Theory of Judgments and Derivations

Dedicated to the memory of Professor Katuzi Ono

Masahiko SATO

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

Abstract. We propose a computational and logical framework NF (Natural Framework) which is suitable for presenting mathematics formally. Our framework is an extendable framework since it is *open-ended* both computationally and logically in the sense of Martin-Löf's theory or types. NF is developed in three steps. Firstly, we introduce a theory of expressions and schemata which is used to provide a universe for representing mathematical objects, in particular, judgments and derivations as well as other usual mathematical objects. Secondly, we develop a theory of judgments within the syntactic universe of expressions. Finally, we introduce the notions of derivation and derivation game and will show that we can develop mathematics as derivation games by regarding mathematics as an open-ended process of defining new concepts and deriving new judgments. Our theory is inspired by Martin-Löf's theory of expressions and Edinburgh LF, but conceptually much simpler. Our theory is also influenced by Gentzen's natural deduction systems.

1 Introduction

The continuous development of mathematics has been supported by the introduction of new *concepts*. For example, the concept *number* has been extended starting from *natural number*, through *integer*, *rational number*, *real number* up to the concept *complex number*. For each object s and a concept P we can form a *judgment* $s : P$ which asserts that the object s falls under the concept P.[1] As is customary in modern mathematics, we can assert a judgment when and only when we could derive the judgment, namely, when we could construct a *derivation* whose conclusion is the judgment. Here the derivation provides an *evidence* for the judgment. Firstly, the derivation makes the judgment evident to the very person who constructed the derivation because, supposedly, he has just constructed it by applying *derivation rules* correctly. Secondly, the derivation makes the judgment evident to any reader of the derivation, because[2] the

[1] See, e.g., Bell [7, page 9] and Frege [22].

[2] Here, we assume that the reader also accepts the rules as evident. So, if the reader is an intuitionist and the derivation contained an application of the law of the excluded middle for an undecidable proposition, then the derivation is not evident to the reader. But, even in such a case, the reader can check whether or not the derivation is a *correct* derivation obeying the rules of classical mathematics.

reader can read and check the *correctness* of the derivation with respect to a given set of rules and thereby he can *experience* the process of constructing the derivation for himself.

In order to make the above analysis valid, it is necessary that (1) the derivation is written in a common language shared by the writer and reader of the derivation and (2) a derivation rule in the derivation sends evident judgments to an evident judgment. In this paper we are mostly interested in the first condition of the above sentence, and we will provide a general and uniform framework NF (Natural Framework) for defining various formal systems conveniently. We will fulfill the first condition by realizing Kreisel's dictum (see [6, 31, 43–45, 80]) which asks the (primitive) recursive decidability of whether or not a given derivation is a correct derivation of a given judgment. The point is that the *correctness* of a derivation can be checked mechanically against a given set of formal rules while the *evidence* of the judgment proven by the derivation is determined semantically. We will achieve the decidability by defining the possible forms of *judgments* and *derivations* syntactically without referring to the meanings of judgments. Thus our method is along the line of Hilbert's program (see, e.g. [42, 44, 56, 77]), according to which formalistic notions of formulas and theorems must be defined and treated in a *finitistic* manner.

There are already numerous attempts to provide general frameworks for defining formal systems. Among these works, our work is greatly influenced by Martin-Löf's theory of expressions [55] and the Edinburgh Logical Framework [31].[3] Martin-Löf developed a theory of expressions which is used as an underlying language for representing his intuitionistic theory of types. Edinburgh LF was inspired by Martin-Löf's theory of expressions. It is a dependently typed λ-calculus and it can encode various logical systems by using the judgments-as-types principle. We will compare our work with these works in detail later. Here, we only mention that we have designed our framework NF so that it will become much simpler than these frameworks. We think the simplicity of the framework is crucial since we wish to provide a foundational framework for defining various formal systems on top of it.

Martin-Löf [46, 49] classified judgments into *categorical judgments* which are judgments made without hypotheses and *hypothetical judgments* which are judgments made under several hypotheses. We will write $H \Rightarrow J$ for the hypothetical judgment which asserts the judgment J under the hypothesis H. In Martin-Löf's notation, this judgment is written as $J\ (H)$. Thus, the hypothetical judgment $J\ (H_1, \ldots, H_n)$ in Martin-Löf's notation can be written as $H_1 \Rightarrow \cdots \Rightarrow H_n \Rightarrow J$ in our notation. A typical hypothetical judgment takes the form $J\ (x : A)$ where x is a variable and it means that for any x, if $x : A$ holds, then J also holds. Therefore, the variable x is universally quantified in the hypothetical judgment. To make this quantification explicit, we introduce judgments of the form $(x)\,[J]$ and call such judgments *universal judgments*. Using universal judgments, Martin-

[3] Throughout this paper, the Edinburgh Logical Framework will be called Edinburgh LF or, simply, LF.

Löf's hypothetical judgment $J\ (x : A)$ can be written as $(x)\,[x : A \Rightarrow J]$[4] which we will also abbreviate as $(x : A)\,[J]$.

In order to represent judgments and derivations formally, we first introduce a set E of symbolic expressions. The set E contains two infinite denumerable sets of variables and constants, and it is closed under the operation of forming a *pair* $\texttt{<}e\,|\,f\texttt{>}$ of two expressions e and f, and the operation of forming an *abstract* $(x)\,[e]$ which is obtained by binding a specified variable x in a given expression e. The set E is rich enough to represent judgments naturally. For instance, the universal judgment $(x)\,[x : A \Rightarrow J]$ can be written as the expression

$$(x)\,[\texttt{<}\Rightarrow, \texttt{<}\,:, x, A\texttt{>}, J\texttt{>}]$$

where '$\Rightarrow$' and ':' are constants, and $\texttt{<}\,:, x, A\texttt{>}$, for example, is a list which we can construct by the pairing operation in a usual manner. We will see that derivations can be represented naturally by expressions in E.

Formal systems are commonly presented by specifying derivation rule-schemata whose instances may be used to construct a new derivation D from already constructed derivations $D_1, \ldots, D_n$. Such a rule-schema can be displayed as a figure of the form:

$$\frac{H_1 \quad \cdots \quad H_n}{J}\ N(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$$

where N is the name of the rule schema (which is a constant), $J, H_1, \ldots, H_n$ are judgment-schemata whose free variables stand for unspecified expressions, and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ is a sequence of distinct variables containing all the free variables in $J, H_1, \ldots, H_n$. For example, we can define a formal system which defines the set of natural numbers by the two rule-schemata below.

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

We can represent this formal system by the following expression Nat.

$$\begin{aligned}
\mathsf{Nat} &:\equiv \texttt{<}\mathsf{zero}, \mathsf{succ}\texttt{>},\\
\mathsf{zero} &:\equiv \texttt{<0 : Nat}, \texttt{<zero>}, \texttt{<>>},\\
\mathsf{succ} &:\equiv \texttt{<s}(\boldsymbol{n}) : \texttt{Nat}, \texttt{<succ}, \boldsymbol{n}\texttt{>}, \texttt{<}\boldsymbol{n} : \texttt{Nat>>}.
\end{aligned}$$

In the formal system Nat we can construct the following derivation which shows that 2, which is represented by $\texttt{s(s(0))}$, is a natural number.

$$\cfrac{\cfrac{\cfrac{}{\texttt{0 : Nat}}\ \texttt{zero()}}{\texttt{s(0) : Nat}}\ \texttt{succ(0)}}{\texttt{s(s(0)) : Nat}}\ \texttt{succ(s(0))}$$

[4] In [49], Martin-Löf introduces universal judgments, which he calls general judgments. Edinburgh LF can represent general judgments as dependent types.

The rule-schemata for Nat are *elementary* in the sense that all the premises and conclusions are categorical judgments. Such elementary rule-schemata are often sufficient for presenting syntax and operational semantics of *computational* systems. We will need non-elementary rule-schemata to present *logical* systems in natural deduction style. Consider, for example, the following rule of mathematical induction:

$$\frac{P[0] \quad \begin{array}{c}(P[x])\\ P[\mathtt{s}(x)]\end{array}}{\forall x.P[x]}\ \mathtt{ind}$$

When this rule is applied, the specified occurrences of the assumption $P[x]$ is discharged, and the rule may be applied only if the variable x does not occur free in any assumptions, other than $P[x]$, on which $P[\mathtt{s}(x)]$ depends. In Edinburgh LF, this rule can be defined by declaring the constant IND as follows:

$$\begin{array}{l}\textsc{ind} : \Pi\phi : \iota \to o.\, \mathtt{True}(\phi 0) \to \\ \qquad\qquad (\Pi x : \iota.\mathtt{True}(\phi x) \to \mathtt{True}(\phi(\mathtt{s}(x))) \to \mathtt{True}(\forall(\lambda x : \iota.\phi x))).\end{array}$$

Thus, in LF, the induction rule is represented by a higher-order function that accepts proofs of the two premises $\mathtt{True}(\phi 0)$ and $\Pi x : \iota.\mathtt{True}(\phi x) \to \mathtt{True}(\phi(\mathtt{s}(x)))$, and returns a proof of the conclusion $\mathtt{True}(\forall(\lambda x : \iota.\phi x))$. The second premise is an LF encoding of a general judgment whose body is a hypothetical judgment.

In NF, the induction rule is given by the following rule-schema:

$$\frac{\boldsymbol{P}[\![0]\!] : \mathtt{True} \quad (x : \mathtt{Nat})\,[\boldsymbol{P}[\![x]\!] : \mathtt{True} \Rightarrow \boldsymbol{P}[\![\mathtt{s}(x)]\!] : \mathtt{True}]}{\forall(x : \mathtt{Nat})\,[\boldsymbol{P}[\![x]\!]] : \mathtt{True}}\ \mathtt{ind}(\boldsymbol{P})$$

In this rule-schema, $\boldsymbol{P}$ is a *schematic parameter* of the rule, and when we use the rule-schema, we must *instantiate* the rule-schema by assigning an abstract expression $e \in \mathsf{E}$ to $\boldsymbol{P}$. So, if we wish to prove, for instance, the judgment

$$\forall(x : \mathtt{Nat})\,[x + x = 2 \cdot x] : \mathtt{True}$$

by applying the $\mathtt{ind}$-rule, then we must assign the abstract $(x)\,[x + x = 2 \cdot x]$ to $\boldsymbol{P}$. In NF, in order to express such rule-schemata formally, we extend the set E of expressions to the set S of *schemata* by adding *schematic variables* which will be used as parameters of rule-schemata and *application schemata* which are of the form $\boldsymbol{X}[\![S]\!]$ where $\boldsymbol{X}$ is a schematic variable and S is a schema. Although elementary rule-schemata can be represented by expressions in E we use schematic variables so that rule-schemata can be presented uniformly as schemata.

In addition to the mechanism of instantiating rule-schemata, we need a general mechanism to generate valid *derivations* of judgments under a given set of rule-schemata. We will introduce the notion of *derivation game*[5] which provides

[5] We use the term 'game' to reflect the formalistic view of mathematics according to which mathematical theories are presented by spcifying the *rules* of the games (i.e., theories) which determine the derivable judgments of the theories.

such a mechanism. A derivation game G is simply a finite list of rule-schemata and hence it is an element in S. Relative to a derivation game G, the set of derivations valid in G is determined as a recursively enumerable subset of E and it is always decidable whether a given expression is a valid derivation in G or not. We can use derivation games to define various formal systems in a completely formal way.

The paper is organized as follows. In §2, we introduce a theory of expressions and schemata. In §3, we introduce three forms of judgments, namely, categorical, hypothetical and universal judgments syntactically. The set of judgments becomes a primitive recursive subset of E. We then define derivation games, which can be used to assign meanings to these judgments. In this way, one and the same judgment can have different meanings in different derivation games. NF is neutral semantically in this sense, and can define both intuitionistic and classical mathematics equally well. All the definitions and proofs in §§2–3 will be given in a finitistic fashion. Then in §§4–5, we give examples of defining computational and logical systems in NF. These examples will show the expressiveness as well as the naturalness of NF. In §6, we compare our work with related works, in particular, we compare NF with Edinburgh LF and also with the theory of expressions by Martin-Löf. Our theory of schemata is inspired not only by Martin-Löf's theory of expressions but also by Takeuti's definition of formulae in his generalized logic calculus GLC. We will describe this as well in §6. In §7, we conclude the paper by explaining metamathematical motivations for this work and suggesting possible future work.

2 Theory of expressions and schemata

We define the set E of *expressions* as a structure which is equipped with the following two operations. The first operation is *pair formation*. If e and f are expressions then we can form an expression $\mathtt{<}e\mathtt{|}f\mathtt{>}$ which is the pair of e and f. The second operation is *abstraction*. This operation abstracts away a specified variable x from a given expression e yielding an abstract of the form $\mathtt{(}x\mathtt{)[}e\mathtt{]}$.

We assume that we have a countably infinite set of atomic symbols which consists of two mutually disjoint infinite sets of *variables* and *constants*. Variables will also be called *object variables* to distinguish them from other kinds of variables we introduce later in this section. Furthermore, we assume that the set of object variables is a union of two infinite and mutually disjoint sets of *general variables* and *derivation variables*. Object variables will be designated by letters x, y, z and derivation variables will be designated by X, Y, Z, and constants will be designated by the letter c. We will also assume that strings of characters in `typewriter` font are included among constants and they are distinct if they are spelled differently.

Definition 1 (Context). *A sequence*

$$x_1, \ldots, x_n$$

of distinct object variables is called a context.

We will say that a variable x is *declared* in a context Γ if x occurs in Γ.

Definition 2 (Expression). *We define* expressions *relative to a context Γ as follows.*

1. Object variable. *If Γ is a context and x is declared in Γ, then x is an expression under Γ.*
2. Constant. *If Γ is a context and c is a constant, then c is an expression under Γ.*
3. Pair. *If Γ is a context and e, f are expressions under Γ, then $\texttt{<}e\,\texttt{|}\,f\texttt{>}$ is an expression under Γ.*
4. Abstract. *If Γ, x is a context and e is an expression under Γ, x, then $\texttt{(}x\texttt{)}\,\texttt{[}e\texttt{]}$ is an expression under Γ*

In the 4th clause of the above definition, the variable x which was (globally) declared in Γ, x becomes locally declared in the expression $\texttt{(}x\texttt{)}\,\texttt{[}e\texttt{]}$. The syntax $\texttt{(}x\texttt{)}\,\texttt{[}e\texttt{]}$ may look heavy compared to more conventional notations like $x.\ e$ or $(x)e$. However, we prefer this notation, since it clearly shows that the variable x in $\texttt{(}x\texttt{)}$ is a binding occurrence and the scope of the binder x is e.

We will identify two expressions e and f if they are defined in the same way except for the choices of variables, and we will write $e \equiv f$ in this case. Then, we can define the capture avoiding substitution operation on expressions as usual, and we will write $[x := d](e)$ for the result of substituting d for x in e. For an expression e, we can define the set $\mathrm{FV}(e)$ of *free variables* in e as usual and can verify that if e is an expression under Γ, then $\mathrm{FV}(e) \subseteq \Gamma$. An expression is *closed* if $\mathrm{FV}(e) = \emptyset$ or, equivalently, if it is an expression under the empty context.

As a notational convention, we will write

$$\texttt{<}e_1, e_2, \ldots, e_n\texttt{>} \quad \text{for} \quad \texttt{<}e_1\,\texttt{|}\,\texttt{<}e_2\,\texttt{|}\cdots\texttt{<}e_n\,\texttt{|}\,\texttt{nil}\texttt{>}\cdots\texttt{>}\texttt{>}.$$

Such an expression is called a *list* and we will define concatenation of two lists by:

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

We will extend this notational convention to schemata we define below. We will also identify a context $x_1, \ldots, x_n$ with the list $\texttt{<}x_1, \ldots, x_n\texttt{>}$.

We now generalize the notion of an expression to that of a *schema*. We will write S for the set of schemata. We first extend the atomic symbols by adding a countably infinite set V_s of *schematic variables*. Schematic variables are classified into two mutually disjoint subsets of *expression variables*[6] and *abstract variables*[7]. Schematic variables will be written by bold italic letters such as $\boldsymbol{z}$, $\boldsymbol{X}$ etc.

Definition 3 (Schema). *We define a* schema *relative to a context where, as before, a context is a sequence of distinct object variables.*

[6] So called since they can be instantiated to arbitrary expressions.
[7] So called since they can be instantiated to arbitrary abstracts.

1. Expression variable. *If Γ is a context and $\boldsymbol{z}$ is an expression variable, then $\boldsymbol{z}$ is a schema under Γ.*
2. Object variable. *If Γ is a context and x is declared in Γ, then x is a schema under Γ.*
3. Constant. *If Γ is a context and c is a constant, then c is a schema under Γ.*
4. Pair. *If S and T are schemata under Γ, then $\texttt{<}S\,|\,T\texttt{>}$ is a schema under Γ.*
5. Abstract. *If S is a schema under $\Gamma \oplus \texttt{<}x\texttt{>}$, then $\texttt{(}x\texttt{)}\,\texttt{[}S\texttt{]}$ is a schema under Γ*
6. Application. *If $\boldsymbol{X}$ is an abstract variable and S is a schema under Γ, then $\boldsymbol{X}\,[\![S]\!]$ is a schema under Γ.*

Any expression is a schema because of clauses 2 – 5. Here again we will identify two schemata S and T if they are defined in the same way except for the choices of binding variables in clause 5. In this case, we will write $S \equiv T$ and say that S and T are *definitionally equal*. It is clear that if $S \equiv T$ and S is an expression, then T is also an expression. A schema is *closed* if it is a schema under the empty environment. A closed schema may contain schematic variables but may not contain free object variables. Schematic variables will be used as *parameters* of rule-schemata and there is no need to abstract over schematic variables. This is the reason why we do not have a clause to bind schematic variables in the above definition. We now define a very important operation of *instantiation*.

Definition 4 (Environment and Instance). *A function*

$$\rho : \mathsf{V}_s \to \mathsf{E}$$

is an environment *if $\rho(\boldsymbol{X})$ is an abstract for any abstract variable $\boldsymbol{X}$ and $\rho(\boldsymbol{z}) \equiv \texttt{(}x\texttt{)}\,\texttt{[}x\texttt{]}$ for all but finitely many schematic variables $\boldsymbol{z}$. An environment ρ can be extended to a function $[\![\,\cdot\,]\!]_\rho : \mathsf{S} \to \mathsf{E}$ as follows.*

1. Expression variable. $[\![\boldsymbol{z}]\!]_\rho :\equiv \rho(\boldsymbol{z})$.
2. Object variable. $[\![x]\!]_\rho :\equiv x$.
3. Constant. $[\![c]\!]_\rho :\equiv c$.
4. Pair. $[\![\texttt{<}S\,|\,T\texttt{>}]\!]_\rho :\equiv \texttt{<}[\![S]\!]_\rho\,|\,[\![T]\!]_\rho\texttt{>}$.
5. Abstract. $[\![\texttt{(}x\texttt{)}\,\texttt{[}S\texttt{]}]\!]_\rho :\equiv \texttt{(}x\texttt{)}\,\texttt{[}[\![S]\!]_\rho\texttt{]}$. *(We assume, without loss of generality, that for any $\boldsymbol{z}$ occurring in S, x is not free in $\rho(\boldsymbol{z})$.)*
6. Application. $[\![\boldsymbol{X}\,[\![S]\!]\,]\!]_\rho \equiv [x := [\![S]\!]_\rho](e)$, *where* $\rho(\boldsymbol{X}) \equiv \texttt{(}x\texttt{)}\,\texttt{[}e\texttt{]}$.

The expression $[\![S]\!]_\rho$ is called the instantiation of S by ρ *or an* instance of S.

We see from clause 6 that an *application* $\boldsymbol{X}\,[\![S]\!]$ is instantiated by a call-by-value manner.

3 Judgments and Derivation Games

Based on the theory of expressions and schemata, we can describe the syntax of *judgment* as follows. In this way, we can characterize the possible *form* a judgment can take. A *general variable context* is a context such that all the variables declared in it are general variables. In the following definition, Γ stands for an arbitrary general variable context.

Definition 5 (Judgment). Judgments *are defined relative to general variable contexts and thereby classified into the following three types.*

1. Categorical Judgment. *If s and P are expressions under Γ, then*

$$s : P$$

   is a judgment under Γ.[8] *In this case, s (P, resp.) is called the* object part *(*concept part*, resp.) of the categorical judgment.*
2. Hypothetical Judgment. *If H and J are judgments under Γ, then*

$$H \Rightarrow J$$

   is a judgment under Γ.[9]
3. Universal Judgment. *If x is a general variable not declared in Γ, and J is an expression under $\Gamma \oplus \texttt{<}x\texttt{>}$, then*

$$\texttt{(}x\texttt{)}\,\texttt{[}J\texttt{]}$$

   is a judgment under Γ.

One can verify that if J is a judgment under a general variable context Γ, then J is an expression under Γ. We can explain the meanings of judgments informally as follows. A categorical judgment $s : P$ means that the object s falls under the *concept* P. When P is a constant, it is possible to regard P as a *name* of the concept which is characterized by those s for which the judgment $s : P$ holds. For example, if `Nat` is a name given to the concept of the natural number, then `0` : `Nat` means that `0` is a natural number. A hypothetical judgment $H \Rightarrow J$ means that we can derive the judgment J whenever H is derivable. A universal judgment of the form $\texttt{(}x\texttt{)}\,\texttt{[}J\texttt{]}$ means that we can derive the judgment $[x := e](J)$ for any expression e. Since we will often consider judgments of the form $\texttt{(}x\texttt{)}\,\texttt{[}x : P \Rightarrow J\texttt{]}$, we will abbreviate this form by $\texttt{(}x : P\texttt{)}\,\texttt{[}J\texttt{]}$.

Given a context Γ, we put J_Γ to be the set of judgments under Γ. It is easy to see that J_Γ is a primitive recursive subset of E.

Based on the above informal explanation of the intended meanings of judgments, we introduce the notion of derivation game.

[8] This is an abbreviation of the expression: $\texttt{<:}, s, P\texttt{>}$.
[9] This is an abbreviation of the expression: $\texttt{<}\Rightarrow, H, J\texttt{>}$.

Definition 6 (Rule-Schema). *A closed schema of the form:*

$$<\mathtt{RS}, J, <c, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_m>, <H_1, \ldots, H_n>>$$

is a rule-schema *if c is a constant (called the* name *of the rule-schema) and $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m$ is a list (called the* parameter list *of the rule-schema) of distinct schematic variables exactly covering all the schematic variables in the schemata $J, H_1, \ldots, H_n$.*

A rule-schema can be displayed as a figure of the following form:

$$\frac{H_1 \quad \cdots \quad H_n}{J}\ R(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m)$$

We will also use the following Prolog-like notation for the same rule-schema:

$$J \ \texttt{:-}_{\,R(\boldsymbol{z}_1,\ldots,\boldsymbol{z}_n)}\ H_1, \ldots, H_n.$$

We will often omit the rule name and its parameter list in these notations; or we keep the rule name only and omit the parameter list; or we keep the rule name and the list of abstract variables and omit the expression variables from the parameter list. If $J, H_1, \ldots, H_n$ are all of the form $s : c$ where $s \in \mathsf{S}$ and c is a constant, then the rule-schema is said to be an *elementary rule schema*.

Definition 7 (Derivation Game). *A* derivation game *is a schema of the form $<R_1, \ldots, R_n>$ where each R_i is a rule-schema.*

A derivation game is a closed schema since a rule-schema is closed. We write G for the set of derivation games. The set G is a primitive recursive subset of S. A derivation game is *elementary* if each rule-schema of the game is elementary.

A derivation game G determines a set $\mathsf{D}(G)$ of expressions called the set of *G-derivations*, which is the set of derivations derivable in G. To define $\mathsf{D}(G)$, we must first define the notion of a derivation context.

Definition 8 (Derivation Context). *We define a* derivation context *Γ together with its* general variable part $\mathrm{GV}(\Gamma)$ *which is a context in the sense of Definition 1.*

1. Empty context. *The empty list $<>$ is a derivation context and its general variable part is $<>$.*
2. General variable declaration. *If Γ is a derivation context, and x is a general variable not declared in Γ, then $\Gamma \oplus <x>$ is a derivation context and its general variable part is $\mathrm{GV}(\Gamma) \oplus <x>$.*
3. Derivation variable declaration. *If Γ is a derivation context, J is a judgment under $\mathrm{GV}(\Gamma)$, and X is a derivation variable not declared in Γ, then $\Gamma \oplus <X :: J$[10]$>$ is a derivation context and its general variable part is $\mathrm{GV}(\Gamma)$.*

[10] $X :: J$ is an abbreviation of $<::, X, J>$.

Let Γ be a derivation context. An environment ρ is a *Γ-environment* if $\rho(\boldsymbol{z})$ is an expression under $\mathrm{GV}(\Gamma)$ for any schematic variable $\boldsymbol{z}$.

Definition 9 (G-derivation). *We define a G-*derivation *(also called a* derivation in G*) 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 an expression under* $\mathrm{GV}(\Gamma)$.

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. *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 an instantiation of a rule-schema in G by a Γ-environment and J is a categorical judgment under $\mathrm{GV}(\Gamma)$*, then*
$$\frac{D_1 \quad \cdots \quad D_n}{J}\ R(e_1, \ldots, e_m)^{11}$$
is a G-derivation and its conclusion is J.
3. Hypothetical derivation. *If D is a G-derivation under $\Gamma \oplus$ <$X :: H$> and its conclusion is J, then*
$$\texttt{(}X :: H\texttt{)[}D\texttt{]}^{12}$$
is a G-derivation under Γ and its conclusion is $H \Rightarrow J^{13}$.
4. Universal derivation. *If D is a G-derivation under $\Gamma \oplus$<x> and its conclusion is J, then*
$$\texttt{(}x\texttt{)[}D\texttt{]}$$
is a G-derivation under Γ and its conclusion is $\texttt{(}x\texttt{)[}J\texttt{]}$.

We will write
$$\Gamma \vdash_G D :: J$$
if D is a derivation in G under Γ whose conclusion is J. For example, for the game Nat we gave in §1 and for the derivation D given there, we have
$$\vdash_{\mathsf{Nat}} D :: \texttt{s(s(0))} : \texttt{Nat}.$$

[12] This is an abbreviation of the expression: <HD, H, $\texttt{(}X\texttt{)[}D\texttt{]}$>.

[13] We sometimes write the hypothetical judgment $H \Rightarrow J$ also by $\texttt{(}X :: H\texttt{)[}J\texttt{]}$ to emphasize the intended meaning of the judgment; namely, we can derive J if we have a derivation X of H. This notation also matches well with the notation for the hypothetical derivation we introduced here.

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:

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

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

There are certain syntactic structures that are common among G-derivations for all derivation games G. To capture such structures, we introduce the notion of derivation.

Definition 10 (Derivation). *We define a* derivation *and its* conclusion *relative to a derivation context as follows. In the following definition, Γ stands for an arbitrary derivation context.*

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

$$X$$

is a derivation under Γ and its conclusion is H.

2. Composition. *If $D_1, \ldots, D_n$ are derivations under Γ, R is a constant, $e_1, \ldots, e_m$ are expressions under Γ, and J is a categorical judgment under* $\mathrm{GV}(\Gamma)$*, then*

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

is a derivation under Γ and its conclusion is J.

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

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

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

We will write

$$\Gamma \vdash D :: J$$

if Γ is a derivation context and D is a derivation under Γ whose conclusion is J. We have the following theorem which characterizes derivations in terms of G-derivations.

Theorem 1 (Characterization of Derivations). *For any expressions Γ, D and J, we have $\Gamma \vdash D :: J$ if and only if $\Gamma \vdash_G D :: J$ for some derivation game G.*

Proof. Since *if-part* is trivial, we only show the *only-if-part* by induction on the construction of D. The only non-trivial case is the case where D is a derivation under Γ of the form:

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

We let J_i be the conclusion of D_i for all i such that $1 \leq i \leq n$. In this case, we may assume by induction hypothesis, that we have derivation games G_i $(1 \leq i \leq n)$ such that $\Gamma \vdash_{G_i} D_i :: J_i$ $(1 \leq i \leq n)$. Then, we can obtain the derivation game with the required property by putting:

$$G :\equiv G_1 \oplus \cdots \oplus G_n \oplus \mathtt{<<RS,<}R, e_1, \ldots, e_m\mathtt{>,<}J_1, \ldots, J_n\mathtt{>>>}.$$

Note that the last rule-schema of G is actually an expression and not a proper schema. □

Given a context Γ, we let D_Γ be the set of derivations under Γ. For any $e \in \mathsf{E}$, we can decide whether $\Gamma \vdash e :: J$ for some J by induction on the construction of $e \in \mathsf{E}$. So, we see that D_Γ is a primitive recursive subset of E, and there is a primitive recursive function:

$$\mathrm{conc}_\Gamma : \mathsf{D}_\Gamma \to \mathsf{J}_\Gamma$$

such that $\Gamma \vdash_G D :: \mathrm{conc}_\Gamma(D)$ holds for all $D \in \mathsf{D}_\Gamma$. Then, by the characterization theorem of derivations, we see that if $\Gamma \vdash_G D :: J$, then $D \in \mathsf{D}_\Gamma$ and $J \equiv \mathrm{conc}_\Gamma(D) \in \mathsf{J}_\Gamma$.

Theorem 2 (Decidability). *If $G \in \mathsf{S}$ and $\Gamma, D, J \in \mathsf{E}$, then it is primitive recursively decidable whether $\Gamma \vdash_G D :: J$ or not.*

Proof. By induction on the construction of D as an element of E. □

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.*

Proof. Suppose that an element $e \in \mathsf{E}$ is given. If $e \notin \mathsf{D}_{<>}$ then, by the characterization theorem, $e \notin \mathsf{D}(G)$. If $e \in \mathsf{D}_{<>}$, then by the Decidability Theorem 2, we can decide $e \in \mathsf{D}(G)$ or not by checking $<> \vdash_G e :: \mathrm{conc}_{<>}(e)$ or not. $\square$

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 Γ.*

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 $\Gamma \oplus <X :: H> \oplus \Gamma' \vdash_G D :: J$ and $\Gamma \vdash_G D' :: H$, then $\Gamma \oplus \Gamma' \vdash_G [X := D'](D) :: J$.*
5. *Substitution for general variable. If $\Gamma \oplus <x> \oplus \Gamma' \vdash_G D :: J$, and $e \in \mathsf{E}$, then $\Gamma \oplus [x := e](\Gamma') \vdash_G [x := e](D) :: [x := e](J)$.*
6. *Exchange. If $\Gamma \oplus <e, f> \oplus \Gamma' \vdash_G D :: J$, and $\Gamma \oplus <f, e> \oplus \Gamma'$ is a derivation context, then $\Gamma \oplus <f, e> \oplus \Gamma' \vdash_G D :: J$.*

These basic properties of derivations and G-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) [73] which realizes this idea.

4 Computational Aspects of NF

We show the expressiveness of elementary derivation games by giving a complete description of pure Lisp as described in McCarthy [51] including its syntax and operational semantics. By the following examples, we can see that elementary derivation games contain Horn clause logic and Smullyan's elementary formal systems [79] as special cases.

4.1 Syntax of pure Lisp expressions

The syntax of pure Lisp expressions is extremely simple owing to its simple and elegant structures. We will define pure Lisp expressions formally by the derivation game PureLispExp which is a list consisting of the following 42 elementary rule-schemes.[14] The derivation game PureLispExp is defined as the concatenation of five games:

$$\mathsf{PureLispExp} :\equiv \mathsf{Alpha} \oplus \mathsf{Num} \oplus \mathsf{Char} \oplus \mathsf{Atom} \oplus \mathsf{Sexp}.$$

Lisp's symbolic expressions (`Sexp`) are constructed from atomic symbols (`Atom`) by means of the *cons* operation which constructs a cons pair of `Sexp`s out of two already constructed `Sexp`s. It should be clear that our expressions E are inspired from Lisp's symbolic expressions.

Lisp's atomic symbols are constructed by upper case alphabetic characters (`Alpha`) and numeric characters (`Num`). Therefore, we define `Alpha` and `Num` as follows.

Alpha

$$\frac{}{\texttt{A : Alpha}} \qquad \frac{}{\texttt{B : Alpha}} \qquad \cdots \qquad \frac{}{\texttt{Z : Alpha}}$$

Num

$$\frac{}{\texttt{0 : Num}} \qquad \frac{}{\texttt{1 : Num}} \qquad \cdots \qquad \frac{}{\texttt{9 : Num}}$$

Char We define a character as either an `Alpha` or a `Num`:

$$\frac{\boldsymbol{c} : \texttt{Alpha}}{\boldsymbol{c} : \texttt{Char}} \qquad \frac{\boldsymbol{c} : \texttt{Num}}{\boldsymbol{c} : \texttt{Char}}$$

These definitions are clearly equivalent to the following definitions by BNF notation.

$$\begin{aligned}
\texttt{<Alpha>} &::= \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{G} \mid \texttt{H} \mid \texttt{I} \mid \texttt{J} \mid \texttt{K} \mid \texttt{L} \mid \texttt{M} \mid \\
&\quad\;\; \texttt{N} \mid \texttt{O} \mid \texttt{P} \mid \texttt{Q} \mid \texttt{R} \mid \texttt{S} \mid \texttt{T} \mid \texttt{U} \mid \texttt{V} \mid \texttt{W} \mid \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \\
\texttt{<Num>} &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\texttt{<Char>} &::= \texttt{<Alpha>} \mid \texttt{<Num>}
\end{aligned}$$

Atom An atomic symbol is a sequence of `Char`s whose first character is an `Alpha`:

$$\frac{\boldsymbol{c} : \texttt{Alpha}}{\boldsymbol{c} : \texttt{Atom}} \qquad \frac{\boldsymbol{a} : \texttt{Atom} \quad \boldsymbol{c} : \texttt{Char}}{\texttt{<}\boldsymbol{a}\,|\,\boldsymbol{c}\texttt{>} : \texttt{Atom}}$$

For example, we have

$$\vdash_{\mathsf{PureLispExp}} \texttt{<E|Q>} : \texttt{Atom}$$

and we can convince ourself of this fact by looking at the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\texttt{E : Alpha}}}{\texttt{E : Atom}} \quad \dfrac{\dfrac{}{\texttt{Q : Alpha}}}{\texttt{Q : Char}}}{\texttt{<E|Q> : Atom}}$$

For the sake of readability, we will write EQ for `<E|Q>`, and will use the same convention for other `Atom`s. For example, CAR stands for `<<C|A>|R>`.

[14] Among the 42 rules, 36 are rules for 36 characters used in pure Lisp.

Sexp Now, we can define Lisp's symbolic expressions. In the following definition, we follow Lisp's convention and write $\texttt{(}s\,\texttt{.}\,t\texttt{)}$ for $\texttt{<cons}, s, t\texttt{>}$. We will also use the *list notation* which abbreviates, e.g., $\texttt{(}s\ \texttt{.}\ \texttt{(}t\ \texttt{.}\ \texttt{(}u\ \texttt{.}\ \mathsf{NIL}\texttt{)))}$ to $\texttt{(}s\ t\ u\texttt{)}$[15].

$$\frac{\boldsymbol{a} : \texttt{Atom}}{\boldsymbol{a} : \texttt{Sexp}} \qquad \frac{\boldsymbol{s} : \texttt{Sexp} \quad \boldsymbol{t} : \texttt{Sexp}}{\texttt{<cons}, \boldsymbol{s}, \boldsymbol{t}\texttt{>} : \texttt{Sexp}}$$

4.2 Equality on Lisp atoms

Two $\texttt{Atoms}$ a and b are *equal* if they are constructed in the same way by means of the rule-schemes given in the game PureLispExp. We can decide the equality as follows. Since we know that a and b are $\texttt{Atoms}$, we have derivations D_a and D_b such that:

$$\vdash_{\mathsf{PureLispExp}} D_a :: a : \texttt{Atom} \quad \text{and} \quad \vdash_{\mathsf{PureLispExp}} D_b :: b : \texttt{Atom}.$$

So, we can decide the equality by deciding the definitional equality of D_a and D_b. We can internalize the process of deciding $D_a \equiv D_b$ as follows. We first define games to decide equality of two characters and use them to decide equality of two atoms.

In our framework, an n-ary relation R among expressions $e_1, \ldots, e_n$ can be represented by the categorical judgment $\texttt{<}e_1, \ldots, e_n\texttt{>} : R$. We will also write this judgment as $R(e_1, \ldots, e_n)$ which is a common notation for asserting that the relation R holds for $e_1, \ldots, e_n$.

EqAlpha Equality of two $\texttt{Alpha}$s is defined by the following 26 rules:

$$\frac{}{\texttt{EqAlpha(A,A)}} \qquad \frac{}{\texttt{EqAlpha(B,B)}} \qquad \cdots \qquad \frac{}{\texttt{EqAlpha(Z,Z)}}$$

NeqAlpha Inequality of two $\texttt{Alpha}$s is defined by the following 650 rules:

$$\frac{}{\texttt{NeqAlpha(A,B)}} \qquad \frac{}{\texttt{NeqAlpha(A,C)}} \qquad \cdots \qquad \frac{}{\texttt{NeqAlpha(Z,Y)}}$$

EqNum Equality of two $\texttt{Num}$s is defined by the following 10 rules:

$$\frac{}{\texttt{EqNum(0,0)}} \qquad \frac{}{\texttt{EqNum(1,1)}} \qquad \cdots \qquad \frac{}{\texttt{EqNum(9,9)}}$$

NeqNum Inequality of two $\texttt{Num}$s is defined by the following 90 rules:

$$\frac{}{\texttt{NeqNum(0,1)}} \qquad \frac{}{\texttt{NeqNum(0,2)}} \qquad \cdots \qquad \frac{}{\texttt{NeqNum(9,8)}}$$

[15] In [51] it is written as $\texttt{(}s, t, u\texttt{)}$, but, instead, we will use the now common notation.

EqChar Equality of two characters is defined as follows.

$$\frac{\texttt{EqAlpha}(\boldsymbol{c},\boldsymbol{d})}{\texttt{EqChar}(\boldsymbol{c},\boldsymbol{d})} \qquad \frac{\texttt{EqNum}(\boldsymbol{c},\boldsymbol{d})}{\texttt{EqChar}(\boldsymbol{c},\boldsymbol{d})}$$

NeqChar Inequality of two characters is defined as follows.

$$\frac{\texttt{NeqAlpha}(\boldsymbol{c},\boldsymbol{d})}{\texttt{NeqChar}(\boldsymbol{c},\boldsymbol{d})} \qquad \frac{\texttt{NeqNum}(\boldsymbol{c},\boldsymbol{d})}{\texttt{NeqChar}(\boldsymbol{c},\boldsymbol{d})} \qquad \frac{\boldsymbol{c} : \texttt{Alpha} \quad \boldsymbol{d} : \texttt{Num}}{\texttt{NeqChar}(\boldsymbol{c},\boldsymbol{d})} \qquad \frac{\boldsymbol{c} : \texttt{Num} \quad \boldsymbol{d} : \texttt{Alpha}}{\texttt{NeqChar}(\boldsymbol{c},\boldsymbol{d})}$$

EqAtom We can define two equal atoms as follows:

$$\frac{\texttt{EqAlpha}(\boldsymbol{c},\boldsymbol{d})}{\texttt{EqAtom}(\boldsymbol{c},\boldsymbol{d})} \qquad \frac{\texttt{EqAtom}(\boldsymbol{a},\boldsymbol{b}) \quad \texttt{EqChar}(\boldsymbol{c},\boldsymbol{d})}{\texttt{EqAtom}(\texttt{<}\boldsymbol{a}\,|\,\boldsymbol{c}\texttt{>},\texttt{<}\boldsymbol{b}\,|\,\boldsymbol{d}\texttt{>})}$$

NeqAtom Inequality of two atoms is defined as follows.

$$\frac{\texttt{NeqAlpha}(\boldsymbol{c},\boldsymbol{d})}{\texttt{NeqAtom}(\boldsymbol{c},\boldsymbol{d})}$$

$$\frac{\boldsymbol{a} : \texttt{Atom} \quad \boldsymbol{c} : \texttt{Char} \quad \boldsymbol{d} : \texttt{Alpha}}{\texttt{NeqAtom}(\texttt{<}\boldsymbol{a}\,|\,\boldsymbol{c}\texttt{>},\boldsymbol{d})} \qquad \frac{\boldsymbol{c} : \texttt{Alpha} \quad \boldsymbol{b} : \texttt{Atom} \quad \boldsymbol{d} : \texttt{Char}}{\texttt{NeqAtom}(\boldsymbol{c},\texttt{<}\boldsymbol{b}\,|\,\boldsymbol{d}\texttt{>})}$$

$$\frac{\texttt{NeqAtom}(\boldsymbol{a},\boldsymbol{b}) \quad \boldsymbol{c} : \texttt{Char} \quad \boldsymbol{d} : \texttt{Char}}{\texttt{NeqAtom}(\texttt{<}\boldsymbol{a}\,|\,\boldsymbol{c}\texttt{>},\texttt{<}\boldsymbol{b}\,|\,\boldsymbol{d}\texttt{>})} \qquad \frac{\texttt{EqAtom}(\boldsymbol{a},\boldsymbol{b}) \quad \texttt{NeqChar}(\boldsymbol{c},\boldsymbol{d})}{\texttt{NeqAtom}(\texttt{<}\boldsymbol{a}\,|\,\boldsymbol{c}\texttt{>},\texttt{<}\boldsymbol{b}\,|\,\boldsymbol{d}\texttt{>})}$$

4.3 Lisp forms

In pure Lisp, forms are expressions that may be evaluated under a given environment, and they are syntactically represented by the so-called *meta-expressions.* Here, we define forms directly as `Sexps` without using meta-expressions.

Form Lisp forms consist of constants, variables, five forms for applications of the five Lisp primitive functions, function applications and conditional forms.

$\boldsymbol{e}$: `Form` :- $\boldsymbol{e}$: `Const`.
$\boldsymbol{e}$: `Form` :- $\boldsymbol{e}$: `Var`.
$\boldsymbol{e}$: `Form` :- $\boldsymbol{e}$: `ConsF`.
$\boldsymbol{e}$: `Form` :- $\boldsymbol{e}$: `CarF`.
$\boldsymbol{e}$: `Form` :- $\boldsymbol{e}$: `CdrF`.
$\boldsymbol{e}$: `Form` :- $\boldsymbol{e}$: `AtomF`.
$\boldsymbol{e}$: `Form` :- $\boldsymbol{e}$: `EqF`.
$\boldsymbol{e}$: `Form` :- $\boldsymbol{e}$: `AppF`.
$\boldsymbol{e}$: `Form` :- $\boldsymbol{e}$: `Cond`.

Const

$(\mathsf{QUOTE}\ \boldsymbol{s}) : \mathtt{Const}$:- $\boldsymbol{s} : \mathtt{Sexp}$.

Var

$\boldsymbol{s} : \mathtt{Var}$:- $\boldsymbol{s} : \mathtt{Atom}$.

ConsF, CarF, CdrF, AtomF and EqF These forms are evaluated by the Lisp primitive functions *cons*, *car*, *cdr*, *atom* and *eq*, respectively.

$(\mathsf{CONS}\ \boldsymbol{e1}\ \boldsymbol{e2}) : \mathtt{ConsF}$:- $\boldsymbol{e1} : \mathtt{Form}, \boldsymbol{e2} : \mathtt{Form}$.

$(\mathsf{CAR}\ \boldsymbol{e}) : \mathtt{CarF}$:- $\boldsymbol{e} : \mathtt{Form}$.

$(\mathsf{CDR}\ \boldsymbol{e}) : \mathtt{CdrF}$:- $\boldsymbol{e} : \mathtt{Form}$.

$(\mathsf{ATOM}\ \boldsymbol{e}) : \mathtt{AtomF}$:- $\boldsymbol{e} : \mathtt{Form}$.

$(\mathsf{EQ}\ \boldsymbol{e1}\ \boldsymbol{e2}) : \mathtt{EqF}$:- $\boldsymbol{e1} : \mathtt{Form}, \boldsymbol{e2} : \mathtt{Form}$.

AppF, Fun, FormList and VarList The predicate `AppF` checks if a given `Sexp` is of the form of a function applied to a list of forms.

$(\boldsymbol{f}\ .\ \boldsymbol{E}) : \mathtt{AppF}$:- $\boldsymbol{f} : \mathtt{Fun}, \boldsymbol{E} : \mathtt{FormList}$.

$\boldsymbol{f} : \mathtt{Fun}$:- $\boldsymbol{f} : \mathtt{Var}, \mathtt{NeqAtom}(\boldsymbol{f}, \mathsf{QUOTE}), \mathtt{NeqAtom}(\boldsymbol{f}, \mathsf{CONS})$,
$\mathtt{NeqAtom}(\boldsymbol{f}, \mathsf{CAR}), \mathtt{NeqAtom}(\boldsymbol{f}, \mathsf{CDR}), \mathtt{NeqAtom}(\boldsymbol{f}, \mathsf{ATOM}), \mathtt{NeqAtom}(\boldsymbol{f}, \mathsf{EQ})$.
$(\mathsf{LAMBDA}\ \boldsymbol{X}\ \boldsymbol{e}) : \mathtt{Fun}$:- $\boldsymbol{X} : \mathtt{VarList}, \boldsymbol{e} : \mathtt{Form}$.
$(\mathsf{LABEL}\ \boldsymbol{x}\ \boldsymbol{f}) : \mathtt{Fun}$:- $\boldsymbol{x} : \mathtt{Var}, \boldsymbol{f} : \mathtt{Fun}$.

$() : \mathtt{FormList}$:- .
$(\boldsymbol{e}\ .\ \boldsymbol{E}) : \mathtt{FormList}$:- $\boldsymbol{e} : \mathtt{Var}, \boldsymbol{E} : \mathtt{FormList}$.

$() : \mathtt{VarList}$:- .
$(\boldsymbol{x}\ .\ \boldsymbol{X}) : \mathtt{VarList}$:- $\boldsymbol{x} : \mathtt{Var}, \boldsymbol{X} : \mathtt{VarList}$.

Cond and CList

$(\mathsf{COND}\ .\ \boldsymbol{L}) : \mathtt{Cond}$:- $\boldsymbol{L} : \mathtt{CList}$.

$() : \mathtt{CList}$:- .
$((\boldsymbol{c}\ \boldsymbol{e})\ .\ \boldsymbol{L}) : \mathtt{CList}$:- $\boldsymbol{c} : \mathtt{Form}, \boldsymbol{e} : \mathtt{Form}, \boldsymbol{L} : \mathtt{CList}$.

4.4 Evaluation of Lisp forms

We are ready to define operational semantics of pure Lisp. We define the semantics as a partial function

$$\mathsf{Eval} : \mathsf{Form} \times \mathsf{Env} \to \mathsf{Sexp}$$

where Env realizes an environment as a list of variable-value pair. The overall evaluation strategy is call-by-value.

Eval and Ev The categorical judgment $\texttt{Eval}(e, a, v)$ means that the form e evaluated under the environment a has value v.

$\texttt{Eval}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{e} : \texttt{Form}, \boldsymbol{a} : \texttt{Env}, \texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$.

$\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{e} : \texttt{Const}, \texttt{EvConst}(\boldsymbol{e}, \boldsymbol{v})$.
$\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{e} : \texttt{Var}, \texttt{EvVar}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$.
$\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{e} : \texttt{ConsF}, \texttt{EvCons}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$.
$\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{e} : \texttt{CarF}, \texttt{EvCar}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$.
$\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{e} : \texttt{CdrF}, \texttt{EvCdr}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$.
$\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{e} : \texttt{AtomF}, \texttt{EvAtom}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$.
$\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{e} : \texttt{EqF}, \texttt{EvEq}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$.
$\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{e} : \texttt{AppF}, \texttt{EvAppF}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$.
$\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{e} : \texttt{Cond}, \texttt{EvCondF}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$.

Env An environment is a list of variable-value pairs known as an *association list*.

$\texttt{()} : \texttt{Env}$:- .
$\texttt{((}\boldsymbol{x}\ .\ \boldsymbol{v}\texttt{)}\ .\ \boldsymbol{a}\texttt{)} : \texttt{Env}$:- $\boldsymbol{x} : \texttt{Var}, \boldsymbol{v} : \texttt{Sexp}, \boldsymbol{a} : \texttt{Env}$.

$\mathsf{EvConst}$

$\texttt{EvConst}(\texttt{(QUOTE } \boldsymbol{v}\texttt{)}, \boldsymbol{v})$:- .

EvVar and Assoc The value of a variable is computed by looking up the environment using the function $\texttt{Assoc}$.

$\texttt{EvVar}(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{v})$:- $\texttt{Assoc}(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{v})$.

$\texttt{Assoc}(\boldsymbol{x}, \texttt{((}\boldsymbol{x}\ .\ \boldsymbol{v}\texttt{)}\ .\ \boldsymbol{a}\texttt{)}, \boldsymbol{v})$.
$\texttt{Assoc}(\boldsymbol{x}, \texttt{((}\boldsymbol{y}\ .\ \boldsymbol{w}\texttt{)}\ .\ \boldsymbol{a}\texttt{)}, \boldsymbol{v})$:- $\texttt{NeqAtom}(\boldsymbol{x}, \boldsymbol{y}), \texttt{Assoc}(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{v})$.

EvCons *Cons* of s and t is $\texttt{(}s\ .\ t\texttt{)}$, so we have the following rule.

$\texttt{EvCons}(\texttt{(CONS } \boldsymbol{e1}\ \boldsymbol{e2}\texttt{)}, \boldsymbol{a}, \texttt{(}\boldsymbol{v1}\ .\ \boldsymbol{v2}\texttt{)})$:- $\texttt{Ev}(\boldsymbol{e1}, \boldsymbol{a}, \boldsymbol{v1}), \texttt{Ev}(\boldsymbol{e2}, \boldsymbol{a}, \boldsymbol{v2})$.

EvCar and EvCdr *Car* of (*s* . *t*) is *s* and *cdr* of (*s* . *t*) is *t*:

$\texttt{EvCar}((\mathsf{CAR}\ \boldsymbol{e}), \boldsymbol{a}, \boldsymbol{v1})$:- $\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, (\boldsymbol{v1} \ . \ \boldsymbol{v2}))$.

$\texttt{EvCdr}((\mathsf{CDR}\ \boldsymbol{e}), \boldsymbol{a}, \boldsymbol{v2})$:- $\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, (\boldsymbol{v1} \ . \ \boldsymbol{v2}))$.

EvAtom The Lisp primitive function *atom* decides whether a given `Sexp` is an `Atom` or not, and returns T (true) if it is an `Atom` and F[16] (false) if it is not an `Atom`. Since each of the two rule-schemes in the game Sexp corresponds to one of these cases, we have the following rules.

$\texttt{EvAtom}((\mathsf{ATOM}\ \boldsymbol{e}), \boldsymbol{a}, \mathsf{T})$:- $\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v}), \boldsymbol{v} : \texttt{Atom}$.
$\texttt{EvAtom}((\mathsf{ATOM}\ \boldsymbol{e}), \boldsymbol{a}, \mathsf{F})$:- $\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, (\boldsymbol{v1} \ . \ \boldsymbol{v2}))$.

EvEq The Lisp primitive function *eq* decides whether given two `Sexp`s are equal `Atom`s. We can use the games EqAtom and NeqAtom to implement the *eq* function.

$\texttt{EvEq}((\mathsf{EQ}\ \boldsymbol{e1}\ \boldsymbol{e2}), \boldsymbol{a}, \mathsf{T})$:- $\texttt{Ev}(\boldsymbol{e1}, \boldsymbol{a}, \boldsymbol{v1}), \texttt{Ev}(\boldsymbol{e2}, \boldsymbol{a}, \boldsymbol{v2}), \texttt{EqAtom}(\boldsymbol{v1}, \boldsymbol{v2})$.
$\texttt{EvEq}((\mathsf{EQ}\ \boldsymbol{e1}\ \boldsymbol{e2}), \boldsymbol{a}, \mathsf{F})$:- $\texttt{Ev}(\boldsymbol{e1}, \boldsymbol{a}, \boldsymbol{v1}), \texttt{Ev}(\boldsymbol{e2}, \boldsymbol{a}, \boldsymbol{v2}), \texttt{NeqAtom}(\boldsymbol{v1}, \boldsymbol{v2})$.

EvAppF, EvArgs, Apply and PairUp

$\texttt{EvAppF}((\boldsymbol{f} \ . \ \boldsymbol{E}), \boldsymbol{a}, \boldsymbol{v})$:- $\texttt{EvArgs}(\boldsymbol{E}, \boldsymbol{a}, \boldsymbol{V}), \texttt{Apply}(\boldsymbol{f}, \boldsymbol{V}, \boldsymbol{a}, \boldsymbol{v})$.

$\texttt{EvArgs}((), \boldsymbol{a}, ())$:- .
$\texttt{EvArgs}((\boldsymbol{e} \ . \ \boldsymbol{E}), \boldsymbol{a}, (\boldsymbol{v} \ . \ \boldsymbol{V}))$:- $\texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v}), \texttt{EvArgs}(\boldsymbol{E}, \boldsymbol{a}, \boldsymbol{V})$.

$\texttt{Apply}(\boldsymbol{x}, \boldsymbol{V}, \boldsymbol{a}, \boldsymbol{v})$:- $\boldsymbol{x} : \texttt{Var}, \texttt{Assoc}(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{f}), \boldsymbol{f} : \texttt{Fun}, \texttt{Apply}(\boldsymbol{f}, \boldsymbol{V}, \boldsymbol{a}, \boldsymbol{v})$.
$\texttt{Apply}((\mathsf{LAMBDA}\ \boldsymbol{X}\ \boldsymbol{e}), \boldsymbol{V}, \boldsymbol{a}, \boldsymbol{v})$:- $\texttt{PairUp}(\boldsymbol{X}, \boldsymbol{V}, \boldsymbol{a}, \boldsymbol{b}), \texttt{Ev}(\boldsymbol{e}, \boldsymbol{b}, \boldsymbol{v})$.
$\texttt{Apply}((\mathsf{LABEL}\ \boldsymbol{x}\ \boldsymbol{f}), \boldsymbol{V}, \boldsymbol{a}, \boldsymbol{v})$:- $\texttt{Apply}(\boldsymbol{f}, \boldsymbol{V}, ((\boldsymbol{x} \ . \ \boldsymbol{f}) \ . \ \boldsymbol{a}), \boldsymbol{v})$.

$\texttt{PairUp}((), (), \boldsymbol{a}, \boldsymbol{a})$:- .
$\texttt{PairUp}((\boldsymbol{x} \ . \ \boldsymbol{X}), (\boldsymbol{v} \ . \ \boldsymbol{V}), \boldsymbol{a}, ((\boldsymbol{x} \ . \ \boldsymbol{v}) \ . \ \boldsymbol{b}))$:- $\texttt{PairUp}(\boldsymbol{X}, \boldsymbol{V}, \boldsymbol{a}, \boldsymbol{b})$.

EvCondF and EvCond

$\texttt{EvCondF}((\mathsf{COND} \ . \ \boldsymbol{L}), \boldsymbol{a}, \boldsymbol{v})$:- $\texttt{EvCond}(\boldsymbol{L}, \boldsymbol{a}, \boldsymbol{v})$.

$\texttt{EvCond}((), \boldsymbol{a}, ())$.
$\texttt{EvCond}(((\boldsymbol{c}\ \boldsymbol{e}) \ . \ \boldsymbol{L}), \boldsymbol{a}, \boldsymbol{v})$:- $\texttt{Ev}(\boldsymbol{c}, \boldsymbol{a}, \mathsf{T}), \texttt{Ev}(\boldsymbol{e}, \boldsymbol{a}, \boldsymbol{v})$.
$\texttt{EvCond}(((\boldsymbol{c}\ \boldsymbol{e}) \ . \ \boldsymbol{L}), \boldsymbol{a}, \boldsymbol{v})$:- $\texttt{Ev}(\boldsymbol{c}, \boldsymbol{a}, \mathsf{F}), \texttt{EvCond}(\boldsymbol{L}, \boldsymbol{a}, \boldsymbol{v})$.

[16] Modern Lisp systems represent false by NIL.

4.5 Comparison with other approaches

The definition of the syntax and the operational semanitics of pure Lisp we just gave is basically along the line of specifying programming languages by structural operational semanitcs. For example, we introduced a language called Hyperlisp in [65, 66] and defined its semantics in a similar way as above. We also pointed out in [67] that the definitions given by such an approach can be regarded as PROLOG-like programs. Furthermore, we pointed out in [67] that Smullyan's elementary formal system [79] provides a similar framework for defining formal systems.

Elementary derivation games, by themselves, are as useful as other approaches such as PROLOG or elementary formal systems. The true advantage of our apporoach is that an elementary derivation game can be combined in a seamless way with other derviation games that embody logical and mathematical systems so that, in the combined game, we can formally reason about and even develop programs defined by the elementary derivation game. In other approaches, such a task can only be carried out by developing a meta-system which is quite different from the original systems.

5 Logical Aspects of NF

In this section, we show that we can naturally define various logical systems by means of derivation games. The naturalness of the definition owes very much to the facts that the discharging of an assumption is simply expressed by using a hypothetical judgment as a premise of rule-schemata and that the so-called eigenvariable condition on *local* variables in a derivation is automatically guaranteed by using an application schema in a premise of rule-schemata.

triv Before we proceed, consider the following trivial rule:

$$\frac{\boldsymbol{J}}{\boldsymbol{J}}\ \mathtt{triv}$$

It is clear that for any game G, the derivable judgments in G and $G \oplus \mathsf{triv}$ are the same. Therefore, we will assume that any game contains the `triv` rule-schema. It may seem that the rule is not only harmless but also useless. However, this rule is sometimes useful in practice, since, as we see below later we can use this rule to remind us of the judgment which a derivation variable or a general derivation proves.

5.1 Intuitionistic and Classical Propositional Logics

We first define intuitionistic and classical propositional logics as derivation games.

Prop We define formulas of propositional logic as follows. The judgment $p : \texttt{Prop}$ means that p is a formula. The derivation game Prop is elementary.

$$
\begin{array}{rll}
\bot : \texttt{Prop} & \texttt{:-}_{\bot\mathrm{F}} & . \\
\boldsymbol{p} \supset \boldsymbol{q} : \texttt{Prop} & \texttt{:-}_{\supset\mathrm{F}} & \boldsymbol{p} : \texttt{Prop}, \boldsymbol{q} : \texttt{Prop}. \\
\boldsymbol{p} \wedge \boldsymbol{q} : \texttt{Prop} & \texttt{:-}_{\wedge\mathrm{F}} & \boldsymbol{p} : \texttt{Prop}, \boldsymbol{q} : \texttt{Prop}. \\
\boldsymbol{p} \vee \boldsymbol{q} : \texttt{Prop} & \texttt{:-}_{\vee\mathrm{F}} & \boldsymbol{p} : \texttt{Prop}, \boldsymbol{q} : \texttt{Prop}.
\end{array}
$$

True We define provable formulas of propositional logic as follows. The game True is non-elementary because $\supset\texttt{I}$ rule-schema and $\vee\texttt{E}$ rule-schema below contain hypothetical judgment schemata in their premises.

$$
\begin{array}{rll}
\boldsymbol{p} \supset \boldsymbol{q} : \texttt{True} & \texttt{:-}_{\supset\mathrm{I}} & \boldsymbol{p} : \texttt{Prop},\ \boldsymbol{p} : \texttt{True} \Rightarrow \boldsymbol{q} : \texttt{True}. \\
\boldsymbol{q} : \texttt{True} & \texttt{:-}_{\supset\mathrm{E}} & \boldsymbol{p} \supset \boldsymbol{q} : \texttt{True},\ \boldsymbol{p} : \texttt{True}. \\
\boldsymbol{p} \wedge \boldsymbol{q} : \texttt{True} & \texttt{:-}_{\wedge\mathrm{I}} & \boldsymbol{p} : \texttt{True},\ \boldsymbol{q} : \texttt{True}. \\
\boldsymbol{p} : \texttt{True} & \texttt{:-}_{\wedge\mathrm{EL}} & \boldsymbol{p} \wedge \boldsymbol{q} : \texttt{True}. \\
\boldsymbol{q} : \texttt{True} & \texttt{:-}_{\vee\mathrm{ER}} & \boldsymbol{p} \wedge \boldsymbol{q} : \texttt{True}. \\
\boldsymbol{p} \vee \boldsymbol{q} : \texttt{True} & \texttt{:-}_{\vee\mathrm{IL}} & \boldsymbol{p} : \texttt{True},\ \boldsymbol{q} : \texttt{Prop}. \\
\boldsymbol{p} \vee \boldsymbol{q} : \texttt{True} & \texttt{:-}_{\vee\mathrm{IR}} & \boldsymbol{p} : \texttt{Prop},\ \boldsymbol{q} : \texttt{True}. \\
\boldsymbol{r} : \texttt{True} & \texttt{:-}_{\vee\mathrm{E}} & \boldsymbol{p} \vee \boldsymbol{q} : \texttt{True},\ \boldsymbol{p} : \texttt{True} \Rightarrow \boldsymbol{r} : \texttt{True},\ \boldsymbol{q} : \texttt{True} \Rightarrow \boldsymbol{r} : \texttt{True}. \\
\boldsymbol{r} : \texttt{True} & \texttt{:-}_{\bot\mathrm{E}} & \bot : \texttt{True},\ \boldsymbol{r} : \texttt{Prop}.
\end{array}
$$

We define the intuitionistic propositional logic NJ by putting

$$\mathsf{NJ} :\equiv \mathsf{Prop} \oplus \mathsf{True}.$$

Here is an example of a derivation in NJ.

$$
\dfrac{U \quad (X :: A : \texttt{True}) \left[\dfrac{\dfrac{U \quad V}{A \supset B : \texttt{Prop}}\,{\scriptstyle\supset\mathrm{F}} \quad (Y :: A \supset B : \texttt{True}) \left[\dfrac{X \quad Y}{B : \texttt{True}}\,{\scriptstyle\supset\mathrm{E}} \right]}{(A \supset B) \supset B : \texttt{True}}\,{\scriptstyle\supset\mathrm{I}} \right]}{A \supset (A \supset B) \supset B : \texttt{True}}\,{\scriptstyle\supset\mathrm{I}}
$$

This derivation is a derivation under the derivation context:

$$\texttt{<}A, U :: A : \texttt{Prop}, B, V :: B : \texttt{Prop}\texttt{>}.$$

Note that in ordinary natural deduction style formulation of the propositional logic, the above context is shifted to meta-level by declaring, at meta-level, that the variables A, B range over propositions. In derivation games such implicit assumptions are all made into explicit assumptions at object-level. So, if we allow ourselves to suppress such assumptions and suppress derivations of categorical judgments asserting A and $A \supset B$ are propositions, then[17] we get the following derivation.

$$\dfrac{(X :: A) \left[\dfrac{(Y :: A \supset B) \left[\dfrac{Y \quad X}{B} \supset\mathrm{E} \right]}{(A \supset B) \supset B} \supset\mathrm{I} \right]}{A \supset (A \supset B) \supset B} \supset\mathrm{I}$$

We stress that we may allow derivations like above as it is always possible to recover the suppressed information mechanically. A complete derivation of intuitionistic propositional logic, in general, must use rules in Prop as well as rules in True.

The above derivation is essentially the same as the following derivation[18] in the conventional natural deduction style. Note, however, that our derivation explicitly indicates scopes of the discharged assumptions and this enhances the readability of the derivation. Also, note that our derivation is an element in E while conventional derivation is an informal figure of the natural deduction system. Because of this difference, while we can easily define the substitution operation on derivations as we did in §2, the definition of substitution operation on conventional derivations, e.g., in [61], is rather complex.

$$\dfrac{\dfrac{\dfrac{A \supset B^Y \quad A^X}{B} \supset\mathrm{E}}{(A \supset B) \supset B} \supset\mathrm{I}, Y}{A \supset (A \supset B) \supset B} \supset\mathrm{I}, X$$

LEM (Law of the Excluded Middle) We can obtain the classical propositional logic NK by putting

$$\mathsf{NK} :\equiv \mathsf{NJ} \oplus \mathsf{LEM}$$

where the game LEM is defined by the following rule-schema, and in the schema we write $\neg A$ for $A \supset \bot$.

$$\boldsymbol{p} \vee \neg \boldsymbol{p} : \mathtt{True} \ \text{:-}_{\mathtt{LEM}} \ \boldsymbol{p} : \mathtt{Prop}.$$

5.2 Second Order Typed Lambda Calculus

The rule-schemata we have considered so far do not use universal judgments. We can express rules that handle quantifications by using universal judgments in premises of rule-schemata. Here, we give examples of such rule-schemata by defining Girard's second order typed lambda calculus F [25].

[17] We will also simply write C for $C : \mathtt{True}$.

[18] Gentzen [24] uses numbers as labels for assumptions but recently variables are also used as labels [89].

Type We define *types* of F by the following rules.

$$\frac{\boldsymbol{A} : \mathtt{Type} \quad \boldsymbol{B} : \mathtt{Type}}{\boldsymbol{A} \to \boldsymbol{B} : \mathtt{Type}} \to \mathtt{F} \qquad \frac{(x)\,[x : \mathtt{Type} \Rightarrow \boldsymbol{A}[\![x]\!] : \mathtt{Type}]}{\Pi\,(x : \mathtt{Type})\,[\boldsymbol{A}[\![x]\!]] : \mathtt{Type}} \,\Pi\mathtt{F}$$

Obj The game Obj defines *objects* of each type. Namely, we define the judgment $<a, A>$: `Obj`, which means that a is an object of type A. We will write $a \in A$ for $<a, A>$: `Obj`[19]

$$\frac{\boldsymbol{A} : \mathtt{Type} \quad (x)\,[x \in \boldsymbol{A} \Rightarrow \boldsymbol{b}[\![x]\!] \in \boldsymbol{B}]}{\lambda(x \in \boldsymbol{A})\,[\boldsymbol{b}[\![x]\!]] \in \boldsymbol{A} \to \boldsymbol{B}} \,\lambda\mathtt{I}(\boldsymbol{b}) \qquad \frac{\boldsymbol{b} \in \boldsymbol{A} \to \boldsymbol{B} \quad \boldsymbol{a} \in \boldsymbol{A}}{\mathsf{app}(\boldsymbol{b}, \boldsymbol{a}) \in \boldsymbol{B}} \,\lambda\mathtt{E}$$

$$\frac{(x)\,[x : \mathtt{Type} \Rightarrow \boldsymbol{b}[\![x]\!] \in \boldsymbol{B}[\![x]\!]]}{\Lambda(x : \mathtt{Type})\,[\boldsymbol{b}[\![x]\!]] \in \Pi\,(x : \mathtt{Type})\,[\boldsymbol{B}[\![x]\!]]} \,\Lambda\mathtt{I}(\boldsymbol{b}, \boldsymbol{B})$$

$$\frac{\boldsymbol{b} \in \Pi\,(x : \mathtt{Type})\,[\boldsymbol{B}[\![x]\!]] \quad \boldsymbol{A} : \mathtt{Type}}{\mathsf{App}(\boldsymbol{b}, \boldsymbol{A}) \in \boldsymbol{B}[\![\boldsymbol{A}]\!]} \,\Lambda\mathtt{E}(\boldsymbol{B})$$

We can now define F by putting:

$$\mathsf{F} :\equiv \mathsf{Type} \oplus \mathsf{Obj}.$$

We give a derivation of the typing judgment for the polymorphic identity function in F.

```
⊢ Λ(A:Type)[λ(x∈A)[x]] ∈ Π(A:Type)[A→A] in F since
Λ(A:Type)[λ(x∈A)[x]] ∈ Π(A:Type)[A→A]
by ΛI((A)[λ(x∈A)[x]],(A)[A→A]) {
    (A)[(X::A:Type)[
        λ(x∈A)[x] ∈ A→A by λI((x)[x]) {
            A:Type by triv(A:Type) {X};
            (x)[(Y::x∈A)[
                x∈A by triv(x∈A) {Y}
            ]]
        }
    ]]
}
```

In the above derivation, we have instantiated application schemata when we applied the rule-schemata λI and ΛI. Let us see in detail, how these rule-schemata are applied. In the case of the application of ΛI above, the rule-schema is applied

[19] Conventional notation for this judgment in type theory is $a : A$, but since ':' has different meaning in our theory, we are using a slightly non-standard notation.

under the derivation context <>, and the environment ρ for instantiating the rule-schema is such that $\rho(\boldsymbol{b}) \equiv \texttt{(}A\texttt{)[}\lambda\texttt{(}x \in A\texttt{)[}x\texttt{]]}$ and $\rho(\boldsymbol{B}) \equiv \texttt{(}A\texttt{)[}A{\rightarrow}A\texttt{]}$. So, we can compute the instance of the rule-schema by ρ as in the above derivation. The rule-schema $\lambda\texttt{I}$ is applied under the derivation context $A, X :: A : \texttt{Type}$ and the environment ρ for instantiating the rule-schema is such that $\rho(\boldsymbol{b}) \equiv \texttt{(}x\texttt{)[}x\texttt{]}$.

5.3 Untyped Lambda Calculus

As another example of a calculus that requires application schema in the defining rules, we define the untyped $\lambda\beta$-calculus $\mathsf{LambdaBeta}$ as follows.

Term The λ-terms are defined as follows.

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

An example of a λ-term is given by the following derivation:

```
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:Term by triv{X};
            y:Term by triv{Y}
        }
    ]]
}
```

Traditionally, λ-terms are defined as an inductively defined set of words over a fixed alphabet. So, for example, in [5, page 22] , the set of λ-terms Λ is defined as follows:

1. $x \in \Lambda$;
2. $M \in \Lambda \Rightarrow (\lambda x M) \in \Lambda$;
3. $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$;

where x in 1 or 2 is an arbitrary variable.

Fiore et al. [21] reformulates this definition in a *logical* way as follows. We let Γ be a context in the sense of Definition 1.

$$\frac{x \in \Gamma}{\Gamma \vdash x \in \Lambda} \qquad \frac{\Gamma \oplus \texttt{<}x\texttt{>} \vdash M \in \Lambda}{\Gamma \vdash (\lambda x M) \in \Lambda} \qquad \frac{\Gamma \vdash M \quad \Gamma \vdash N}{\Gamma \vdash (MN) \in \Lambda}$$

In this formulation, λ-terms are always viewed under a context which declares all the free variables in the term. Thus, the 1st rule is an assumption rule which says that x is a λ-term under the assumption that x is a λ-term. In our formulation, as we can see from the derivation above, this rule is a built-in rule common to all the derivation games.[20] The 2nd rule shows that the *assumption* x is discharged when we construct $(\lambda x M)$ out of M, and this suggests the *natural* reading of the 2nd clause of Barendregt's [5] definition of Λ. Namely, we can read it as the following judgment:

$$(x)\,[x \in \Lambda \Rightarrow M \in \Lambda] \Rightarrow (\lambda x M) \in \Lambda.$$

Our rule-schema $\lambda\mathsf{F}$ simply formalized this judgment as a rule-schema.

EqTerm We can define the β-equality relation on λ-terms by the game EqTerm which is defined by the following rule-schemata, where we write $M = N$ for $\texttt{<}M, N\texttt{>} : \texttt{EqTerm}$.

$$\frac{\lambda(x)\,[\boldsymbol{M}[\![x]\!]] : \texttt{Term} \quad \boldsymbol{N} : \texttt{Term}}{\mathsf{app}(\lambda(x)\,[\boldsymbol{M}[\![x]\!]], \boldsymbol{N}) = \boldsymbol{M}[\![\boldsymbol{N}]\!]}\ \beta(\boldsymbol{M}, \boldsymbol{N})$$

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

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

$$\frac{(x)\,[x : \texttt{Term} \Rightarrow \boldsymbol{M}[\![x]\!] = \boldsymbol{N}[\![x]\!]]}{\lambda(x)\,[\boldsymbol{M}[\![x]\!]] = \lambda(x)\,[\boldsymbol{N}[\![x]\!]]}\ \xi(\boldsymbol{M}, \boldsymbol{N})$$

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

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

The game $\mathsf{LambdaBeta}$ is a completely *formal* definition of the $\lambda\beta$-calculus, while, in [5, pp. 22–23], the $\lambda\beta$-calculus is *informally* defined as the equational theory $\boldsymbol{\lambda}$. For example, in [5, page 23], the rule-schema $\texttt{appL}$ is informally given as:

$$M = N \Rightarrow MZ = NZ.$$

and there the sign '$\Rightarrow$' means informal implication and also the condition that Z is a λ-term is not explicitly mentioned but it is implicitly declared as a variable ranging over λ-terms.[21] In $\boldsymbol{\lambda}$, the β-conversion rule is given as:

$$(\lambda x.\ M)N = M[x := N].$$

Here again the conditions that $\lambda x.\ M$ and N are λ-term are assumed only implicitly.

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

[20] See Definition 9 in §3.

[21] In [5, page 22], it is stated that '$M, N, L, \ldots$ denote arbitrary λ-terms' and, by common sense, we understand that the letter Z is embedded in '$\ldots$'

```
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,x):Term by appF {
                x:Term by triv {X};
                x:Term by triv {X}
            }
        ]]
    };
    y:Term by triv {Y};
}
```

In the above derivation, the β-rule-schema is instantiated by the environment ρ such that $\rho(\boldsymbol{M}) \equiv (x)\,[\mathsf{app}(x,x)]$ and $\rho(\boldsymbol{N}) \equiv y$. Hence the schema $\boldsymbol{M}\,[\![\boldsymbol{N}]\!]$ is instantiated as follows:

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

6 Related Works

Our work presented here is strongly influenced by Martin-Löf's philosophical view of mathematics expressed in e.g. [46, 49], and by the design principle of Edinburgh LF [31]. In this section, we will compare our work to these works in detail. We will also compare our work with other related works including the generalized logic calculus GLC of Takeuti [82] which inspired our theory of expressions and schemata.

6.1 Edinburgh LF

We begin with the comparison of NF with Edinburgh LF utilizing a concrete example. In NF, we can define the implicational fragment of intuitionistic propositional logic, which we tentatively call IFIPL, by the following derivation game $\mathsf{IPL}_{\supset}$:

$$
\begin{aligned}
\boldsymbol{p}\supset\boldsymbol{q} : \mathtt{Prop} \ &\text{:-}_{\supset\mathtt{F}}\ \boldsymbol{p} : \mathtt{Prop}, \boldsymbol{q} : \mathtt{Prop}. \\
\boldsymbol{p}\supset\boldsymbol{q} : \mathtt{True} \ &\text{:-}_{\supset\mathtt{I}}\ \boldsymbol{p} : \mathtt{Prop},\ \boldsymbol{p} : \mathtt{True} \Rightarrow \boldsymbol{q} : \mathtt{True}. \\
\boldsymbol{q} : \mathtt{True} \ &\text{:-}_{\supset\mathtt{E}}\ \boldsymbol{p}\supset\boldsymbol{q} : \mathtt{True},\ \boldsymbol{p} : \mathtt{True}.
\end{aligned}
$$

Edinburgh LF employs *judgments-as-types* principle and represents judgements as LF types. LF has three levels of terms: *objects*, *families* and *kinds*, and it has a distinguished kind Type which is used to classify types. In LF, a logical system is defined by a *signature* which is a finite set of declarations of constants specific to the logical system to be defined. So, IFIPL can be defined by the following signature $\Sigma_{\supset}$ consisting of the following declarations:

$$\begin{aligned} o &: \mathtt{Type}, \\ \supset &: o \to o \to o, \\ \mathtt{True} &: o \to \mathtt{Type}, \\ \supset\mathtt{I} &: \Pi p : o. \Pi q : o. (\mathtt{True}(p) \to \mathtt{True}(q)) \to \mathtt{True}(\supset (p)(q)), \\ \supset\mathtt{E} &: \Pi p : o. \Pi q : o. \mathtt{True}(\supset (p)(q)) \to \mathtt{True}(p) \to \mathtt{True}(q). \end{aligned}$$

First, the constant o is declared as a type, and it is intended to represent the type of propositions. Then, $\supset$ is introduced as a function that constructs a proposition $\supset (p)(q)$ from propositions p and q. For a proposition p, the judgment 'p is true' is represented by the type $\mathtt{True}(p)$ following the judgments-as-types principle, and proofs (derivations) of the judgment is represented as objects having type $\mathtt{True}(p)$. Finally, derivation rules are declared as functions mapping propositions and proofs to proofs. We compare NF and LF using this example.

(1) NF defines the logic as a derivation game $\mathsf{IPL}_{\supset}$ which is a concrete element in S consisting of 3 rule-schemata. LF defines the logic by the signature $\Sigma_{\supset}$ which declares 5 constants. $\Sigma_{\supset}$ has 3 declarations that correspond to the 3 rule-schemata in $\mathsf{IPL}_{\supset}$, and 2 additional declarations. Since LF is a typed calculus, it is necessary to declare the types of o and True, but such explicit declarations are not necessary in NF. The constants o and True corresponds to two concepts Prop and True defined in $\mathsf{IPL}_{\supset}$, and they are *implicitly* declared by the fact that they occur in the concept part of judgment-schemata.

(2) If we reflect on how IFIPL is defined informally[22], we see that it is defined by defining two concepts, namely, the concept *proposition* and the concept *truth*. In fact, if we define IFIPL informally, we would first define *formulae* inductively, and then define *provable formulae* again inductively. Such inductive definitions are given in terms of rules. The following table shows how these concepts are formalized in NF and LF.

Informal judgment	NF	LF
p is a formula	$p : \mathtt{Prop}$	$p : o$
p is provable	$p : \mathtt{True}$	$\mathtt{True}(p)$

In LF only the judgment 'p is provable' is defined as LF type $\mathtt{True}(p)$. We will call such an LF type *judgmental-type*. The following quotation from [31, §4] explains why only one of the two concepts is formalized as an LF judgmental-type.

[22] IFIPL is a formal system, but we usually define it informally using informal meta-language like English or Japanese.

> The judgments-as-types principle is merely the formal expression of the fact that the assertion of a judgment in a *formal system* is precisely the claim that it has a *formal proof* in that system.

From the above quotation we see that LF's approach is to look at a *formal system* (here, it is IFIPL) and identify the assertions of judgments in the formal system (here, they are $\vdash p$ where p are IFIPL-formulas).

In contrast to LF's approach, NF's approach is based on the following observation about informal mathematics we conduct day by day.

> (*To assert is to prove.*) The assertion of a judgment in mathematics is precisely the claim that it has a proof.[23]

For example, imagine that you are a teacher of logic and teaching IFIPL to your students, and just defined inductively what is a *formula*. Then you can assert that:

> Thm A. *If p is a formula, then $p \supset p$ is a formula.*

By making the assertion, you are claiming that it has a proof[24]; and you would give a proof of it by saying: "Assume that p is a formula. Then by the implication formation rule in the definition of formulae, we see that $p \supset p$ is a formula." You will then go on to define what it means that a formula is *provable*, and would make a second assertion:

> Thm B. *If p is a formula, then $p \supset p$ is provable.*

Again, you can justify the claim by giving a poof of it.

The derivation game $\mathsf{IPL}_\supset$ in NF faithfully reflects such informal practice by translating informal judgments 'p is a formula' ('p is provable', resp.) to $p : \mathtt{Prop}$ ($p : \mathtt{True}$, resp.) and giving appropriate rule-schemata to define them. The above two metatheorems become the following closed judgments in NF:

$$(p)\,\mathtt{[}p : \mathtt{Prop} \Rightarrow p \supset p : \mathtt{Prop}\mathtt{]},$$
$$(p)\,\mathtt{[}p : \mathtt{Prop} \Rightarrow p \supset p : \mathtt{True}\mathtt{]}.$$

Moreover, we can construct closed $\mathsf{IPL}_\supset$-derivations D_{ThmA} and D_{ThmB} such that:

$$\vdash (p)\,\mathtt{[}p : \mathtt{Prop} \Rightarrow p \supset p : \mathtt{Prop}\mathtt{]}\ \mathtt{in}\ \mathsf{IPL}_\supset\ \mathtt{since}\ D_{\mathrm{ThmA}},$$
$$\vdash (p)\,\mathtt{[}p : \mathtt{Prop} \Rightarrow p \supset p : \mathtt{True}\mathtt{]}\ \mathtt{in}\ \mathsf{IPL}_\supset\ \mathtt{since}\ D_{\mathrm{ThmB}}.$$

[23] The *to-assert-is-to-prove* principle is usually regarded as an intuitionistic principle (see e.g. [46], [6, page 403]). This is probably because the abstract notion of proof has been well developed for constructive mathematics as theories of constructions by Kreisel [43, 44], Scott [74], Martin-Löf [46] and others based on Brouwer-Heyting-Kolmogorov interpretation of logical constants, and less well developed for classical mathematics. However, the to-assert-is-to-prove principle is effective in practice for all modern mathematics, since only assertions with valid proofs are accepted as theorems (see also, Troelstra and van Dalen [88] and Sato [70]).

[24] It is an informal proof that proves a metatheorem (Thm A).

In contrast, in LF with signature $\Sigma_{\supset}$, only Thm B can be represented by the closed LF judgment:

$$\Pi p : o.\, \mathtt{True}(\supset (p)(p)),$$

and can be justified by constructing a closed LF term M_{ThmB} such that:

$$\vdash_{\Sigma_{\supset}} M_{\mathrm{ThmB}} : \Pi p : o.\, \mathtt{True}(\supset (p)(p)).$$

As for ThmA, a naïve translation of it would be:

$$\Pi p : o.\, \supset (p)(p) : o.$$

Now, in LF we can derive:

$$p : o \vdash_{\Sigma_{\supset}} \supset (p)(p) : o.$$

But, from above, we can only conclude that:

$$\vdash_{\Sigma_{\supset}} \lambda p : o. \supset (p)(p) : o \to o$$

and there are no LF typing rules to conclude

$$\vdash_{\Sigma_{\supset}} M_{\mathrm{ThmA}} : \Pi p : o.\, \supset (p)(p) : o$$

for some closed LF term M_{ThmA}. In fact, $\Pi p : o.\, \supset (p)(p) : o$ is not even a term acceptable by the LF grammar. In summary, we have the following table.

Informal Thm	NF	LF
Thm A	$(p)\,[p : \mathtt{Prop} \Rightarrow p \supset p : \mathtt{Prop}]$	—
Thm B	$(p)\,[p : \mathtt{Prop} \Rightarrow p \supset p : \mathtt{True}]$	$\Pi p : o.\, \mathtt{True}(\supset (p)(p))$

The above argument relies on a particular choice a signature for defining IFIPL in LF and hence does not exclude the possibility of finding another signature Σ' which can formalize and prove both ThmA and ThmB. But, such a signature would have to be more complex than $\Sigma_{\supset}$ and would therefore be much more complex than the derivation game $\mathsf{IPL}_{\supset}$ of NF.

(3) Let us now compare the $\supset\mathtt{I}$ rule in NF and LF. In NF, it is a rule-schema having two premises, $\boldsymbol{p} : \mathtt{Prop}$ and $\boldsymbol{p} : \mathtt{True} \Rightarrow \boldsymbol{q} : \mathtt{True}$, and the conclusion $\boldsymbol{p} \supset \boldsymbol{q} : \mathtt{True}$. So, this rule-schema faithfully translates the informal intuitionistic definition of the logical constant $\supset$ which says that for any expressions p and q, if we can show that p is a proposition and that q is true whenever p is true, then we may conclude that $p \supset q$ is true. In LF, $\supset\mathtt{I}$ is a constant function which accepts three arguments, and returns a proof of $\mathtt{True}(\supset (p)(q))$. Of the three arguments, the fist two are p and q each having the type o of propositions and the third argument guarantees that we can have a proof of $\mathtt{True}(q)$ whenever we

have a proof of $\texttt{True}(p)$. Thus, we see that the implication introduction rule of IFIPL is represented fatithfully both in NF and LF.

There is, however, a minor diffrence in the two representations. Namely, the $\supset$`I` rule-schema in NF does not have a premise that correspond to the second argument of the function $\supset$`I` in LF which restricts q to be a proposition. In NF, it is possible to add a premise $\boldsymbol{q} : \texttt{Prop}$, but we did not do that since this will not change the set of derivable judgments[25]. In LF the second argument cannot be removed since the variable q must be declared toghether with its type, so that the rule becomes a closed LF term.

(4) A more essential difference between the $\supset$`I` rule-schema in NF and the function $\supset$`I` in LF is the following. The NF rule has a hypothetical judgment

$$\boldsymbol{p} : \texttt{True} \Rightarrow \boldsymbol{q} : \texttt{True}$$

in its premise while LF encodes the hypothetical judgment as the type

$$\texttt{True}(p) \to \texttt{True}(q).$$

In NF, the operator $\Rightarrow$ constructs a hypothetical judgment, and its meaning is completely determined by the 3rd clause of the Definition 9 and other clauses of G-derivation, where these clauses are defined by purely combinatorial conditions. Therefore, due to the finitistic character of the definition, we can be sure that the meaning of the hypothetical judgment is equally well understood among the readers who might be an intuitionist or not. On the other hand, in LF, $\to$ is a function type constructor in the LF type theory, and the hypothetical judgment in Edinburgh LF is, so to speak, *implemented* by the function type by applying LF's judgments-as-types principle. Now, as is well known, we can assign meanings to such function types in various ways, and this makes the meaning of the original hypothetical judgment unclear. Thus, although, technically speaking, the same (that is the same after the obvious translation) judgments (in IFIPL) are derivable both in NF and LF, LF does not provide an answer to the foundational question: "What is a hypothetical judgment?" We will touch upon this point again in §6.2.

(5) There is yet another difference between NF and LF concerning the extendability and modifiability of the set of rules. In intuitionistic logic, it is possible to give a more liberal formation rule for implication than the $\supset$`F` rule-schema we gave above. Martin-Löf gives the following implication formation rule in [49]:

$$\frac{A \text{ prop} \quad \begin{array}{c}(A \text{ true})\\ B \text{ prop}\end{array}}{A \supset B \text{ prop}}$$

In NF, we can easily represent this rule by the $\supset$`F+` rule-schema as follows:

[25] This is because the rule-schemata is designed to guarantee that any true expression is always a proposition. So, in NF, q can be any expression while in LF, it must be a proposition.

$$\boldsymbol{p} \supset \boldsymbol{q} : \texttt{Prop} \;\texttt{:-}_{\supset\texttt{F+}}\; \boldsymbol{p} : \texttt{Prop}, \boldsymbol{p} : \texttt{True} \Rightarrow \boldsymbol{q} : \texttt{Prop}.$$

It is however not clear how we should modify the LF declaration $\supset : o \to o \to o$, since it is now possible that $p \supset q$ becomes a proposition even if q is not a proposition. In fact, in NF, it is possible to extend $\mathsf{IPL}_\supset$ to a theory of rational numbers that has categorical judgments $x : \texttt{Q}$ meaning that x is a rational number as well as other categorical judgments that represent standard informal relations and operations on rational numbers. Then in such a theory, which is a game in NF, we can show the following judgment:

$$(x : \texttt{Q})\,[x \neq 0 \supset x \cdot x^{-1} = 1 : \texttt{True}].$$

By specializing x to 0, we get a true proposition $0 \neq 0 \supset 0 \cdot 0^{-1} = 1$ whose conclusion is not a proposition.

We have thus compared NF with Edinburgh LF using a concrete example. There is one more point we wish compare concerning the 'purity' of the definitional equality relations of the frameworks. We will make a comparison on this point in the next subsection.

6.2 Martin-Löf's theory of expressions and his philosophy

We first compare our theory of expressions and schemata with Martin-Löf's theory of expressions with the notion of arity and then describe the influence of Martin-Löf's philosophy on our work.

Theory of expressions Just like the design of Edinburgh LF was influenced by Martin-Löf's theory of expressions, our framework is also inspired by his theory. According to [55], Martin-Löf's theory of expressions is intended to provide a syntactic framework which can be used to represent various syntactic entities of intuitionistic type theory in a uniform and abstract manner.

In Martin-Löf's theory of expressions, expressions are constructed from given stocks of constants and variables by means of the four operations of (i) forming a *combination* of n expressions out of an ordered list of n expressions, (ii) *selecting* the i-th component of a combination, (iii) forming an *abstract* out of an expression by abstacting away a specified variable, and (iv) *applying* an abstract expression to an expression. If we allow all the expressions freely constructed by these operations, then the definitional equality would become undecidable just as the β-equality on untyped lambda terms is undecidable. So, Martin-Löf restricts such free constructions by assigning an *arity*, which is a notion similar to type[26], to each expression. Thanks to this restriction, the definitional equality on expressions becomes a decidable relation on expressions, and Martin-Löf's

[26] As pointed out in [4, 55], Martin-Löf's theory of expressions is essentially a simply typed lambda calculus.

theory of expressions provides a framework for developing formal systems that require the notions of abstraction and substitution.

Since the notion of arity is weaker than the notion of dependent type, Martin-Löf's theory of expressions serves as a foundational framework for defining his intuitionistic theory of types on top of it. However, the theory still looked too strong to us because in order to show the decidablity of definitional equality one must verify a number of combinatorial properties of expressions such as the Church-Rosser property and the normalization theorem. Another reason why we felt the theory too strong was that although definitional equality is decidable, deciding the definitional equality of two given expressions is sometimes not entirely evident, and in such a case we would like to have the *evidence* for the equality, and such an evidence should be provided by a derivation like the one we developed in this paper[27]. In this way, we thought that there should be more basic theory of expressions whose defintioal equality is syntactic identity[28]. Our theory of expressions was designed with this motivation.

We have already developed a theory of symbolic expressions in [66, 67]. There, the data structure of symbolic expressions is defined only by means of the *constructors* that construct pairs out of two expressions. The selectors which select left and right component of the pair are introduced only *after* the data structure has been defined. By the same approach, in §2, we defined the data structure E of expressions by means of the two constructors of constructing a pair and constructing an abstraction.

Once the set E of expressions is defined only by means of constructors, we can introduce operators to *destruct* and analyze the data structure. We perform such operations by looking at expressions (which are elements of E) from outside, and therefore we need *meta*-expressions if we wish to have linguistic forms to express such operations. So, in §2, we introduced *schemata* as such meta-expressions, so that we could express the operation of applying an abstract to an expression by a schema. We also saw, without incurring the problems of checking the Church-Rosser property etc., that any such application can be evaluated to a unique expression relative to an environment.

To define a data structure only by means of constructors is a common practice in computer science and we simply followed this practice. For example, as we saw in §4, McCarthy [51] defined symbolic expressions only by means of the pairing operation, and then he introduced meta-expressions, such as *car*[(A . B)], which can be used to analyze symbolic expressions.

Now, concerning the strength of the frameworks, de Bruijn [14] pointed out that:

[27] As a matter of fact, students and, for that matter, even experts make mistakes in converting an expression to another expression. We can pinpoint such mistakes only if the judgment that two expressions are definitionally equal is asserted together with its derivation.

[28] Then, we can check the equality of two linguistic expressions by comparing each symbol in the expressions one by one. The definitional equality of our expressions is syntactic identity modulo α-equivalence, but it is possible to modify the definition of expressions so that the definitional equality will become syntactic identity.

> Over the years there has been the tendency to strengthen the frameworks by rules that enrich the notion of definitional equality, thus causing impurities in the backbones of those frameworks: the typed lambda calculi.

Then, he made a plea for seeking weaker frameworks by remarking that:

> For several reasons (theoretical, practical, educational and sociological) it may be recommended to keep the frames as simple as possible and as weak as possible.

We cannot agree more with what de Bruijn says in the above two qutoations. However, we do not follow what he then says:

> We restrict the discussions to frames that have typed lambda calculi as their backbone, an idea that is gaining field today, along with the rising popularity of the principle to treat proofs and similar constructions in exactly the same way as the more usual mathematical objects.

Martin-Löf's theory of expressions with arity is 'impure' as a framework in the sense that expressions are defined not only by constructors but also by destructors which are unnecessary as we have shown, and other typed lambda calculi (including calculi of AUTOMATH and Edinburgh LF) have the same impurity; and this impurity is the source of the difficulty in verifying the decidability of definitional equality of these calculi. We believe that we have succeeded in designing a framework which is simpler than the frameworks based on typed lambda calculi, and yet can treat proofs and similar constructions in the same way as the more usual mathematical objects.

Martin-Löf's philosophy We now turn to the influence of Martin-Löf's philosophy of mathematics on our work.[29] From his articles, e.g., [46, 47, 49], we find that Martin-Löf always wanted to provide a formal framework suitable for developing full-fledged intuitionistic mathematics. Such a framework must be foundationally well-founded and he developed his theory of expressions equipped with decidable definitional equality. He then uses these expressions as a medium to present his intuitionistic theory of types. In developing intuitionistic logic and mathematics, Martin-Löf is very careful (for a natural reason we explain later) about the priority among various notions to be introduced. For example, in [49, page 19], he says:

> Now, the question, What is a judgment? is no small question, because the notion of judgment is just about the first of all the notions of logic, the one that has to be explained before all the others, before even the notions of proposition and truth, for instance.

[29] What follows is our interpretation of Martin-Löf's philosophy and can be different from what he thinks. See also [6] for Beeson's account of Martin-Löf's philosophy.

We completely agree with this remark about the notion of judgment,[30] and we placed the notion of judgment as the most basic notion of our theory.

After explaining judgments in [49], Martin-Löf characterizes an *evident* judgment as a judgment for which a *proof* has been constructed. According to him, a proof is an act of grasping (or knowing) the evidence of the judgment. Therefore, since we conduct the act of proof by applying proof-rules, each proof-rule must be sound in the sense that it sends evident judgments to an evident judgment. To guarantee the soundness of the proof-rule, prior to introduce the rule, one must know the meanings of the categorical judgments involved in the rule. Then, to know the meanings of the categorical judgments properly, the *concepts* involved in the categorical judgments must be introduced in a proper order. Thus, in Martin-Löf's intuitionistic theory of types, *semantical* explanation of each concepts involved is essential in establishing the soundness of the proof-rules of the theory, although the theory itself is a formal theory that manipulates syntactic entities.

In our theory, we introduced the notion of derivation which corresponds to the notion of proof in Martin-Löf's theory. Just like Martin-Löf's proofs, our derivations are attached to judgments and never attached to propositions or formulae. Martin-Löf stressed the character of a proof as an act, but we think that he also accepts the possibility of recording such an act in a linguistic form. So, we think that our deriviations which are elements of E contain proofs in Martin-Löf's sense.

In this paper, we have introduced the notions of judgments and derivations syntactically without referring to the meanings of categorical judgments. This was made possible by introducing the notion of *derivation game* which we have arrived at by observing the activitiy of mathematicians including ourselves. The result of the observation was that not only an intuitionist mathematician but also a classical mathematician has the right to *assert* a judgment, say the judgment that Fermat's Last Theorem is true, only if he has *proven* it, namely, has carried out the act of proof and has a derivation as the record, or the *trace*[31] in computer science terminology, of his act of making the judgment evident.[32] Thus, the evidence, that is, the derivation, is initially private to the mathematician who proved the judgment, but the derivation can later be used to make the judgment evident to other mathematicians. As we have mentioned briefly in §1, the notion of evidence is not an absolute notion but a notion that depends on the philosophical (epistemological and ontological) standpoint of each mathematician. Thus the law of the excluded middle is evident to classical mathematicians but not so to intuitionist mathematicians. So, evidence is a semantical notion, and we therefore developed our theory without using the notion of evidence, but using the notion of proof-rule (which we call rule-schema) which is a syntactic

[30] Thus, for us, the *judgments-as-types* principle is an unnatural principle since it is an attempt to explain judgments in terms of types.

[31] Goto [30] observed that the trace of a computation can be regarded as a derivation.

[32] This is the to-assert-is-to-prove principle upon which our theory of judgments and derivations is based.

notion. This is not to say that semantics is unimportant. On the contrary, precisely because we consider semantics to be more important than syntax, we have tried to develop a general theory of judgments and derivations by referring only to syntactic objects, namely, expressions and schemata. In fact, all the notions and theorems we introduced in §§2–3 are explained in a finitistic manner[33] (as advocated by Hilbert in his program) and therefore can be equally well understood both by intuitionist and classical mathematicians. In this way, NF has been designed so as to provide a common framework in which we can formally develop both intuitionistic and classical mathematics.

Another important characterisitic of Martin-Löf's theory of types is its *open-endedness*. (See, e.g., [46, page 3], [49, page 17], [16, 90].) The open-endedness of the theory allows us to introduce new types without affecting the evidence of the judgments already established. This reflects well the common practice of mathematics where mathmaticians extend mathematics by introducing new concepts. NF is also designed to be open-ended, and in NF we can introduce a new concept by adding new rule-schemata that mention the concept to the derivation game we are interested in. For example, consider the mathematical induction rule `ind` we showed in §1 again. In traditional presentation of formal logical calculi, the class of *formulae* must be specifed before introducing inference rules. Thus if the calculus is first-order, then the *formula* $P[x]$ must be a first-order formula where the possible terms that may occur in $P[x]$ are also determined by the language of the calculus. So, if one wishes to extend the calculus to, say, a second-order calculus, then one must re-define the language and re-state the `ind`-rule, in spite of the fact that the principle of mathematical induction is valid for any *proposition* on a natural number. By the same reason, recalling the remark in §6.1(5), the IND-rule of Edinburgh LF we cited in §1 is also not open-ended. In contrast, the $\mathtt{ind}(\boldsymbol{P})$ rule of NF is open-ended just as in Martin-Löf's type theory.

6.3 Takeuti's GLC

In 1954, Takeuti [82] introduced the generalized logic calculus (GLC) which is a higher type extension of Gentzen's classical sequent calculus LK. GLC has the notion of *finite type* which is defined by the following grammar:[34]

$$\tau ::= \iota \mid [\tau_1, \ldots, \tau_n].$$

[33] There is no consensus as to the exact extent of the finitistically acceptable method. However, it is generally accepted that finitisitic arithmetic contains PRA (primitive recursive arithmetic) (see, e.g., [91, page 303], Takeuti [83, pp. 81 – 85], Tait [81], Simpson [75], Feferman [20, page 196] and Okada [56, §3.2]). Our method is finitistic in the sense that we can carry out all the arguments in §§2–3 in PRA. A natural consequence of this is the primitive-recursive decidability of whether a given expression is a G-derivation or not, and this answers Kresel's dictum in an affirmative way. Thus, we can recognize a proof when we see it.

[34] Takeuti uses the symbol 0 instead of ι, but here we use ι to relate Takeuti's finite type to Church's simple type. Takeuti's $[\tau_1, \ldots, \tau_n]$ roughly corresponds to Church's $\tau_1 \to \cdots \to \tau_n \to o$.

In formulating the basic syntax of GLC, Takeuti introduced the notion of *variety* which he defined simultaneously with the notion of *formula*. Varieties and formulae are therefore related with each other, and among the defining clauses of these notions he has the following clauses.[35]

- If A is a formula and $\varphi_1, \ldots, \varphi_n$ are distinct variables each of type τ_i, then $\{\varphi_1, \ldots, \varphi_n\}A$ is a variety of type $[\tau_1, \ldots, \tau_n]$.
- If φ is a variable of type $[\tau_1, \ldots, \tau_n]$ and $V_1, \ldots, V_n$ are varieties each of type τ_i, then $\varphi[V_1, \ldots, V_n]$ is a formula.

We can clearly see that these clauses are structurally the same as the *abstract* and *application* clauses in Definition 3. Here, we note an important difference between Takeuti's definition of formula and Church's definition of terms of type o in [15]. Namely, while Takeuti does not admit, for example, $\{x, y\}(x = y)[2, 3]$ as a valid formula, Church admits $(\lambda x, y. =xy)(2)(3)$ as a valid formula, i.e., a valid term of type o. Takeuti, however, admits $\{x, y\}(x = y)[2, 3]$ and similar expressions as meta-expressions which can be calculated at meta-level, rather than at object-level which is the case for Church's simple theory of types. Calculations of such meta-expressions are carried out by substitution in a similar fashion as our computation of application schemata relative to environments. It is to be noted that Takeuti uses the term '*abstract*' instead of 'variety' in his later publications [83, 84].

We may thus say that although Takeuti did not develop a general theory of expressions, he introduced essentially the same mechanism of abstraction and instantiation which we developed in §2.

6.4 Other related works

Just as Martin-Löf did in [46, page 3], Katuzi Ono [57] questioned the way the formalistic notions of formula and theorem are presented in standard logic texts, and he developed a unique notion-object theory whose basic concepts are *notion*, *object* and *categorical judgment*, and where a categorical judgment sP (which corresponds to our $s : P$) means that the notion P applies to the object s. The design of NF is considerably influenced by this book[36] of Ono, especially by his attitude to treat the notion of *notion* as more basic than other basic notions such as *set* or *type* as well as by his strong will to *naturally* formalize mathematics as a whole.[37]

Abstract theory of proofs has been developed in order to metamathematically analyze the intuitionistic notion of proof given by Brouwer-Heyting (see for example [34]). There are a lot of studies on this and related research area. Theory of constructions was initially studied by Kreisel [43], Goodman [27], Scott

[35] For the sake of simplicity we use a simplified version of the defintion given in [84]. Also, following Gentzen, Takeuti distinguishes free and bound variables syntactically by providing two disjoint sets of variables, but we do not do so.

[36] The book is written in Japanese.

[37] Axiomatized set theory ZF, for instance, is not an answer because of its unnaturalness to treat every mathematical entity as a set.

[74] and others (e.g., Aczel [3], chapter XVII of Beeson [6], Kobayashi [40], Sato [69]), and seems to be eventually merged with type theory. Theoretical foundations for de Bruijn's AUTOMATH system [13] and many other proof-checking systems, e.g., [9, 10, 58, 60] so far developed are mostly given by various forms of typed lambda calculi. Kleene's recursive realizability [41] initiated the research in realizability interpretaion and found applications in automatic extraction of programs from proofs of the specifications (see, e.g., Troelstra [87], Hayashi [33], Tatsuta [86]). Gödel's theory T [26] may also be regarded as an abstract theory of proofs, and it was also applied to program extraction by Sato [64] and Goto [28, 29]. Type theory can also be used to develop automatic program extraction systems and its theoretical foundation is given by the Curry-Howard isomorphism (see, e.g., [55]). It is hoped that NF will provide a common framework to implement and compare these different approaches.

In a section on proof theory ([44, pp. 149 – 179]), Kreisel pointed out the possible ambiguities of representation of the derivability relation ('P is a proof of A in the formal system F') within F (via encoding of syntax within F) and asked for a workable framework for presenting formal systems that will admit canonical representation of the derivability relation. Usefulness of pairing structure for providing such frameworks was acknowledged by Feferman [18], Sato [67] and Voda [92, 93]. NF is also intended to provide such a workable framework.

7 Conclusion

We have presented a constructive (finitistic) theory of judgments and derivations, and introduced the Natural Framework (NF) which can be used to implement various forms of computational and logical systems uniformly and naturally as derivation games.

The design goal of NF was to provide a canonical framework for *formally* presenting metamathematical objects such as proofs and theorems, and thereby enable us to treat such objects in the same way as we treat ordinary mathematical objects like numbers and topological spaces. This is also the goal set by de Bruijn[38] and by Kreisel[39]. Our plan was that a framework that attains the goal would be obtained by first observing informal mathematics carefully and then translate it into a framework as *naturally* as possible. The approach we took was very similar to the approach taken by Gentzen [24] who carefully analyzed informal proofs and formulated his natural deduction systems NJ and NK.

In our case, we made three key observations. The first observation was the *to-assert-is-to-prove* principle we described in §6.1. The second observation, which Martin-Löf [49] also made already, was that what we prove in informal mathematics is not a proposition but a *judgment*. The third observation we made about informal mathematics was that the notion of *free variable* which we find in formal mathematics does not exist in informal mathematics. This is because, in informal mathematics, whenever we wish to use a variable, we declare the

[38] See §6.2.
[39] See §6.4.

variable before the usage. Thus, we say 'Let V be a vector space over the field of real numbers' or the like, and then use the declared variables within the scopes of the declarations. This means that any mathematical expression is always considered under a *context*.[40] In formal mathematics, it seems that it is only recently that this practice in working mathematics has started to be respected, notably in type theory and by Fiore et al. [21] in the untyped lambda calculus.

The notions of judgment and derivation have been introduced in this paper in a finitistic way so that these notions are equally well understandable by both intuitionist and classical mathematicians. We could achieve this finitistic character of these fundamental notions by defining these notions without referring to the *meanings* of categorical judgments. By doing so, we could also define the *derivability*, relative to a given game G, of *hypothetical judgments* and *universal judgments* in a finitistic way. It should be noted that our finitistic definition of the *meanings*[41] of these two forms of judgments we gave in Definition 9 is in accordance with the explanations Martin-Löf gave for these forms of judgments in [49]. In [49, page 41], Martin-Löf explains a proof of a hypothetical judgment as follows.

> Now, quite generally, a proof of a hypothetical judgment, or logical consequence, is nothing but a hypothetical proof of the thesis, or consequent, from the hypotheses, or antecedents. The notion of hypothetical proof, in turn, which is a primitive notion, is explained by saying that it is a proof which, when supplemented by proofs of the hypotheses[42], or antecedents, becomes a proof of the thesis, or consequent.

Now, in NF, a *hypothetical proof* of, say, $H \Rightarrow J$ under Γ is nothing but a derivation of J under $\Gamma \oplus \langle X :: H \rangle$, and by Theorem 3.4, we see that when substituted by a derivation of H (under Γ) it becomes a proof of J (under Γ). Then, in page 43, he explains a proof of a universal judgments as a *free variable proof* as follows.

> And what is a free variable proof? It is a proof which remains a proof when you substitute anything you want for its free variables, that is, any expressions you want, of the same arities as those variables.

Here, again, Definition 9 and Theorem 3.5 are in complete accordance with the above explanation of a free variable proof. Definition 9 seems to reflect the intrinsically linguistic character of logical reasoning[43], since it defines the meaning of a logical consequence, that is, a hypothetical judgment, by characterizing the linguistic forms of derivations of the logical consequence.

We now turn to the directions we wish to pursue in future. On the practical side, we wish to implement the NF system on a computer. As reported in [73], we

[40] See also Frege [23, page 659].

[41] Namely, the explanation of what count as the proofs (derivations) of these forms of judgments.

[42] Here he is considering a hypothetical judgment of the form $J\ (H_1, \ldots, H_n)$.

[43] With regard to this claim, the reader is referred to Okada [56] who analyzes Hilbert's and Husserl's philosophy of mathematics.

have been teaching a computer aided course on computation and logic at Kyoto University. The computer environment for the course is called CAL and through the teaching experience of the course and through the implementation experience of the CAL system we have gradually arrived at the notion of derivation game. The current implementation of CAL does not support the user definition of a derivation game and moreover each derivation game is separately implemented. The NF system, when implemented, will change the situation.

We will then be able to define NF within NF system in a somewhat similar manner as McCarthy 'defined' pure Lisp by writing a meta-circular interpreter [51] of pure Lisp in pure Lisp[44]. The technical device needed is a method to encode S (schemata) into E (expressions) since rule-schemata are elements in S and judgments and derivations are elements in E. McCarthy used the method of quotation in [51] which is much more efficient than Gödel numbering in terms of space ratio between original expressions and encoded expressions. We are planning to use the mechanism of quasi-quotation invented by Quine [62] and used in several dialects of modern Lisp systems, for instance, Scheme [2].

As described in the previous paragraph, we will be able to use the NF system to *define* (or *specify*) computational and logical systems as deriviation games. Then, for each such derivation game G, NF will function as a *proof checker*, and it will check for any expression D whether it is a correct derivation in G. However, demanding the user to manually supply a derivation in a complete form will discourage the use of the system. Therefore, it will be important to provide various means of automating the construction of a derivation. For elementary derivation games, we would be able to fully automate the construction by adding the unification and the backtracking mechanisms to the NF system. Such an NF system can be used as a generic interpreter of the language specified by G. For non-elementary derivation games, addition of the mechanism of higher-order unification (see, Huet [37]) as used, for instance, in the systems described in [39, 53] would become necessary.

As for the theoretical side, we have seen that our theory presented here is more foundational than other frameworks. However, we think that the theory should be made more foundational in at least two points.

The first point is concerned with the fact that we have *not* defined the notion of α-equivalence and substitution on expressions and schemata but relied on the (supposedly) common and standard definitions of these notions. But, as is well known, these notions are very subtle, and, in spite of this, we must choose specific definitions when we actually implement NF on a computer. However, none of existing definitions proposed by others, including, for instance, definitions using de Bruijn indices [12] are satisfactory to us.[45] Since we have our own alternative definitions of substitution and α-equivalence [65–68, 72], we would like to use them to define these subtle notions.

The second point is related to a recent trend in computer science to internalize meta-level syntactic operations and make such operations available at the

[44] The interpreter was incorrect and corrected in [52] (see [78]).
[45] See also [73].

level of programming languages. Such a trend was geared up by the work of Abadi et al. [1] which introduced the notion of *explicit substitution*. Since then explicit substitution has been extensively studied and other related notions such as *environment* [54, 71] and *context* [8, 11, 32, 50, 59, 63, 72, 85] has been studied. The notion of context here is not the one we used to define expressions, but a context is (intuitively) an expression with holes in it. When the holes are filled with another expression, capturing of variables are not avoided but rather solicited. We can find a perfect example of a context in the mathematical induction rule we considered in §1. The P, which we also write $P[\ ]$, in the rule is such a context, and in the conclusion of the rule, the holes are filled with the variable x and these x's become bound by the quantifier $\forall x$ as expected. As we showed in §1, the induction rule is represented by the rule-schema $\mathtt{ind}(\boldsymbol{P})$ in NF, and when the rule is actually applied, the schematic variable $\boldsymbol{P}$ must be instantiated by an abstract. This means that an abstract expression of NF is nothing but a context. In Edinburgh LF, the context P is realized by a higher-order function ϕ of type $\iota \to o$, and Pitts [59] and Sands [63] used such higher-order functions to represent contexts. We believe that our identification of *context-as-abstract* is more natural than the representations of contexts by higher-order functions. However, although abstracts are expressions in E, we cannot internally instantiate (that is, fill in) the abstracts in the current formulation of NF, since instantiation is a meta-level operation. To internalize the instantiation operation, we must, again, face the problem of encoding schemata into expressions.

These observations seem to be suggesting us that we will be able to get a more foundational and powerful theory by re-designing the structure of expressions by adding the mechanism of quotation or quasi-quotation. This is, however, only the author's personal speculation. Even for the Natural Framework presented here, there remains a lot to be done, and among which semantical analysis of foundational notions such as *proposition*, *truth*, *type* and *set* by means of derivation games seems to be most interesting and challenging from a theoretical point of view.

Acknowledgements

The author would like to thank John McCarthy and Per Martin-Löf for giving him so much inspiration and philosophical influence through conversations and through their articles.

The author would like to thank Setsuo Arikawa who conceived the concept *discovery science* and led a three year project on discovery science under the sponsorship of the Ministry of Education, Culture, Sports, Science and Technology, Japan. In these three years, the author had the opportunity to organize a group consisting of philosophers, logicians and computer scientists, and discuss freely on foundational issues in discovery science. The focus of attention of the group was the analysis of discovery of concepts by human while other groups mainly studied discovery by machine. The author is grateful to Hiroakira Ono, Keiich Noé, Mitsuhiro Okada, Tetsuo Ida, Masami Hagiya, Yukiyoshi Kameyama and Syun Tutiya who shared many hours of discussions on discovery of concepts with him.

The author benefitted from discussions with many people on matters related to the present work. In particular, the author wishes to thank the following people: Peter Aczel, Michael Beeson, Henk Barendregt, Rod Burstall, Robert Constable, Thierry Coquand, Peter Dybjer, Solomon Feferman, Shigeki Goto, Susumu Hayashi, Rodger Hindley, Gerard Huet, Shigeru Igarashi, Leslie Lamport, Robin Milner, J. Strother Moore, Gopalan Nadathur, Minoru Nagamitsu, Bengt Nordström, Atsushi Ohori, Gordon Plotkin, Randy Pollack, Takafumi Sakurai, Mikio Sato, Andre Scedrov, Dana Scott, Jan Smith, Carolyn Talcott, Masako Takahashi, Satoru Takasu, Gaisi Takeuti, TAKEUTI Izumi, Yoshihito Toyama, Richard Weyhrauch, Yohei Yamasaki, and Mariko Yasugi.

Finally, the author would like to thank Takayasu Ito who carefully read the earlier drafts of the paper and provided useful comments and advices that contributed to improve the paper.

References

1. Abadi, M., Cardelli, L., Curien, P.-L. and Levy, J.-J., Explicit substitutions, pp. 375-416, *Journal of Functional Programming*, **1**, 1991.
2. Abelson, H. et. al, Revised^5 report on the algorithmic language scheme, *Higher-order and symbolic computation*, **11**, pp. 7 – 105, 1998.
3. Aczel, P., Frege structures and the notions of proposition, truth and set, in Barwsise, J. et al. eds., *The Kleene Symposium*, North-Holland, Amsterdam, pp. 31 – 50, 1980.
4. Aczel, P., Carlisle, D.P., Mendler N., Two frameworks of theories and their implementation in Isabelle, pp. 3 – 39, in [38], 1991.
5. Barendregt, H. P., *The Lambda Calculus, Its Syntax and Semantics*, North-Holland, 1981.
6. Beeson, M.J., *Foundations of constructive mathematics*, Springer, 1980.
7. Bell, D., *Frege's Theory of Judgement*, Clarendon Press, Oxford, 1979.
8. Bognar, M., de Vrijer, R., A calculus of lambda calculus contexts, **27**, pp. 29–59, *J. Automated Reasoning*, 2001.
9. Constable R.L. et al., *Implementing Mathematics with the NuPRL Proof Development System*, Prentice-Hall, Englewood Clifs, NJ, 1986.
10. Coquand, T. and Huet, G., The calculus of construction, *Information and Computation*, **76**, pp. 95 – 120, 1988.
11. Dami, L., A lambda-calculus for dynamic binding, pp. 201-231, *Theoretical Computer Science*, **192**, 1998.
12. de Bruijn, D. G., Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem, *Indag. Math.* 34, pp. 381–392, 1972.
13. de Bruijn, D. G., A survey of the project AUTOMATH, in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 579 – 606, Academic Press, London, 1980.
14. de Bruijn, D. G., A plea for weaker frameworks, pp. 40–67, in [38], 1991.
15. Church, A., A formulation of the simple theory of types, *J. Symbolic Logic*, **5**, pp. 56 – 68, 1940.

16. Dybjer, P., Inductive sets and families in Martin-Löf's type theory and their set theoretic semantics, pp. 280 – 306, in [38], 1991.
17. Feferman, S., A language and axiom for explicit mathematics, in Crossley J.N. ed., *Algebra and Logic*, Lecture Notes in Computer Science, **450**, pp. 87 – 139, 1975.
18. Feferman, S., Indutively presented systems and the formalization of meta-mathemaics, in van Dalen, D. et al. eds., *Logic Colloquim '80*, North-Holland, Amsterdam, 1982.
19. Feferman, S., Finitary inductively presented logics, in Ferro R. et al. eds., *Logic Colloquim '88*, North-Holland, Amsterdam, 1989.
20. Feferman, S., *In the Light of Logic*, Logic and Computation in Philosophy, Oxford University Press, Oxford, 1998.
21. Fiore, M., Plotkin, G., and Turi, D., Abstract syntax and variable binding (extended abstract), in *Proc. 14th Symposium on Logic in Computer Science*, pp. 193-202, 1999.
22. Frege, G., Über Begriff und Gegenstand, *Vierteljahrsschrift für wissenschftliche Philosophie*, **16**, pp. 192–205, 1892.
23. Frege, G., Was ist Funktion?, in *Festschrift Ludwig Boltzmann gewindmet zum sechzigsten Geburstage, 20. Feburar 1904*, Leipzig, pp. 656 – 666, 1904.
24. Genzten, G., Untersuchungen über das logische Schließen, I, Mathematische Zeitschrift, **39**, pp. 175 – 210, 1935, English translation in *The collected papers of Gerhard Gentzen*, Szabo, M.E. ed., pp. 68 – 131, North-Holland, Amsterdam, 1969.
25. Girard, J-Y., Taylor, P., Lafont, Y., *Proofs and Types*, Cambridge University Press, 1989.
26. Gödel, K., Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes, *Dialectica*, **12**, pp. 280 – 287, 1958.
27. Goodman, N., A theory of constructions equivalent to arithmetic, in *Intuitionism and Proof Theory*, Kino, Myhill and Vesley, eds., North-Holland, pp. 101 – 120, 1970.
28. Goto, S., Program synthesis through Gödel's interpretation, Lecutre Notes in Computer Science, **75**, pp. 302 – 325, 1978.
29. Goto, S., Program synthesis from natural deduction proofs, in *Sixth International Joint Conference on Artificial Intelligence (IJCAI-79)*, pp. 339 – 341, 1979.
30. Goto, S., How to formalize traces of computer programs, *Jumulage Meeting on Typed Lambda Calculi*, Edinbrugh, September, 1989.
31. Harper, R., Honsell, F. and Plotkin, G., A framework for defining logics, *Journal of the Association for Computing Machinery*, **40**, pp. 143–184, 1993.
32. Hashimoto, M., Ohori, A., A typed context calculus, *Theoretical Computer Science*, to appear.
33. Hayashi, S. and Nakano, H., *PX: A Computational Logic*, Foundation of Computing, The MIT Press, Cambridge, 1988.
34. Heyting, A.K., Intuitionism in mathematics, in *Philosophy in the mid-century, a survey*, Klibansky, ed., Florence, pp. 101 – 115, 1958.
35. Hilbert, D., Über das Unendliche, *Mathematische Annalen*, **95**, pp. 161 – 190, English translation in van Heijenoort, J. ed., *From Frege to Gödel: A Source Book in Mathematical Logic*, Harvard University Press, Cambridge, MA, 1967.
36. Howard, W.A., The formulae-as-types notion of construction, in Seldin J.P. and Hindley, J.R. eds., *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479 – 490, Academic Press, London, 1980.
37. Huet, G., A unification algorithm for typed λ-calculus, *Thoretical Computer Science*, **1**, pp. 27 – 57, 1975.

38. Huet, G., and Plotkin, G. eds., *Logical Frameworks*, Cambridge University Press, 1991.
39. Ida, T., Marin, M., Suzuki, T., Reducing search space in solving higher-order equations, in this volume.
40. Kobayashi, S., Consitency of Beeson's formal system RPS and some related results, in Shinoda, J., Slaman, T.A. and Tugué, T. eds., *Mathematical Logic and Applications, Proceedings of the Logic Meeting held in Kyoto, 1987*, Lecture Notes in Mathematics, **1388**, Springer, pp. 120 – 140, 1989.
41. Kleene, S.C., On the interpretation of intuitionistic number theory, *J. Symbolic Logic*, **10**, pp. 109 – 124, 1945.
42. Kreisel, G., Hilbert's programme, *Dialectica*, **12**, pp. 346–372, 1958.
43. Kreisel, G., Foundations of Intuitionistic Logic, Nagel, Suppes, Tarski eds, *Logic, Methodology and Philosophy of Science*, Stanford University Press, pp. 198 – 210.
44. Kreisel, G., Mathematical Logic, Saaty ed., *Lectures on Modern Mathematics, III*, Wiley, New York, pp. 95 – 195, 1965.
45. Martin-Löf, P., An intuitionistic theory of types: Predicative part, *Logic Colloquium '73*, Rose, H.E. and Shepherdson, J.C., Eds, Studies in Logic and Foundation of Mathematics, **80**, North-Holland, pp. 73 – 118, 1975.
46. Martin-Löf, P., *Intuitionistic Type Theory*, Bibliopolis, 1984.
47. Martin-Löf, P., Truth of a proposition, evidence of a judgment, validity of a proof, *Synthese*, **73**, pp. 407–420, 1987.
48. Martin-Löf, P., Analytic and synthetic judgments in type theory, Parrini, P. ed., *Kant and Contemporary Epistemology*, Kluwer Academic Publishers, pp. 87 – 99, 1994.
49. Martin-Löf, P., On the meanings of the logical constants and the justifications of the logical laws, *Nordic J. of Philosophical Logic*, **1**, pp. 11–60, 1996.
50. Mason, I., Computing with contexts, *Higher-Order and Symbolic Computation*, **12**, pp. 171–201, 1999.
51. McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, Part I, *Comm. ACM*, **3**, pp. 184–195, 1960.
52. McCarthy, J., et. al, LISP 1.5 programmer's manual, MIT Press, 1965.
53. Nadathur, G., Miller, D., Higher-order logic programming, in Gabbay, D.M. et al. eds., *Handbook of Logic in AI and Logic Programming*, **5**, Clarendon Press, Oxford, pp. 499 – 590, 1998.
54. Nishizaki, S., Simply typed lambda calculus with first-class environments, *Publ. RIMS, Kyoto U.*, **30**, pp. 1055 – 1121, 1994.
55. Nordström, B., Petersson,K. and Smith, J.M., *Programming in Martin-Löf's Type Theory*, Oxford University Press, 200 pages, 1990.
56. Okada, M., Ideal concepts, intuitions, and mathematical knowledge acquisitions: Husserl's and Hilbert's cases (a preliminary report), in this volume.
57. Ono, K., *Gainen-Taishou Riron no Kousou to sono Tetugakuteki Haikei* (Eng. *A plan of notion-object theory and its philosophical background*), Nagoya University Press, Nagoya, 1989.
58. Paulson, L., *Logic and Computation*, Cambridge University Press, Cambridge, 1988.
59. Pitts, A.M., Some notes on inductive and co-inductive techniques in the semantics of functional programs, Notes Series BRICS-NS-94-5, Department of Computer Science, University of Aarhus, 1994.
60. Pollack, R., A verified type-checker, in Dezani-Ciancaglini, M. and Plotkin, G. eds., *Proceedings of the Second International Conference on Typed Lambda Calculi and*

Applications, TLCA '95, Edinburgh, Springer-Verlag, Lecture Notes in Computer Science **902**, 1995.
61. Prawitz, D., *Natural Deduction: A Proof-Theoretical Study*, Almquist and Wiksell, Stockholm, 1965.
62. Quine, W.V.O., *Mathematical Logic*, Harvard University Press, 1951.
63. Sands, D., Computing with Contexts – a simple approach, in *Proc. Higher-Order Operational Techniques in Semantics, HOOTS II*, 16 pages, *Electronic Notes in Theoretical Computer Science* **10**, 1998.
64. Sato, M., Towards a mathematical theory of program synthesis, in *Sixth International Joint Conference on Artificial Intelligence (IJCAI-79)*, pp. 757 – 762, 1979.
65. Sato, M., and Hagiya, M., Hyperlisp, in de Bakker, van Vliet eds., *Algorithmic Languages*, North-Holland, pp. 251–269, 1981.
66. Sato, M., Theory of symbolic expressions, I, *Theoretical Computer Science*, **22**, pp. 19 – 55, 1983.
67. Sato, M., Theory of symbolic expressions, II, *Publ. of Res. Inst. for Math. Sci.*, Kyoto Univ., **21**, pp. 455–540, 1985.
68. Sato, M., An abstraction mechanism for symbolic expressions, in V. Lifschitz ed., *Artificial Intelligence and Mathematical Theory of Computation (Papers in Honor of John McCarthy)*, Academic Press, pp. 381–391, 1991.
69. Sato, M., Adding proof objects and inductive definition mechanism to Frege structures, in Ito, T. and Meyer, A. eds., *Theoretical Aspects of Computer Software, International Syposium TACS'94 Proceedings*, Lecture Notes in Computer Science, **789**, Springer-Verlag, pp. 179 – 202, 1994.
70. Sato, M., Classical Brouwer-Heyting-Kolmogorov interpretation, in Li, M. Maruoka, A. eds., *Algorithmic Learning Theory, 8th International Workshop, ALT'97, Sendai, Japan, October 1997, Proceedings*, Lecture Notes in Artificial Intelligence, **1316**, Springer-Verlag, pp. 176–196, 1997.
71. Sato, M., Sakurai, T. and Burstall, R., Explicit environments, *Fundamenta Informaticae* 45, pp. 79–115, 2001.
72. Sato, M., Sakurai, T. and Kameyama, Y., A simply typed context calculus with first-class environments, in *Proc. Fifth International Symposium on Functional and Logic Programming (FLOPS)*, Lecture Notes in Computer Science, **2024**, pp. 359–374, 2001.
73. Sato, M., Kameyama, Y., Takeuti, I., CAL: A computer assisted system for learning compuation and logic, in *EuroCAST 2001*, Lecture Notes in Computer Science,**2178**, 2001.
74. Scott, D., Constructive validity, *Symposium on automatic demonstration*, Lecture Notes in Mathematics, **125**, pp. 237 – 275, Springer, Berlin, 1969.
75. Simpson, S.G., Partial realization of Hilbert's program, *J. Symbolic Logic*, **53**, pp. 359 – 394, 1988.
76. Skolem, T., The foundations of elementary arithmetic established by means of the recursive mode of thought, without the use of apparent variables ranging over infinite domains, in [91], pp. 302 – 333.
77. Smorynski, C., The incompleteness theorems, Barwise, J. ed., *Handbook of mathematical logic*, Studies in logic and the foundations of mathematics, **90**, pp. 821 – 865, North-Holland, 1977.
78. Stoyan, H., The influence of the designer on the design – J. McCarthy and LISP, in V. Lifschitz ed., *Artificial Intelligence and Mathematical Theory of Computation (Papers in Honor of John McCarthy)*, Academic Press, pp. 409–426, 1991.
79. Smullyan, R., *Theory of Formal System*, Annals of Mathematics Studies, **47**, Princeton University Press, Princeton, 1961.

80. Sundholm, G., Constructions, proofs and the meaning of logical constants, *J. Phil. Logic*, **12**, pp. 151 – 172, 1983.
81. Tait, W.W., Finitism, *J. of Philosophy*, **78**, pp. 524 – 546, 1981.
82. Takeuti, G., Generalized Logic Calculus, *Journal of Mathematical Society of Japan*, 1954.
83. Takeuti, G., *Proof Theory*, North-Holland, Amsterdam, 1975.
84. Takeuti, G., A conservative extension of Peano arithmetic, *Two Applications of Logic to Mathematics*, Part II, Princeton University Press, and Publications of the Mathematical Society of Japan **13**, 1978.
85. Talcott, C., A Theory of binding structures and applications to rewriting, *Theoretical Computer Science*, **112**, pp. 99-143, 1993.
86. Tatsuta, M., Program synthesis using realizability, *Theoretical Computer Science*, **90**, pp. 309 – 353, 1991.
87. Troelstra, A.S., Realizability and functional interpretation, in Troelstra, A.S. ed., *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, Lecture Notes in Mathematics, **344**, Springer-Verlag, 1973.
88. Troelstra, A.S. and van Dalen, D., *Constructivism in Mathematics, An Introduction, I*, North-Holland, Amsterdam, 1988.
89. Troelstra, A.S., Schwichtenberg, H., *Basic Proof Theory*, Cambridge University Press, 1996.
90. Tsukada, Y., Martin-Löf's type theory as an open-ended framework, *International J. of Foundations of Computer Science*, **12**, pp. 31 – 68, 2001.
91. van Heijenoort, J., *From Frege to Gödel, A source Book in Mathematical Logic, 1879 – 1931*, Harvard University Press, Cambridge, MA, 1967.
92. Voda, P., Theory of pairs, part I: provably recursive functions, *Technical Report of Dept. Comput. Science*, UBC, Vancouver, 1984.
93. Voda, P., Computation of full logic programs using one-variable environments, *New Generation Computing*, **4**, pp. 153 – 187, 1986.

To appear in Arikawa, S. and Shinohara, A. eds., *Progress in Discovery Science*, Lecture Notes in Artificial Intelligence, State-of-the-Art Surveys, Springer, 2002.