A Framework for Checking Proofs Naturally

Masahiko Sato

Graduate School of Informatics, Kyoto University
masahiko@kuis.kyoto-u.ac.jp

Abstract. We propose a natural framework, called NF, which supports development of formal proofs on a computer. NF is based on a theory of Judgments and Derivations.
NF is designed by observing how working mathematical theories are created and developed. Our observation is that the notions of judgments and derivations are the two fundamental notions used in any mathematical activity. We have therefore developed a theory of judgments and derivations and designed a framework in which the theory provides a uniform and common *play ground* on which various mathematical theories can be defined as *derivation games* and can be played, namely, can write and check proofs.
NF is equipped with a higher-order intuitionistic logic and derivations (proofs) are described following Gentzen's natural deduction style.
NF is part of an interactive computer environment CAL (Computation and Logic) and it is also referred to as NF/CAL. CAL is written in Emacs Lisp and it is run within a special buffer of the Emacs editor. CAL consists of user interface, a general purpose parser and a checker for checking proofs of NF derivation games.
NF/CAL system has been successfully used as an education system for teaching computation and logic for undergraduate students for about 8 years.
We will give an overview of the NF/CAL system both from theoretical and practical sides.

1 Introduction

The Curry-Howard isomorphism enables us to identify proofs with programs. Therefore, simply by developing a formal computer environment for checking proofs, we also obtain an environment for checking the correctness of programs. The NF/CAL system which we have been developing provides such a computer environment. Although there are already several powerful environments with the same objective, such as Coq [3], Isabelle [7], our system is unique in that general forms of proofs are uniformly and formally defined and can be treated as first class objects of the system.

Another objectives for developing our system are foundation and education. To achieve these objectives, our system is based on a theory of judgments and derivations introduced in [10]. The theory is later modified by changing the basic data structure of expressions. This new data structure of expressions was introduced in [12].

In this paper, we outline the current status of NF/CAL with examples. We also present a new definition of expressions which enables the classification of expressions according to the three basic syntactic categories of *object*, *judgment* and *derivation*.

2 Expression

Proofs as well as judgments are linguistic objects. We need *expressions* as a means to represent such linguistic objects in a uniform manner. The expressions we introduce here can be used to define various linguistic objects uniformly, and we can define *higher-order abstract syntax* by using them. The key idea behind the following definition of expressions is the usage of *arity*.

We will, of course, use natural language to explain the notion of expressions, and for that we will have to use mathematical expressions (as part of our language). Thus we have expressions in the object-language and also in the meta-language. We will therefore call the former *object-expressions* and the latter *meta-expression* when we feel it necessary to distinguish them clearly.

2.1 Category

Our goal is to define (well-formed) expressions, and those well-formed expressions should be analyzable both syntactically and semantically by human readers. In this way, expressions can express *thoughts* and *objects*. Those expressions which express thoughts will be said to belong to the (syntactic) *category* j (for judgement), and those which express objects will belong to the category o (for objects). We have here one more category d of *derivations*. Derivations are used to provide *evidences* for the truth of judgments. All the categories we need are characterized by the following grammar.

$$\begin{aligned}\gamma &::= \mathsf{o} \mid \mathsf{j} \mid \mathsf{d}\\ \kappa &::= \gamma \mid (\kappa_1, \ldots, \kappa_n)\gamma \ (n > 0)\end{aligned}$$

γ stands for *ground categories* and there are exactly three ground categories o, j and d. κ stands for (general) categories and a category $(\kappa_1, \ldots, \kappa_n)\gamma$ is said to be *higher-order*, and a category γ is said to be *first-order*. A first-order category is nothing but a ground category.

2.2 Arity and Expression

Expressions are built up by combining *variables* and *constants* appropriately. We restrict possible combinations by assigning a *category* to each variable or constant. If a category κ is assigned to a variable x (constant c), we will say that x (c, resp.) has *arity* κ. For an arity $\kappa = (\kappa_1, \ldots, \kappa_n)\gamma$, we call n the *arity number* of κ. The arity number of a ground category is defined to be 0. If a variable x has a positive arity number n then x standing by itself is not an expression, but $x[e_1, \ldots, e_n]$ becomes a valid expression provided that $e_1, \ldots, e_n$ are expressions. Thus, x is an *unsaturated* entity and it expects n arguments to become a *saturated* object. So, x is a variable ranging over *higher-order abstracts*. This way of analyzing an expression is due to Frege [5], and Martin-Löf once adopted this idea in his theory of expressions [6].

The category $(\kappa_1, \ldots, \kappa_n)\gamma$ is also written as $\gamma[\kappa_1, \ldots, \kappa_n]$ when it is treated as an arity.

We define expressions slightly informally by taking the notion of α-equivalence for granted. For each arity κ, we assume that we have countably infinite set set of *variables* (x, y, z) with arity κ. For each arity κ, we assume that we have countably infinite set set of *constants* (c, d) with arity κ. We assume that all these sets are mutually disjoint. We will write $x@\kappa$ if x has arity κ and similarly for constants.

We define expressions as follows, where $e : \kappa$ will mean that e is an *expression* which belongs to category κ.

$$\frac{x@\gamma[\kappa_1, \ldots, \kappa_n] \quad a_1 : \kappa_1 \quad \cdots \quad a_n : \kappa_n}{x[a_1, \ldots, a_n] : \gamma}\ \mathsf{var}$$

$$\frac{c@\gamma[\kappa_1, \ldots, \kappa_n] \quad a_1 : \kappa_1 \quad \cdots \quad a_n : \kappa_n}{c[a_1, \ldots, a_n] : \gamma}\ \mathsf{const}$$

$$\frac{x_1@\kappa_1 \quad \cdots \quad x_n@\kappa_n \quad a : \gamma}{(x_1, \ldots, x_n)a : (\kappa_1, \ldots, \kappa_n)\gamma}\ \mathsf{abs}$$

In the var and const rules, we will understand that $x[a_1, \ldots, a_n]$ ($c[a_1, \ldots, a_n]$) stands for x (c, resp.) when $n = 0$. In the abs rule, n must be a positive natural number. We will also write $(x_1, \ldots, x_n)[a]$ for $(x_1, \ldots, x_n)a$ when we wish to emphasize that $x_1, \ldots, x_n$ are the binding variable and their scope is a.

The notion of *free* and *bound* occurrences of variables in an expression is defined as usual. Namely, free occurrences of $x_1, \ldots, x_n$ in a becomes bound in $(x_1, \ldots, x_n)a$. An expression is *closed* if it contains no free occurrences of variables. We write $\mathrm{FV}(e)$ for the set of variables having free occurrences in e.

2.3 Environment

We define environments which are used to instantiate abstract expressions and also to define substitution. Let x be a variable of arity κ. We say that an expression e is *substitutable for x* if e belongs to category κ.

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 $\{x_1, \ldots, x_k\}$.

2.4 Instantiation

Given an expression e and an environment ρ, we define an expression $[e]_\rho$ as follows. In the clause 5, we choose fresh local variables as necessary.

1. $[x]_\rho :\equiv e$ if $x@\gamma$ 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. $[(x_1, \ldots, x_n)[a]]_\rho :\equiv (x_1, \ldots, x_n)[[a]_\rho]$.

In order to check the well-definedness of the above definition, we define the *rank* $r(\kappa)$ of a category κ as follows.

1. $r(\gamma) = 0$.
2. $r((\kappa_1, \ldots, \kappa_n)\gamma) = \max\{r(\kappa_1), \ldots, r(\kappa_n)\} + 1$.

The rank of an environment ρ is then defined as follows.

$$r(\rho) = \max\{r(\kappa) | x \in |\rho| \text{ and } x@\kappa\}.$$

We define the rank of an empty environment to be 0.

We can now check the well-definedness of the instantiation operation inductively as follows. Let ρ be an environment and e be an expression. If the rank of ρ is 0, then we can compute $[e]_\rho$ without using the clause 2, and this means that we can compute it by induction on the construction of e. If the rank of ρ is positive, we can again compute $[e]_\rho$ by induction on the construction of e. We note that the clause 2 is now taken care of by appealing to the induction hypothesis.

It is essential to distinguish arities of 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 arities of variables, the above computation is not possible. To see this, let us write $x_1[x_2]$ for $x[x]$ to distiguishes the two occurrences of x in $x[x]$. Then we have $x_1[x_2] : \gamma$, $x_1@\gamma[\kappa]$ and $x_2 : \kappa$ for some ground category γ and category κ. That $x_2 : \kappa$ implies $x_2@\kappa$. Therefore x must have arity $\gamma[\kappa]$ and κ, which is impossible.

3 Natural Framework

In this section we introduce the Natural Framework (NF) which was originally given in Sato [10]. In [10], 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 [9].

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*. A derivation context provides a context in which we construct our derivation, and it is essentially a list of assumptions available in the current context.

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:

$$\begin{aligned}
\Rightarrow &\ @\ \mathsf{j}[\mathsf{j},\mathsf{j}],\\
\forall &\ @\ \mathsf{j}[(\mathsf{o})\mathsf{j}],\\
\mathtt{CD} &\ @\ \mathsf{d}[\mathsf{j},(\mathsf{d})\mathsf{d}],\\
\mathtt{UD} &\ @\ \mathsf{d}[(\mathsf{o})\mathsf{d}],\\
: &\ @\ \mathsf{j}[\mathsf{o},\mathsf{o}].
\end{aligned}$$

We use the following notational convention.

$$\begin{aligned}
H \Rightarrow J &:\equiv\ \Rightarrow[H, J],\\
\forall(x)J &:\equiv \forall[(x)J].
\end{aligned}$$

In order to manipulate sequences of expressions formally, we will write $\langle e_1, \dots, e_n \rangle$ for the sequence $e_1, \dots, e_n$ of expressions. Thus $\langle\rangle$ stands for the empty sequence. We define concatenation of two sequences by:

$$\langle e_1, \dots, e_m \rangle \oplus \langle f_1, \dots, f_n \rangle :\equiv \langle e_1, \dots, e_m, f_1, \dots, f_n \rangle.$$

3.1 Judgments and derivations

We first define the notion of *judgment*.

Definition 1 (Judgment). *We will call any expression which belongs to category* j *a* judgment. *A judgment of the form* $H \Rightarrow J$ *is called a* conditional judgment *and a judgment of the form* $\forall(x)J$ *is called a* universal judgment.

Thus, formally speaking, any expression belonging to category j 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*. We will call a variable with arity d a *derivation variable*.

Definition 2 (Derivation Context). *If $H_1, \ldots, H_n$ $(n \geq 0)$ are judgments and $X_1, \ldots, X_n$ are distinct derivation variables, then a meta-expression $\varGamma$ of the form:*

$$X_1 :: H_1, \cdots, X_n :: H_n \vdash [\ ]$$

is a derivation context.

Each $X_i :: H_i$ is called an *assumption* of the derivation context. If $\varGamma$ is a derivation context of the above form and J is a judgment, then the meta-expression:

$$X_1 :: H_1, \cdots, X_n :: H_n \vdash J$$

is called a *hypothetical judgment*, and it is also written as $\varGamma[J]$. We define the set of free variables in $\varGamma[J]$ by putting $\mathrm{FV}(\varGamma[J]) = \mathrm{FV}(\varGamma) \cup \mathrm{FV}(H_1) \cdots \mathrm{FV}(H_n) \cup \mathrm{FV}(J)$. The variables in $\mathrm{FV}(\varGamma[J])$ are called the *parameters* of $\varGamma[J]$. The parameters $X_1, \ldots, X_n$ are called *derivation parameters.*

We define the notion of a *rule* as follows. If $H_1, \ldots, H_n$ and J are judgments, $\langle x_1@\kappa_1, \ldots, x_m@\kappa_m \rangle$ is an enumeration of free variables in these judgments, and c is a constant of arity $\mathsf{d}[\kappa_1, \ldots, \kappa_m, \mathsf{d}, \ldots, \mathsf{d}]$ with arity level $m + n$, then the meta-expression R:

$$\langle \langle H_1, \ldots, H_n \rangle, J, c, \langle x_1, \ldots, x_m \rangle \rangle$$

is called a rule. c is called the *name* of the rule. The rule R is also figuratively written as:

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

Informally speaking, this rule represents an inference rule which allows us to infer J provided that we can derive all the H_i's. The variables, $x_1, \ldots, x_m$ are called the *parameters* of the rule. Reflecting this informal meaning of a rule, we also write the rule R above as follows, using a notation used to represent Horn clauses:

$$c(x_1, \ldots, x_m) :: J \texttt{ :- } H_1, \cdots, H_n.$$

Suppose that R is a rule of the above form, and that $\rho = \{x_1 = e_1, \ldots, x_m = e_m\}$ is an environment. Then the *ρ-instance* of R, written $[R]_\rho$, is defined as the meta expression:

$$\langle \langle [H_1]_\rho, \ldots, [H_n]_\rho \rangle, [J]_\rho, c, \langle e_1, \ldots, e_m \rangle \rangle.$$

This meta expression is also written:

$$\frac{[H_1]_\rho \quad \cdots \quad [H_n]_\rho}{[J]_\rho}\ c(e_1, \ldots, e_m).$$

We now define derivation games.

Definition 3 (Derivation Game). *A sequence of the form $\langle R_1, \ldots, R_n \rangle$ is called a* derivation game *if each R_i is a rule and the names of the rules are 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 [10] for more examples of derivation games. (The notion of derivation game introduced in [10] is based on our earlier conception of expressions, but most of derivation games there can be considered as derivation games in the sense of this paper with a slight modification.)

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

Definition 4 (Derivation). *Let G be a derivation game, J be a judgment and Γ be a derivation context. We will inductively define the meaning of meta judgment:* D is a G-derivation of J in Γ. *We also read this judgment as:* D is a G-derivation of $\Gamma \vdash J$.

1. Derivation variable. *If X is a derivation variable and $X :: H$ is an assumption in Γ, then*
$$X$$
is a G-derivation of H in Γ.
2. Application of a rule. *Suppose that R is a rule in G. If $D_1, \ldots, D_n$ are G-derivations in Γ of $H_1, \ldots, H_n$, respectively, and*
$$\frac{H_1 \quad \cdots \quad H_n}{J}\ c(e_1, \ldots, e_m)$$
is an instance of R, then
$$c[e_1, \ldots, e_m, D_1, \ldots, D_n]$$
is a G-derivation of J in Γ. We may also write this derivation as:
$$\frac{D_1 \quad \cdots \quad D_n}{J}\ c(e_1, \ldots, e_m).$$
3. Conditional derivation. *If D is a G-derivation of J in $\Gamma, X :: H$, then*
$$\mathtt{CD}[H, (X)D]$$
is a G-derivation of $H \Rightarrow J$ in Γ. This derivation is also written as: $(X :: H)D$.
4. Universal derivation. *If D is a G-derivation of J in Γ, and x@$\mathtt{o}$ is a variable not in $\mathrm{FV}(\Gamma)$, then*
$$\mathtt{UD}[(x)D]$$
is a G-derivation of $\forall(x)J$ in Γ. This derivation is also simply written $(x)D$ when it is clear from the context that we are talking about derivations.

We remark that each derivation introduced above is an object expression belonging to category d because of the arity of the constants used to construct derivations.

We will write

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

if D is a G-derivation of J in Γ.

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

$$\mathsf{Nat} :\equiv \langle\ \mathtt{zero} :: \mathtt{0} : \mathtt{Nat}\ \mathtt{:-}\ ,\ \mathtt{succ}(n) :: \mathtt{s}[n] : \mathtt{Nat}\ \mathtt{:-}\ n : \mathtt{Nat}\ \rangle,$$

where $\mathtt{zero}@\mathsf{d}$, $\mathtt{0}@\mathsf{o}$, $\mathtt{Nat}@\mathsf{o}$, $\mathtt{succ}@\mathsf{d}[\mathsf{o},\mathsf{d}]$, $n@\mathsf{o}$ and $\mathtt{s}@\mathsf{o}[\mathsf{o}]$. 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

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

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

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

will be written as:

$$J\ \mathtt{by}\ c(e_1, \ldots, e_m)\ \mathtt{\{}D_1\mathtt{;}\ \ldots\ \mathtt{;}D_n\mathtt{\}}$$

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 $\mathtt{s(s(}x\mathtt{))}$, and the derivation shows us how to actually construct a derivation of $\mathtt{s(s(}x\mathtt{)):Nat}$ given a derivation X of $x\mathtt{:Nat}$.

We can prove that is is decidable whether a given expression is a correct derivation of a given game in the same way as in [10]. Therefore 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) [9] which realizes this idea.

There are already several powerful computer systems for developing mathematics with formal verification, including Isabelle [7], Coq [3] and Theorema [4]. 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.

3.2 Predicate calculus in NF

As an example of a derivation that requires higher-order variables in the defining rules, we define the intuitionistic predicate calculus Pred as follows. The game consists of the following 15 rules.

$$
\begin{array}{rcl}
\langle\ \Rightarrow\texttt{-I}(P@\mathsf{j}, Q@\mathsf{j}) & :: & P \Rightarrow Q \texttt{ :- } P \Rightarrow Q, \\
\Rightarrow\texttt{-E}(P@\mathsf{j}, Q@\mathsf{j}) & :: & Q \texttt{ :- } P \Rightarrow Q, P, \\
\neg\texttt{-I}(P@\mathsf{j}) & :: & \neg P \texttt{ :- } P \Rightarrow \bot, \\
\neg\texttt{-E}(P@\mathsf{j}) & :: & \bot \texttt{ :- } \neg P, P, \\
\wedge\texttt{-I}(P@\mathsf{j}, Q@\mathsf{j}) & :: & P \wedge Q \texttt{ :- } P, Q, \\
\wedge\texttt{-EL}(P@\mathsf{j}, Q@\mathsf{j}) & :: & P \texttt{ :- } P \wedge Q, \\
\wedge\texttt{-ER}(P@\mathsf{j}, Q@\mathsf{j}) & :: & Q \texttt{ :- } P \wedge Q, \\
\vee\texttt{-IL}(P@\mathsf{j}, Q@\mathsf{j}) & :: & P \vee Q \texttt{ :- } P, \\
\vee\texttt{-IR}(P@\mathsf{j}, Q@\mathsf{j}) & :: & P \vee Q \texttt{ :- } Q, \\
\vee\texttt{-E}(P@\mathsf{j}, Q@\mathsf{j}, R@\mathsf{j}) & :: & R \texttt{ :- } P \vee Q, P \Rightarrow R, Q \Rightarrow R, \\
\bot\texttt{-E}(P@\mathsf{j}) & :: & P \texttt{ :- } \bot, \\
\forall\texttt{-I}(P@\mathsf{j}[\mathsf{o}]) & :: & \forall(x)P[x] \texttt{ :- } \forall(x)P[x], \\
\forall\texttt{-E}(P@\mathsf{j}[\mathsf{o}], a@\mathsf{o}) & :: & P[a] \texttt{ :- } \forall(x)P[x], \\
\exists\texttt{-I}(P@\mathsf{j}[\mathsf{o}], a@\mathsf{o}) & :: & \exists(x)P[x] \texttt{ :- } P[a], \\
\exists\texttt{-E}(P@\mathsf{j}[\mathsf{o}], Q@\mathsf{j}) & :: & Q \texttt{ :- } \exists(x)P[x], \forall(x)[P[x] \Rightarrow Q]\ \rangle
\end{array}
$$

In the above rules, we have explicitly declared the arities of the parameters. Note that in the last four rules involving quantifiers, the variable P has higher-order arity $\mathsf{j}[\mathsf{o}]$. The arities of the names of the game can be easily inferred. For example, the constant $\vee\texttt{-E}$ has arity $\mathsf{d}[\mathsf{j},\mathsf{j},\mathsf{j},\mathsf{d},\mathsf{d},\mathsf{d}]$ and $\forall\texttt{-E}$ has arity $\mathsf{d}[(\mathsf{o})\mathsf{j},\mathsf{o},\mathsf{d}]$ Logical operators have the following arities:

$$\bot@\mathsf{j}, \neg@\mathsf{j}[\mathsf{j}], \wedge@\mathsf{j}[\mathsf{j},\mathsf{j}], \vee@\mathsf{j}[\mathsf{j},\mathsf{j}], \exists@\mathsf{j}[(\mathsf{o})\mathsf{j}].$$

We remark that the rules $\Rightarrow\texttt{-I}$ and $\forall\texttt{-I}$ are redundant since both of these rules only derive the same judgments as their premises, and there are already rules to derive conditional judgments and universal judgments as part of built-in rules for constructing derivations. We have nevertheless included these rules so that the derivations become closer to ordinary derivations in the natural deduction style predicate calculus.

CAL provides a mechanism to automatically convert rules of games into LaTeX source file which can be used to show rules in human-friendly form. For example, the ∃-E rule becomes as follows:

$$\dfrac{\exists(x)\,[\boldsymbol{P}[x]] \quad \forall(x)\,[\boldsymbol{P}[x] \Rightarrow \boldsymbol{Q}]}{\boldsymbol{Q}}\ \exists\text{-E}$$

We give an example of a derivation in Pred.

```
A ⊢ ¬¬(A∨¬A) in Pred since
¬¬(A∨¬A) by ¬-I {
  (X::¬(A∨¬A))[
    ⊥ by ¬-E {
      ¬¬A by ¬-I {
        (Y::¬A)[
          ⊥ by ¬-E {
            X;
            A∨¬A by ∨-IR {Y}
          }]};
      ¬A by ¬-I {
        (Z::A)[
          ⊥ by ¬-E {
            X;
            A∨¬A by ∨-IL {Z}
          }]}}]}
```

In the above derivation A is a variable of arity j. This derivation can be converted into the following natural deduction style proof by CAL.

$$\begin{array}{l}(X\,::\,\neg(A\vee\neg A))\\ \dfrac{\left[\dfrac{\dfrac{\begin{array}{l}(Y::\neg A)\\ \left[\dfrac{X \quad \dfrac{Y}{A\vee\neg A}\,\vee\text{-IR}}{\bot}\,\neg\text{-E}\right]\end{array}}{\neg\neg A}\,\neg\text{-I} \qquad \dfrac{\begin{array}{l}(Z::A)\\ \left[\dfrac{X \quad \dfrac{Z}{A\vee\neg A}\,\vee\text{-IL}}{\bot}\,\neg\text{-E}\right]\end{array}}{\neg A}\,\neg\text{-I}}{\bot}\,\neg\text{-E}\right]}{\neg\neg(A\vee\neg A)}\,\neg\text{-I}\end{array}$$

4 Defining λ-terms in NF

In this section, we define λ-terms formally in NF as a derivation game. The λ-terms we define here is a *conservative* extension of traditional λ-terms with explicit bound variable names. The extension is done by adding the notion of *variable reference* which enables us define the substitution operation naturally even for the case where the name of a binding variable must be renamed to avoid capturing of free variables after substitution.

Here we define λ-terms literally as linguistic objects, namely, as sequences of characters, without using the higher-order abstract syntax available in NF. It is also possible to defined λ-terms using higher-order abstract syntax, and for such a definition the reader is referred to [10].

4.1 λ-terms

We need several auxiliary notions before we can define λ-terms.

Variables are defined by the following rules. The premises of the two rules below are meta-judgments. Although it is possible to define these judgments formally, we skipped the definitions for the sake of simplicity. Thus, for example, the meta-judgment '*x is a variable*' means that x is a sequence of characters which can be classified as a variable according to a certain grammatical rule. In the NF system, the truth of this judgment is checked by a Lisp program which checks the equality of NF objects. In the following we also avoid writing down details of arities of constants introduced.

$$\frac{\boldsymbol{x}\ \textit{is a variable}}{\boldsymbol{x} : \texttt{variable}}\ \texttt{Variable} \qquad \frac{\boldsymbol{x}\ \textit{and}\ \boldsymbol{y}\ \textit{are distinct variables}}{\boldsymbol{x}\ \texttt{NeqVar}\ \boldsymbol{y}}\ \texttt{neqvar}$$

Variable reference are defined as follows. In the following definitions, the notation $\ulcorner\cdots\urcorner$ stands for Quine's quasi-quotation [8], which is also known as back-quotation in the Lisp language. Namely, everything inside the quotation marks $\ulcorner$ and $\urcorner$ is quoted with the exception that variables in ***bold*** font are not quoted but will act as ordinary variables in the same way as those variables outside the quasi-quotation. In this way, we can talk about sequences of tokens in a convenient way.

$$\frac{\boldsymbol{x} : \texttt{variable}}{\boldsymbol{x} : \texttt{varref}}\ \texttt{varref-var} \qquad \frac{\boldsymbol{r} : \texttt{varref}}{\ulcorner\texttt{\#}\boldsymbol{r}\urcorner : \texttt{varref}}\ \texttt{varref-sharp}$$

$$\frac{\boldsymbol{x} : \texttt{variable}}{\boldsymbol{x}\ \texttt{IsCoreOf}\ \boldsymbol{x}}\ \texttt{IC00} \qquad \frac{\boldsymbol{x}\ \texttt{IsCoreOf}\ \boldsymbol{r}}{\boldsymbol{x}\ \texttt{IsCoreOf}\ \ulcorner\texttt{\#}\boldsymbol{r}\urcorner}\ \texttt{IC01}$$

$$\frac{\boldsymbol{x}\ \texttt{IsCoreOf}\ \boldsymbol{r} \quad \boldsymbol{y}\ \texttt{IsCoreOf}\ \boldsymbol{s} \quad \boldsymbol{x}\ \texttt{NeqVar}\ \boldsymbol{y}}{\boldsymbol{r}\ \texttt{NeqCore}\ \boldsymbol{s}}\ \texttt{NEQC}$$

$$\frac{\boldsymbol{x} : \texttt{variable}}{\ulcorner\texttt{\#}\boldsymbol{x}\urcorner\ \texttt{Covers}\ \boldsymbol{x}}\ \texttt{covers1} \qquad \frac{\boldsymbol{x} : \texttt{variable} \quad \boldsymbol{r}\ \texttt{Covers}\ \boldsymbol{x}}{\ulcorner\texttt{\#}\boldsymbol{r}\urcorner\ \texttt{Covers}\ \boldsymbol{x}}\ \texttt{covers2}$$

$$\frac{\boldsymbol{r}\ \texttt{Covers}\ \boldsymbol{s}}{\ulcorner\texttt{\#}\boldsymbol{r}\urcorner\ \texttt{Covers}\ \ulcorner\texttt{\#}\boldsymbol{s}\urcorner}\ \texttt{covers3}$$

We can see that variable references are obtained by prepending a finite number of $\sharp$ symbols in front of variables.

Next, we define *variable sequences* as follows. The first rule below says that an empty sequence is a variable sequence. The judgment $\varGamma$ `EqLen` $\varDelta$ means that $\varGamma$ and $\varDelta$ are variable sequences of the same length.

$$\frac{}{\ulcorner\urcorner : \texttt{varseq}}\ \texttt{varseq0} \qquad \frac{\boldsymbol{\Gamma} : \texttt{varseq} \quad \boldsymbol{x} : \texttt{variable}}{\ulcorner \boldsymbol{\Gamma},\ \boldsymbol{x} \urcorner : \texttt{varseq}}\ \texttt{varseq1}$$

$$\frac{}{\ulcorner\urcorner\ \texttt{EqLen}\ \ulcorner\urcorner}\ \texttt{eqlen0} \qquad \frac{\boldsymbol{\Gamma}\ \texttt{EqLen}\ \boldsymbol{\Delta} \quad \boldsymbol{x} : \texttt{variable} \quad \boldsymbol{y} : \texttt{variable}}{\ulcorner \boldsymbol{\Gamma},\ \boldsymbol{x} \urcorner\ \texttt{EqLen}\ \ulcorner \boldsymbol{\Delta},\ \boldsymbol{y} \urcorner}\ \texttt{eqlen1}$$

We can now define λ-terms. Note that the first rule below is an extension of traditional definition of λ-terms, since there only variables can be used to define the initial set of λ-terms while here we can use variable references as the initial set. The judgment $M : \Lambda$ means that M is a λ-term.

$$\frac{\boldsymbol{r} : \texttt{varref}}{\boldsymbol{r} : \Lambda}\ \texttt{lambda-varref} \qquad \frac{\boldsymbol{x} : \texttt{variable} \quad \boldsymbol{M} : \Lambda}{\ulcorner \lambda \boldsymbol{x}[\boldsymbol{M}] \urcorner : \Lambda}\ \texttt{lambda-abs}$$

$$\frac{\boldsymbol{M} : \Lambda \quad \boldsymbol{N} : \Lambda}{\ulcorner \boldsymbol{M}(\boldsymbol{N}) \urcorner : \Lambda}\ \texttt{lambda-app}$$

4.2 Substitution

We define *substitution* in this subsection. First, we define how we can *push* a term through a variable reference. The judgment $M \uparrow r = N$ means that the result of pushing a λ-term M through a variable reference r is N.

$$\frac{\boldsymbol{r} : \texttt{varref}}{\boldsymbol{r} \uparrow \boldsymbol{r} = \ulcorner \# \boldsymbol{r} \urcorner}\ \texttt{push-hit} \qquad \frac{\boldsymbol{r}\ \texttt{Covers}\ \boldsymbol{s}}{\boldsymbol{r} \uparrow \boldsymbol{s} = \ulcorner \# \boldsymbol{r} \urcorner}\ \texttt{push-up}$$

$$\frac{\boldsymbol{s}\ \texttt{Covers}\ \boldsymbol{r}}{\boldsymbol{r} \uparrow \boldsymbol{s} = \boldsymbol{r}}\ \texttt{push-keep} \qquad \frac{\boldsymbol{r}\ \texttt{NeqCore}\ \boldsymbol{s}}{\boldsymbol{r} \uparrow \boldsymbol{s} = \boldsymbol{r}}\ \texttt{push-miss}$$

$$\frac{\boldsymbol{s} \uparrow \boldsymbol{x} = \boldsymbol{r} \quad \boldsymbol{M} \uparrow \boldsymbol{r} = \boldsymbol{P}}{\ulcorner \lambda \boldsymbol{x}[\boldsymbol{M}] \urcorner \uparrow \boldsymbol{s} = \ulcorner \lambda \boldsymbol{x}[\boldsymbol{P}] \urcorner}\ \texttt{push-abs}$$

$$\frac{\boldsymbol{M} \uparrow \boldsymbol{s} = \boldsymbol{P} \quad \boldsymbol{N} \uparrow \boldsymbol{s} = \boldsymbol{Q}}{\ulcorner \boldsymbol{M}(\boldsymbol{N}) \urcorner \uparrow \boldsymbol{s} = \ulcorner \boldsymbol{P}(\boldsymbol{Q}) \urcorner}\ \texttt{push-app}$$

Substitution can be defined as follows. The judgment $M[P/x] = N$ means that the result of substituting P for x in M is N. By means of the pushing operation, we can define capture avoiding substitution without the need of renaming of binding variables.

$$\frac{\boldsymbol{r} : \texttt{varref}}{\boldsymbol{r}[\boldsymbol{P}/\boldsymbol{r}] = \boldsymbol{P}}\ \texttt{subst-hit} \qquad \frac{\ulcorner \# \boldsymbol{r} \urcorner\ \texttt{Covers}\ \boldsymbol{s}}{\ulcorner \# \boldsymbol{r} \urcorner[\boldsymbol{P}/\boldsymbol{s}] = \boldsymbol{r}}\ \texttt{subst-down}$$

$$\frac{\boldsymbol{s}\ \texttt{Covers}\ \boldsymbol{r}}{\boldsymbol{r}[\boldsymbol{P}/\boldsymbol{s}] = \boldsymbol{r}}\ \texttt{subst-keep} \qquad \frac{\boldsymbol{r}\ \texttt{NeqCore}\ \boldsymbol{s}}{\boldsymbol{r}[\boldsymbol{P}/\boldsymbol{s}] = \boldsymbol{r}}\ \texttt{subst-miss}$$

$$\frac{\boldsymbol{P} \uparrow \boldsymbol{x} = \boldsymbol{Q} \quad \boldsymbol{r} \uparrow \boldsymbol{x} = \boldsymbol{s} \quad \boldsymbol{M}[\boldsymbol{Q}/\boldsymbol{s}] = \boldsymbol{N}}{\ulcorner \lambda \boldsymbol{x}[\boldsymbol{M}] \urcorner[\boldsymbol{P}/\boldsymbol{r}] = \ulcorner \lambda \boldsymbol{x}[\boldsymbol{N}] \urcorner}\ \texttt{subst-abs}$$

$$\frac{\boldsymbol{M}[\boldsymbol{P}/\boldsymbol{r}] = \boldsymbol{Q} \quad \boldsymbol{N}[\boldsymbol{P}/\boldsymbol{r}] = \boldsymbol{R}}{\ulcorner \boldsymbol{M}(\boldsymbol{N}) \urcorner[\boldsymbol{P}/\boldsymbol{r}] = \ulcorner \boldsymbol{Q}(\boldsymbol{R}) \urcorner}\ \texttt{subst-app}$$

The systematic use of the lambda-bar operator $\natural$ is due originally to Berkling and Fehr [2]. Sato [11] independently introduced the same operator, but it was used only to substitute first-order terms. Our definition here is slightly simpler than [2] since we could avoid case analysis in the definitions of push and substitution for λ-abstracts.

We can now proceed to define the reduction rules of the λ-calculus. Here we only give the crucial rule of β-conversion.

$$\frac{\boldsymbol{M}\,\texttt{[}\boldsymbol{N}\,\texttt{/}\,\boldsymbol{x}\texttt{]} \texttt{ = } \boldsymbol{P}}{\ulcorner\lambda\boldsymbol{x}\,\texttt{[}\boldsymbol{M}\texttt{]}\,\texttt{(}\boldsymbol{N}\texttt{)}\urcorner \rightarrow \boldsymbol{P}}\ \texttt{beta}$$

4.3 α-equivalence

In this subsection we define α-equivalence relation on λ-terms. Our definition uses *context* which enable us to define α-equivalence without involving any notion of renaming of variables. Thus our definition is simpler and more natural than the traditional definition of α-equivalence.

Let Γ, Δ be sequences of variables and M, N be λ-terms. Below, we define the judgment: $\Gamma \vdash M \equiv \Delta \vdash N$, which we read M and N are α-equivalent relative to Γ and Δ.

$$\frac{\boldsymbol{\Gamma}\ \texttt{EqLen}\ \boldsymbol{\Delta} \quad \boldsymbol{x} : \texttt{variable} \quad \boldsymbol{y} : \texttt{variable}}{\ulcorner\boldsymbol{\Gamma}\texttt{, }\boldsymbol{x}\urcorner\vdash\boldsymbol{x} \equiv \ulcorner\boldsymbol{\Delta}\texttt{, }\boldsymbol{y}\urcorner\vdash\boldsymbol{y}}\ \texttt{alpha-bound}$$

$$\frac{\boldsymbol{\Gamma}\vdash\boldsymbol{r} \equiv \boldsymbol{\Delta}\vdash\boldsymbol{s} \quad \boldsymbol{r}\uparrow\boldsymbol{x} \texttt{ = } \boldsymbol{t} \quad \boldsymbol{s}\uparrow\boldsymbol{y} \texttt{ = } \boldsymbol{u}}{\ulcorner\boldsymbol{\Gamma}\texttt{, }\boldsymbol{x}\urcorner\vdash\boldsymbol{t} \equiv \ulcorner\boldsymbol{\Delta}\texttt{, }\boldsymbol{y}\urcorner\vdash\boldsymbol{u}}\ \texttt{alpha-push}$$

$$\frac{\boldsymbol{r} : \texttt{varref}}{\ulcorner\urcorner\vdash\boldsymbol{r} \equiv \ulcorner\urcorner\vdash\boldsymbol{r}}\ \texttt{alpha-free}$$

$$\frac{\ulcorner\boldsymbol{\Gamma}\texttt{, }\boldsymbol{x}\urcorner\vdash\boldsymbol{M} \equiv \ulcorner\boldsymbol{\Delta}\texttt{, }\boldsymbol{y}\urcorner\vdash\boldsymbol{N}}{\boldsymbol{\Gamma}\vdash\ulcorner\lambda\boldsymbol{x}\,\texttt{[}\boldsymbol{M}\texttt{]}\urcorner \equiv \boldsymbol{\Delta}\vdash\ulcorner\lambda\boldsymbol{y}\,\texttt{[}\boldsymbol{N}\texttt{]}\urcorner}\ \texttt{alpha-abs}$$

$$\frac{\boldsymbol{\Gamma}\vdash\boldsymbol{M} \equiv \boldsymbol{\Delta}\vdash\boldsymbol{P} \quad \boldsymbol{\Gamma}\vdash\boldsymbol{N} \equiv \boldsymbol{\Delta}\vdash\boldsymbol{Q}}{\boldsymbol{\Gamma}\vdash\ulcorner\boldsymbol{M}\texttt{(}\boldsymbol{N}\texttt{)}\urcorner \equiv \boldsymbol{\Delta}\vdash\ulcorner\boldsymbol{P}\texttt{(}\boldsymbol{Q}\texttt{)}\urcorner}\ \texttt{alpha-app}$$

We define two λ-terms M and N to be *α-equivalent* if the judgment $\ulcorner\urcorner \vdash M \equiv \ulcorner\urcorner \vdash N$ is derivable by using the above rules. It is now easy to verify that this notion of α-equivalence is indeed an equivalence relation on λ-terms.

The author learned by private communication that Per Martin-Löf had developed essentially the same method, and that Robert Staerk also had a similar idea and implemented the algorithm as part of the Minlog system [1] developed by Helmut Shwichtenberg's group.

5 Concluding Remarks

We have given an overview of the NF/CAL system. We believe that it is essential to have a basic structure of object expressions which can represent various mathematical and meta-mathematical entities required in mathematics and computer science. To this end, we have introduced a revised theory of expressions in which we have basic ground categories of $\mathsf{o}, \mathsf{j}, \mathsf{d}$ which respectively represent objects, judgments and derivations. The structure of categories is isomorphic to the simple type structure generated from these ground categories by the arrow type construction, and we have used higher-order categories as *arities* of constants and variables. The advantage of this approach is that we can syntactically sort out expressions according to their intended meanings, and thereby prevent users from writing inadvertently completely meaningless expressions. In this respect, we think that our theory of expressions should be extensible by allowing extension of ground categories depending on various applications.

References

1. H. Benl, U. Berger, M. Seisenberger, H. Schwichtenberg and W. Zuber, Proof theory at work: Program development in the Minlog sytem, in *Automated Deduction*, W. Bibel and P.H. Schmitt, eds., Vol II, Kluwer, 1998.
2. K.J. Berkling and E. Fehr, A Consistent Extension of the Lambda Calculus as a Base for Functional Programming Languages, *Information and Control*, **55**, pp. 89-101.
3. 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.
4. 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.
5. G. Frege, *The Basic Laws of Arithmetic*, University of California Press, 1964.
6. B. Nordström, K. Petersson and J.M. Smith, *Programming in Martin-Löf's Type Theory*, Clarendon Press, Oxford, 1990.
7. T. Nipkow, L.C. Paulson and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science, **2283**, Springer 2002.
8. W.V.O. Quine, *Mathematical Logic*, revised ed., Harvard University Press, 1951.
9. 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.
10. 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.
11. M. Sato, Theory of Symbolic Expressions II, *Publ. RIMS, Kyoto U.*, **21**, pp. 455-540, 1985.
12. M. Sato, A Simple Theory of Expressions, Judgments and Derivations, in Maher, M. J. ed., *ASIAN 2004*, Lecture Notes in Computer Science **3321**, pp. 437 – 451, Springer, 2004.