NF: A Natural Framework for Checking Proofs

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

May 22, 2005

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 judgmens and derivations are the two fundamental notions used in any mathematical activity. We have therefore developed a theory of judgments and derivations and desginged a framwork 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.

1 Introduction

The Curry-Howard isomorphism enables us to identify programs with proofs. Therefore, simply by developing a formal computer enviornment 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 [1], Isabelle [3], 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 developping our system are foundation and education. To achieve these objectives, our system is based on a theory of judgments and derivations introduced in [5]. The theory is later modified by changing the basic data structure of expressions. This new data structure of expressions was introduced in [6].

In this paper, we outline the current status of NF/CAL with examples.

2 Expressions

Proofs as well as judgments are lingustic 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 behing the following definition of expressions is the usage of *arity*. Namely, any variable or constant will have a fixed arity n. If a variable x has a positive arity 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 expression. 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*.

We define expressions slightly informally by taking the notion of α-equivalence for granted.

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 $\mathtt{(}x\mathtt{)}\,\mathtt{[}a\mathtt{]}$ 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.

2.1 Environments and isntantiation

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

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. $[\texttt{(}x\texttt{) [}a\texttt{]}]_\rho :\equiv \texttt{(}x\texttt{) [}[a]_\rho\texttt{]}$.

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.

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 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 can define judgements and derivations. We also have the notion of *derivation game*. Namely, a derivation roughly corresponds the notion of a theory, and each derivation game has a unique set of constants and inference rules which characterizes the game. Thus, the notion of derivation games is the key notion of natrual framework and this notion is based on more fundamental notions of *judgment* and *derivation*.

Here, we give a very simple derivation game Nat which defines natural numbers inductively as follows.

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

where the games has three constants `Nat`, `zero` and `succ` of arity 0, 0 and 2, respcetively. That the rule `succ` has arity 2 means that it is a rule (abstract) which accepts two expressions, namely, a natural number n and a derivation whose conclusion is the judgment $n : \mathtt{Nat} \Rightarrow \mathtt{s}(n) : \mathtt{Nat}$.

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

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

where D is a *derivation* which is obtained by combining the two infrence rules of the game, and which proves the *judgment* $\mathtt{s}(\mathtt{s}(\mathtt{0})) : \mathtt{Nat}$.

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

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

will be written as:

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

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 `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`.

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, *Isabelle/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, 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.