_n=[a_n]_\rho)} & \text{if } \sharp^k x \text{ is defined in } \rho \text{ and } \rho(\sharp^k x) \equiv (x_1,\ldots,x_n)b, \\ \sharp^m x[[a_1]_\rho,\ldots,[a_n]_\rho] & \text{if } \sharp^k x \text{ is free or declared in } \rho \text{ and } \rho(\sharp^k x) \equiv \sharp^m x. \end{cases}$
2. $[c[a_1,\ldots,a_n]]_\rho :\equiv c[[a_1]_\rho,\ldots,[a_n]_\rho]$.
3. $[(x)\,[b]]_\rho :\equiv (x)\,[[b]_{(\rho,x)}]$.

It should be noted that because of item 3, we have to extend the notion of environment by allowing declarations to be its part, and also that the definition in item 3 does not rely on the notion of α-equivalence. The above definition can be seen to be well-defined, by first considering the case where the environment ρ is first-order, and then considering the case where ρ is higher-order. It can also be seen that $[a]_\rho$ is an expression for any environment ρ.

If x and y are variables of the same arity, then we can easily see that for any expression a, $|[a]_{(x=y^*)}| = |a|$. If b is an expression and a is an expression subsitutable for x, then $[b]_{(x=a)}$ is called the result of *substituting a for x in b*.

3.5 α-equivalence

In this subsection, we define the notion of *α-equivalence* and show that it is a decidable relation with the expected property that α-equivalence is preserved by substitution.

The judgment $a \equiv_\alpha b$ which is characterized by the following rules will be read 'a is α-equivalent to b'. Two expressions a and b are *α-equivalent* if and only if the judgment $a \equiv_\alpha b$ can be derived by the following rules.

$$\frac{k \in \mathsf{N} \quad x \in \mathsf{V}_n \quad a_1 \equiv_\alpha b_1 \quad \cdots \quad a_n \equiv_\alpha b_n}{\sharp^k x[a_1,\ldots,a_n] \equiv_\alpha \sharp^k x[b_1,\ldots,b_n]}\ \mathsf{varref}$$

$$\frac{c \in \mathsf{C}_n \quad a_1 \equiv_\alpha b_1 \quad \cdots \quad a_n \equiv_\alpha b_n}{c[a_1,\ldots,a_n] \equiv_\alpha c[b_1,\ldots,b_n]}\ \mathsf{const}$$

$$\frac{x,y \in \mathsf{V}_n \quad \mathrm{occ}(x,a) = \mathrm{occ}(y,b) \quad [a]_{(x=\mathsf{v}_n{}^*)} \equiv_\alpha [b]_{(y=\mathsf{v}_n{}^*)}}{(x)a \equiv_\alpha (y)b}\ \mathsf{abs}$$

We can show that the α-equivalence is indeed an equivalence relation in a straight forward way by induction on the size of expressions. The inductive argument works for the abs-rule case since $|[a]_{(x=\mathsf{v}_n{}^*)}| = |a|$. We can similarly show that the α-equivalence relation is a decidable relation.

We give two simple examples of derivations assuming that x and y are distinct variables of arity 0. In the second example below, we note that $[(x)\sharp^1 x]_{(x=\mathsf{v}_0{}^*)} \equiv x$.

$$\frac{x,y \in \mathsf{V}_0 \quad \mathrm{occ}(x,x) = \mathrm{occ}(y,y) = \{0\} \qquad \dfrac{0 \in \mathsf{N} \quad \mathsf{v}_0 \in \mathsf{V}_0}{\mathsf{v}_0 \equiv_\alpha \mathsf{v}_0}\ \mathsf{var}}{(x)x \equiv_\alpha (y)y}\ \mathsf{abs}$$

$$\frac{x, y \in \mathsf{V}_0 \quad \mathrm{occ}(x, \sharp^1 x) = \mathrm{occ}(y, x) = \emptyset}{(x)\sharp^1 x \equiv_\alpha (y)x}\ \mathsf{abs} \qquad \frac{0 \in \mathsf{N} \quad x \in \mathsf{V}_0}{x \equiv_\alpha x}\ \mathsf{var}$$

We can also prove the basic theorem which establishes that α-equivalence is preserved by substitution of α-equivalent expressions.

Theorem 1. *If $a \equiv_\alpha b$ and $a' \equiv_\alpha b'$, then $[a]_{(z=a')} \equiv_\alpha [b]_{(z=b')}$.*

Proof. The theorem is obtained as a corollary to the following lemma.

Lemma 1. *If $(x_1, \ldots, x_m)s \equiv_\alpha (y_1, \ldots, y_m)t$, $\rho = (x_1 = a_1, \ldots, x_m = a_m)$, $\sigma = (y_1 = b_1, \ldots, y_m = b_m)$ and $a_i \equiv_\alpha b_i$ $(1 \leq i \leq m)$, then $[s]_\rho \equiv_\alpha [t]_\sigma$.*

Proof. We first prove the lemma for the case where ρ is an essentially first-order environment by induction on $|s|$. Here, by an essentially first-order environment we mean an environment such that for each definition $x = a$ in a, either x is of arity 0 or a is of the form r^*.

We do the case analysis on the shape of s.

1. $s \equiv \sharp^k x[s_1, \ldots, s_n]$. In this case, t is of the form $\sharp^\ell y[t_1, \ldots, t_n]$ and $[s_i]_\rho \equiv_\alpha [t_i]_\sigma$ $(1 \leq i \leq n)$.
   (a) $\sharp^k x$ is defined in ρ. In this case we have $\rho(\sharp^k x) \equiv a_i$ and $\sigma(\sharp^\ell y) \equiv b_i$ for some i $(1 \leq i \leq m)$. Since ρ is essentially first-order, we see that $[s]_\rho \equiv a_i$ and. $[t]_\sigma \equiv b_i$; or else $\rho(\sharp^k x) \equiv \sigma(\sharp^\ell y) \equiv r^*$ for some r since $\rho(\sharp^k x) \equiv_\alpha \sigma(\sharp^\ell y)$. So, we have $[s]_\rho \equiv_\alpha [t]_\sigma$.
   (b) $\sharp^k x$ is free or declared in ρ and $\rho(\sharp^k x) \equiv \sharp^{k'} x$. By induction hypothesis, we have
   $$[s]_\rho \equiv \sharp^{k'} x[[s_1]_\rho, \ldots, [s_n]_\rho] \equiv_\alpha \sharp^{k'} x[[t_1]_\sigma, \ldots, [t_n]_\sigma] \equiv [t]_\sigma.$$
2. $s \equiv c[s_1, \ldots, s_n]$. In this case, t is of the form $c[t_1, \ldots, t_n]$ and, by induction hypothesis, we see that
   $$[s]_\rho \equiv c[[s_1]_\rho, \ldots, [s_n]_\rho] \equiv_\alpha c[[t_1]_\sigma, \ldots, [t_n]_\sigma] \equiv [t]_\sigma.$$
3. $s \equiv (x)\,[s']$. In this case t is of the form $(y)\,[t']$ and $[s']_{(x=\mathsf{v}_{n^*})} \equiv_\alpha [t']_{(y=\mathsf{v}_{n^*})}$. We also have $[s]_\rho \equiv [(x)\,[s']]_\rho \equiv (x)\,[[s']_{(\rho,x)}]$ and $[t]_\sigma \equiv [(y)\,[t']]_\sigma \equiv (y)\,[[t']_{(\sigma,x)}]$. So, if we can show that $[[s']_{(\rho,x)}]_{(x=\mathsf{v}_{n^*})} \equiv_\alpha [[t']_{(\sigma,y)}]_{(y=\mathsf{v}_{n^*})}$, we will be done. We can show this as follows. By Lemma 2 which we prove next, we have $[[s']_{(\rho,x)}]_{(x=\mathsf{v}_{n^*})} \equiv [s']_{(\rho', x=\mathsf{v}_{n^*})}$ and $[[t']_{(\sigma,y)}]_{(x=\mathsf{v}_{n^*})} \equiv [t']_{(\sigma', y=\mathsf{v}_{n^*})}$ where
   $$\rho' :\equiv (x_1 = [a_1]_{(x=\mathsf{v}_{n^*})}, \ldots, x_m = [a_m]_{(x=\mathsf{v}_{n^*})})$$
   and
   $$\sigma' :\equiv (y_1 = [b_1]_{(y=\mathsf{v}_{n^*})}, \ldots, y_m = [b_m]_{(y=\mathsf{v}_{n^*})})$$
   Since $|s'| < |s|$, by induction hypothesis, we have $[s']_{(\rho'.x=\mathsf{v}_{n^*})} \equiv_\alpha [t']_{(\sigma', y=\mathsf{v}_{n^*})}$. Therefore, we have $[[s']_{(\rho,x)}]_{(x=\mathsf{v}_{n^*})} \equiv_\alpha [[t']_{(\sigma,y)}]_{(y=\mathsf{v}_{n^*})}$.

Next, we prove the lemma for the case where ρ is a higher-order environment by induction on $|s|$. We do the case analysis on the shape of s.

1. $s \equiv \sharp^k x[s_1, \ldots, s_n]$. In this case, t is of the form $\sharp^k x[t_1, \ldots, t_n]$ and $s_i \equiv_\alpha t_i$ $(1 \leq i \leq n)$.
   (a) $\sharp^k x$ is defined in ρ and $\rho(\sharp^k x) \equiv (x_1, \ldots, x_n)a$. In this case, we have $[s]_\rho \equiv [a]_{(x_1=[s_1]_\rho, \ldots, x_n=[s_n]_\rho)}$. We also see that $\sharp^k x$ is defined in σ and $\sigma(\sharp^k x)$ is of the form $(y_1, \ldots, y_n)b$ and hence we have $[t]_\sigma \equiv [b]_{(y_1=[t_1]_\rho, \ldots, y_n=[t_n]_\rho)}$. Here, both $(x_1 = [s_1]_\rho, \ldots, x_n = [s_n]_\rho)$ and $(y_1 = [t_1]_\rho, \ldots, y_n = [t_n]_\rho)$ are first-order, and by induction hypothesis, $[s_i]_\rho \equiv_\alpha [t_i]_\sigma$ for $(1 \leq i \leq m)$. So, we have
   $$[s]_\rho \equiv [a]_{(x_1=[s_1]_\rho, \ldots, x_n=[s_n]_\rho)} \equiv_\alpha [b]_{(y_1=[t_1]_\rho, \ldots, y_n=[t_n]_\rho)} \equiv [t]_\sigma.$$
   (b) $\sharp^k x$ is free or declared in ρ and $\rho(\sharp^k x) \equiv \sharp^m x$. Same as 1(b) in the first-order case.
2. $s \equiv c[s_1, \ldots, s_n]$. Same as 2 in the first-order case.
3. $s \equiv (x)s'$. Same as 3 in the first-order case.

Lemma 2 (Substitution Lemma). *If* $\rho = (x_1 = a_1, \ldots, x_m = a_m)$ *and* $\rho' = (x_1 = [a_1]_\rho, \ldots, x_m = [a_m]_\rho)$, *then* $[[b]_{(\rho,x)}]_{(x=a)} \equiv [b]_{(\rho',x=a)}$.

Proof. By induction on $|b|$. We actually show $[[b]_{(\rho,x,\bar{z})}]_{(x=a)} \equiv [b]_{(\rho',x=a,\bar{z})}$ where $\bar{z}$ is a sequence of variables.

4 Natural Framework

In this section we introduce the Natural Framework (NF) which was originally given in Sato [5]. In [5], NF was developed based on a restricted theory of expressions. In this section we revise and extend NF by using the simple theory of expressions we have just defined.

NF is a computational and logical framework which supports the formal development of mathematical theories in the computer environment, and it has been implemented by the author's group at Kyoto University and has been successfully used as a computer aided education tool for students [4].

Based on the theory of expressions we just presented we now define judgements and derivations. In doing so, we first introduce the fundamental concept of *derivation context*. This concept is fundamental since, in general, an expression a containing free variables does not have a fixed meaning since its meaning depends on the meaning of its free variables while free variables do not have fixed meaning. However, in mathematical reasoning we often treat expressions containing free variables. In order to cope with this situation, we will make use of derivation contexts. Namely we will treat expressions containing free variables always *under* a derivation context Γ such that all the free variables in these expressions are declared in Γ.

Although it is possible and actually it is more natural and simpler to use the formal theory from a formal point of view, we will present our theory of judgments and derivations using the informal theory for the sake of readability.

In the following, we will use the following specific constants. `Nil`, `zero`, `succ` (arity 0), `s` (arity 1), `Pair`, ::, $\Rightarrow$, `HD`, : (arity 2), and `CD` (arity 3).

We use the following notational convention.

$$\begin{aligned}
\texttt{<>} &:\equiv \texttt{Nil},\\
\texttt{<}e\,\texttt{|}\,f\texttt{>} &:\equiv \texttt{Pair}[e, f],\\
\texttt{<}e_1, e_2, \ldots, e_n\texttt{>} &:\equiv \texttt{<}e_1\,\texttt{|}\,\texttt{<}e_2\,\texttt{|}\,\cdots\texttt{<}e_n\,\texttt{|Nil>}\cdots\texttt{>>},\\
X :: J &:\equiv ::[X, J],\\
H \Rightarrow J &:\equiv \Rightarrow[H, J].
\end{aligned}$$

An expression of the form $\texttt{<}e_1, e_2, \ldots, e_n\texttt{>}$ is called a *list* and we will define concatenation of two lists by:

$$\texttt{<}e_1, \ldots, e_m\texttt{>} \oplus \texttt{<}f_1, \ldots, f_n\texttt{>} :\equiv \texttt{<}e_1, \ldots, e_m, f_1, \ldots, f_n\texttt{>}.$$

If $V = \texttt{<}x_1, \ldots, x_n\texttt{>}$ is a list of distinct variables, then an expression e is a *V-expression* if $(V)e$ is a closed expression.

4.1 Judgments and derivations

We first define the notion of *judgment*.

Definition 1 (Judgment). *We will call any expression a* judgment. *A judgment of the form $H \Rightarrow J$ is called a* hypothetical judgment *and a judgment of the form* `(`x`)[`J`]` *is called a* universal judgment. *A judgment J is called a* V-judgment *if V is a list of variables and* `(`V`)[`J`]` *is closed.*

Thus, formally speaking, any expression is a judgment. However, in order to make a judgment, or, in order to *assert* a judgment, we must *prove* it. Namely, we have to construct a derivation whose conclusion is the judgment. Below, we will make the notion of derivation precise. To this end, we first define *derivation context*.

Definition 2 (Derivation Context). *We define a* derivation context Γ *together with its* general variable part $\mathrm{GV}(\Gamma)$ *and* variable part $\mathrm{V}(\Gamma)$.

1. Empty context. *The empty list* `<>` *is a derivation context. Its general variable part is* `<>` *and variable part is* `<>`.
2. General variable declaration. *If Γ is a derivation context, and x is a variable not declared in Γ, then $\Gamma \oplus$* `<`x`>` *is a derivation context. Its general variable part is* $\mathrm{GV}(\Gamma) \oplus$ `<`x`>` *and variable part is* $\mathrm{V}(\Gamma) \oplus$ `<`x`>`.
3. Derivation variable declaration. *If Γ is a derivation context, H is a* $\mathrm{V}(\Gamma)$*-expression, and X is a 0-ary variable not declared in Γ, then $\Gamma \oplus$* `<`X `::` J`>` *is a derivation context. Its general variable part is* $\mathrm{GV}(\Gamma)$ *and variable part is* $\mathrm{V}(\Gamma) \oplus$ `<`X`>`.

We now define derivation games.

Definition 3 (Derivation Game). *A list of the form* $\langle c_1 :: R_1, \ldots, c_n :: R_n \rangle$ *is called a* derivation game *if each* c_i *is a 0-ary constant and* R_i *is a closed judgment* $(1 \leq i \leq n)$*. Each* R_i *is called a* rule *of the game and* c_i *is called the* name *of the rule* R_i*.* c_i*'s must be all distinct.*

Derivation games are used to define mathematical or logical theories and also to define computation systems. We will give some examples of derivation games later, but see [5] for more examples of derivation games.

Any closed expression R can be uniquely written in the form

$$(x_1, \ldots, x_m)\,[H_1 \Rightarrow \cdots \Rightarrow H_n \Rightarrow J]$$

where J is *not* a hypothetical judgment. This expression can be instantiated as follows. We fix a list of variables V. If $\rho = (x_1 = e_1, \ldots, x_m = e_m)$ where each e_j $(1 \leq j \leq m)$ is a V-expression, then $[R]_\rho$ is called a V-instance of R and we write $R(e_1, \ldots, e_m)$ for it. Since $R(e_1, \ldots, e_m)$ is of the form $H'_1 \Rightarrow \cdots \Rightarrow H'_n \Rightarrow J'$, we will write

$$\frac{H'_1 \quad \cdots \quad H'_n}{J'}\ R(e_1, \ldots, e_m)$$

for $R(e_1, \ldots, e_m)$. We will sometimes write R itself as

$$\frac{H_1 \quad \cdots \quad H_n}{J}\ R(x_1, \ldots, x_m)$$

in order to make the role of R as a rule clear. Actually, R is a rule-schema, and as we will see in the definition of derivations below, instances of R are used as inference rules when we construct derivations.

We can now proceed to the definition of derivations. Derivations are defined with the following informal meanings of judgments in mind. A hypothetical judgment $H \Rightarrow J$ means that we can derive the judgment J whenever H is derivable. A universal judgment of the form $(x)\,[J]$ means that we can derive the judgment $[J]_{(x=e)}$ for any expression e which is substitutable for x.

Definition 4 (Derivation). *Let* G *be a derivation game. We define a* G-derivation *relative to a derivation context* Γ *as follows. We define its* conclusion *at the same time. In the following definition,* Γ *stands for an arbitrary derivation context. We can see from the definition below, that if* D *is a* G*-derivation under* Γ*, then its conclusion is a* $\mathrm{GV}(\Gamma)$ *expression.*

1. Derivation variable. *If* X *is a derivation variable and* $X :: H$ *is in* Γ*, then*

$$X$$

is a G*-derivation under* Γ *and its conclusion is* H*.*

2. Composition. *Suppose that R is a rule in G and c is the name of the rule R. If $D_1, \ldots, D_n$ are G-derivations under Γ such that their conclusions are $H_1, \ldots, H_n$, respectively, and*

$$\frac{H_1 \quad \cdots \quad H_n}{J}\ R(e_1, \ldots, e_m)$$

is a $\mathrm{GV}(\Gamma)$-instance of R, then

$$\frac{D_1 \quad \cdots \quad D_n}{J}\ c(e_1, \ldots, e_m),$$

which is an abbreviation of the expression $\mathtt{CD}[J, \mathtt{<}c, e_1, \ldots, e_m\mathtt{>}, \mathtt{<}D_1, \ldots, D_n\mathtt{>}]$, is a G-derivation and its conclusion is J.

3. Hypothetical derivation. *If D is a G-derivation under $\Gamma \oplus \mathtt{<}X :: H\mathtt{>}$ and its conclusion is J, then*

$$\mathtt{(}X :: H\mathtt{)\,[}D\mathtt{]},$$

which is an abbreviation of the expression $\mathtt{HD}[H, \mathtt{(}X\mathtt{)\,[}D\mathtt{]}]$, is a G-derivation under Γ and its conclusion is $H \Rightarrow J$.

4. Universal derivation. *If D is a G-derivation under $\Gamma \oplus \mathtt{<}x\mathtt{>}$ and its conclusion is J, then*

$$\mathtt{(}x\mathtt{)\,[}D\mathtt{]}$$

is a G-derivation under Γ and its conclusion is $\mathtt{(}x\mathtt{)\,[}J\mathtt{]}$.

We will write

$$\Gamma \vdash_G D :: J$$

if D is a derivation in G under Γ whose conclusion is J.

A very simple example of a derivation game is the game Nat:

$$\mathsf{Nat} :\equiv \ \mathtt{<}\,\mathtt{zero} :: \mathtt{0} : \mathtt{Nat},\ \mathtt{succ} :: \mathtt{(}n\mathtt{)\,[}n : \mathtt{Nat} \Rightarrow \mathtt{s(}n\mathtt{)} : \mathtt{Nat]}\ \mathtt{>},$$

and, by using obvious notational convention, we can display the two rules of this game as follows. We write $\mathtt{s}(x)$ for $\mathtt{s}[x]$.

$$\frac{}{\mathtt{0} : \mathtt{Nat}}\ \mathtt{zero}() \qquad \frac{n : \mathtt{Nat}}{\mathtt{s}(n) : \mathtt{Nat}}\ \mathtt{succ}(n)$$

In Nat, we can have the following derivation

$$\vdash_{\mathsf{Nat}} D :: \mathtt{s(s(0))} : \mathtt{Nat}.$$

NF provides another notation which is conveniently used to input and display derivations on a computer terminal. In this notation, instead of writing $\Gamma \vdash_G D :: J$ we write:

$$\Gamma \vdash J\ \mathtt{in}\ G\ \mathtt{since}\ D.$$

Also, when writing derivations in this notation, a derivation of the form

$$\frac{D_1 \quad \cdots \quad D_n}{J}\ R(e_1, \ldots, e_m)$$

will be written as:

$$J \texttt{ by } R(e_1, \ldots, e_m) \texttt{ \{}D_1\texttt{ ; } \ldots \texttt{ ; }D_n\texttt{\}}$$

Here is a complete derivation in `Nat` in this notation.

```
 ⊢ (x)[x:Nat ⇒ s(s(x)):Nat] in Nat since
(x)[(X::x:Nat)[
    s(s(x)):Nat by succ(s(x)) {
        s(x):Nat by succ(x) {X}
    }
]]
```

The conclusion of the above derivation asserts that for any expression x, if x is a natural number, then so is `s(s(`x`))`, and the derivation shows us how to actually construct a derivation of `s(s(`x`)):Nat` given a derivation X of x`:Nat`.

We can prove the following basic properties of derivations in the same way as in [5].

Theorem 2 (Decidability). *If G is a derivation game, Γ is a derivation context, D is a $\mathrm{V}(\Gamma)$-expression, J is a $\mathrm{GV}(\Gamma)$-expression, then it is primitive recursively decidable whether $\Gamma \vdash_G D :: J$ or not.*

For a derivation game G, we let $\mathsf{D}(G)$ be the set of G-derivations under the empty context. Then, we have the following corollary which fulfills Kreisel's dictum.

Corollary 1. *For any derivation game G, $\mathsf{D}(G)$ is a primitive recursive subset of E.*

We can also check the following properties of derivations.

Proposition 1. *If $\Gamma \vdash D :: J$, then any free variable in J is declared in $\mathrm{GV}(\Gamma)$, and any free variable in D is declared in $\mathrm{V}(\Gamma)$.*

Derivation games enjoy the following fundamental properties.

Theorem 3. *The following properties hold for any derivation game G.*

1. *Weakening. If $\Gamma \vdash_G D :: J$ and $\Gamma \oplus \Gamma'$ is a context, then $\Gamma \oplus \Gamma' \vdash_G D :: J$.*
2. *Strengthening for general variable. If $\Gamma \oplus$ <x> $\oplus \Gamma' \vdash_G D :: J$, and $x \notin \mathrm{FV}(\Gamma') \cup \mathrm{FV}(D) \cup \mathrm{FV}(J)$, then $\Gamma \oplus \Gamma' \vdash_G D :: J$.*
3. *Strengthening for derivation variable. If $\Gamma \oplus$ <$X :: H$> $\oplus \Gamma' \vdash_G D :: J$, and $X \notin \mathrm{FV}(D)$, then $\Gamma \oplus \Gamma' \vdash_G D :: J$.*

4. *Substitution for derivation variable. If* $\Gamma \oplus \texttt{<}X :: H\texttt{>} \oplus \Gamma' \vdash_G D :: J$ *and* $\Gamma \vdash_G D' :: H$*, then* $\Gamma \oplus \Gamma' \vdash_G [D]_{(X=D')} :: J$.
5. *Substitution for general variable. If* $\Gamma \oplus \texttt{<}x\texttt{>} \oplus \Gamma' \vdash_G D :: J$*, and* e *is a* $\mathrm{GV}(\Gamma)$*-expression substitutable for* x*, then* $\Gamma \oplus [\Gamma']_{(x=e)} \vdash_G [D]_{(x=e)} :: [J]_{(x=e)}$.
6. *Exchange. If* $\Gamma \oplus \texttt{<}e, f\texttt{>} \oplus \Gamma' \vdash_G D :: J$*, and* $\Gamma \oplus \texttt{<}f, e\texttt{>} \oplus \Gamma'$ *is a derivation context, then* $\Gamma \oplus \texttt{<}f, e\texttt{>} \oplus \Gamma' \vdash_G D :: J$.

These basic properties of derivations imply that it is possible to implement a system on a computer that can manipulate these symbolic expressions and decide the correctness of derivations. At Kyoto University we have been developing a computer environment called CAL (for Computation And Logic) [4] which realizes this idea.

There are already several powerful computer systems for developing mathematics with formal verification, including Isabelle [3], Coq [1] and Theorema [2]. NF/CAL is being developed with a similar aim, but at the same time it is used as an education system for teaching logic and computation.

4.2 Lambda calculus in NF

As an example of a derivation that requires higher-order variables in the defining rules, we define the untyped $\lambda\beta$-calculus LambdaBeta as follows. We introduce the following new constants. λF, appF, β, refl, sym, trans, appL, appR, ξ (arity 0), λ (arity 1), $=$, app (arity 2).

Term The λ-terms are defined by the game Term which consists of the following two rules.

$$\frac{(x)\,[x : \texttt{Term} \Rightarrow M[x] : \texttt{Term}]}{\lambda[(x)\,[M[x]]] : \texttt{Term}}\ \lambda\mathrm{F}(M) \qquad \frac{M : \texttt{Term} \quad N : \texttt{Term}}{\texttt{app}[M, N] : \texttt{Term}}\ \texttt{appF}(M, N)$$

We can see from the form of the rule that the variable M in the λF-rule is a unary variable, and we can instantiate M by an expression of the form $(x)e$.

An example of a λ-term is given by the following derivation, where we will write $\lambda(x)\,[M]$ for $\lambda[(x)\,[M]]$ and app(M, N) for app$[M, N]$.

```
Y::y:Term ⊢ λ(x)[app(x, y)]:Term in Term since
λ(x)[app(x, y)]:Term by λF {
    (x)[(X::x:Term)[
        app(x, y):Term by appF {X; Y}
    ]]
}
```

EqTerm We can define the β-equality relation on λ-terms by the game EqTerm which is defined by the following rules, where we write $M = N$ for $= [M, N]$.

$$\frac{M : \texttt{Term}}{M = M}\ \texttt{refl}(M) \qquad \frac{M = N}{N = M}\ \texttt{sym}(M, N) \qquad \frac{M = N \quad N = L}{M = L}\ \texttt{trans}(L, M, N)$$

$$\frac{M = N \quad Z : \texttt{Term}}{\texttt{app}[M, Z] = \texttt{app}[N, Z]}\ \texttt{appL}(M, N, Z) \qquad \frac{Z : \texttt{Term} \quad M = N}{\texttt{app}[Z, M] = \texttt{app}[Z, N]}\ \texttt{appR}(M, N, Z)$$

$$\frac{\lambda[(x)\,[M[x]]] : \texttt{Term} \quad N : \texttt{Term}}{\texttt{app}[\lambda[(x)\,[M[x]]], N] = M[N]}\ \beta(M, N) \qquad \frac{(x)\,[x : \texttt{Term} \Rightarrow M[x] = N[x]]}{\lambda[(x)\,[M[x]]] = \lambda[(x)\,[N[x]]]}\ \xi(M, N)$$

We can see from the form of the rule that the variable M in the β-rule and variables M, N in the ξ-rule are of arity 1.

We can now define the untyped $\lambda\beta$-calculus LambdaBeta by putting:

$$\mathsf{LambdaBeta} :\equiv \mathsf{Term} \oplus \mathsf{EqTerm}.$$

We give below an example of a formal derivation of a reduction in the $\lambda\beta$-calculus.

```
Y::y:Term ⊢ app(λ(x)[app(x, x)], y) = app(y, y) in LambdaBeta since
app(λ(x)[app(x, x)], y) = app(y, y) by β((x)[app(x, x)], y) {
    λ(x)[app(x, x)]:Term by λF {
        (x)[(X::x:Term)[
             app(x, y):Term by appF {X; Y}
        ]]
    };
    Y
}
```

In the above derivation, the β-rule is instantiated by the environment $\rho = (M = (x)\,[\texttt{app}[x, x]], N = y)$. Hence $M[N]$ is instantiated as follows:

$$\begin{aligned}
[M[N]]_\rho &\equiv [\texttt{app}[x, x]]_{(x=[N]_\rho)} \text{ since } (M = (x)\,[\texttt{app}[x, x]]) \in \rho \\
&\equiv [\texttt{app}[x, x]]_{(x=y)} \\
&\equiv \texttt{app}[y, y]
\end{aligned}$$

5 Conclusion

We have introduced a simple theory of expressions equipped with the operations of abstraction and instantiation. Abstraction is realized by a syntactic constructor but instantiation is realized by an external operation. In the usual systems

of expression with named variables for binders, it is necessary to rename local binding variables to avoid unsolicited capture of free variables. In our system, we have introduced variable references which can refer to any surrounding variable. Variable references are already introduced in [6], but the definition of substitution in it is very complicated. We could simplify the definition by introducing the extended notion of environment.

The theory of expressions introduced here is a modification of our previous theory of expressions given in [5]. The previous theory did not have the notion of arity, and simpler than the current theory. However the previous theory could not define derivation games as objects in the theory. In the current theory it is possible to define rules of derivation games by closed expressions.

We have also shown that using this new theory of expressions, we can reformulate the theory of judgments and derivations introduced in [6].

Acknowledgements

The author wishes to thank Bruno Buchberger, Murdoch Gabbay, Atsushi Igarashi, Yukiyoshi Kameyama, Per Martin-Löf, Koji Nakazawa, Takafumi Sakurai, and René Vestergaard, for having fruitful discussions on expressions with the author.

References

1. Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer, 2004.
2. B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, W. Windsteiger, The Theorema Project: A Progress Report, in *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, August 6-7, 2000, St. Andrews, Scotland)*, M. Kerber and M. Kohlhase (eds.), A.K. Peters, Natick, Massachusetts, pp. 98-113.
3. T. Nipkow, L.C. Paulson and M. Wenzel, *Isabell/HOL — A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science, **2283**, Springer 2002.
4. M. Sato, Y. Kameyama and I. Takeuti, CAL: A computer assisted learning system for computation and logic, in Moreno-Diaz, R., Buchberger, B. and Freire, J-L. eds., *Computer Aided Systems Theory – EUROCAST 2001*, Lecture Notes in Computer Science, **2718**, pp. 509 – 524, Springer 2001.
5. M. Sato, Theory of Judgments and Derivations, in Arikawa, S. and Shinohara, A. eds., *Progress in Discovery Science*, Lecture Notes in Artificial Intelligence **2281**, pp. 78 – 122, Springer, 2002.
6. M. Sato, T. Sakurai, Y. Kameyama, *A Simply Typed Context Calculus with First-Class Environments*, Journal of Functional and Logic Programming, Vol. 2002, No. 4, March 2002.
7. R. Vestergaard, *The primitive proof theory of the λ-calculus*, PhD thesis, School of Maths and Computer Sciences, Heriot-Watt University, 2003.