CAL: A Computer Assisted Learning system for Computation and Logic

Masahiko SATO[1], Yukiyoshi KAMEYAMA[2], and TAKEUTI Izumi[1]

[1] Graduate School of Informatics, Kyoto University
{masahiko,takeuti}@kuis.kyoto-u.ac.jp
[2] Institute of Information Sciences and Electronics, University of Tsukuba
kam@is.tsukuba.ac.jp

Abstract. In this paper, we describe a computer system CAL which assists students' understanding of a lecture course in logic and computation. We introduce a new foundational theory of expressions, which is inspired by the theory of expressions due to Martin-Löf [6], and use the expressions determined by the theory to implement various syntactic objects in CAL. We also introduce the concept of *derivation game*, and show that various logical and computational systems given in the lecture course can be defined uniformly by derivation games.

1 Introduction

In this paper, we describe a computer assisted learning system for computation and logic. The system is intended to complement a lecture course on Computation and Logic given to undergraduate students of Kyoto University. All the students are asked to use the system in order to deepen their understanding of the contents of the lecture. The lecture course is an introduction to formal logics. Since most of the students are familiar with a number of programming languages but not familiar with formal logics, we designed our lecture course by stressing the similarity between programming and proving. We used the Curry-Howard isomorphism to achieve this design goal.

Traditionally, courses in logic are taught in class rooms and exercises are done by paper and pencil. However, in view of the facts that the subject matter of the course consists of formal syntactic entities which can be manipulated formally, we think that it is important and necessary to provide a computer environment in which students can solve problems and the computer can tell if the students solved the problems correctly. Although there are already several computer software like Tarski's World [2] for learning logic, we have designed and implemented our own system since existing software does not cover the materials we wish to teach.

The name of the lecture course is "Computation and Logic" and it covers the following formal systems.

1. Intuitionistic propositional logic.
2. Simply typed λ-calculus.

3. Reduction of derivations in the intuitionistic propositional logic and terms in the simply typed λ-calculus.
4. Heyting arithmetic.
5. Typed λ-calculus with dependent types.
6. Reduction of derivations in the Heyting arithmetic and terms in the dependently typed λ-calculus.

The name of the computer environment for the course is CAL. CAL implements all the formal systems we teach in the course. Therefore students can use CAL to play with all the systems listed above. CAL is entirely implemented in Emacs Lisp and runs under GNU Emacs. GNU Emacs is a very powerful text editor and it has its own programming language Emacs Lisp which we used to implement CAL. A student can invoke CAL from within Emacs, and when CAL is invoked, a special buffer is created and then the user and the CAL system can interact with each other in this buffer. As a system, CAL consists of the following three parts.

1. User interface
2. Parser
3. Checker

We will explain each part in more detail in section 4.

The fundamental purpose of our lecture course is to convey students the facts that formal systems taught at the course are, or at least should be, reflections of the informal logical or mathematical systems they have already learned in (informal) mathematics. Therefore, we tried to reflect the informal mathematical practices we carry out in daily mathematics as naturally and precisely as possible. This approach forced us to formally present several mathematical practices that are usually left at the informal meta level. For example, in traditional logic text books, the concept of abbreviations are only informally sketched as a meta level technical means to simplify notations for formal syntactic objects like terms or formulas. In our course we have defined abbreviations formally so that students can actually check on a computer whether they understand what expressions abbreviate what expressions.

For this lecture we prepared both the course note which describes the formal systems given above and the software system CAL which implements these formal systems. Although these formal systems are standard ones, we have fine tuned them so that they can be presented uniformly and also they can be smoothly manipulated formally on a computer. In particular, to achieve the fundamental purpose mentioned above, we have identified informal logic and mathematics as human mental activities to arrive at sound logical or mathematical *judgments*, and we formalized judgments as formal expressions which we can derive by constructing derivations whose conclusions are the given judgments. Each formal system we introduce in the course has its own unique form of judgments and derivations, but they are presented in a uniform way as we will describe below.

In order to achieve the uniform presentations of the formal systems, we use two general defining mechanisms to define formal systems. The first one is what we call *definition games*. Definition games are formalizations of a simple class of informal inductive definitions commonly used in mathematics. Each definition game, say G, defines a class of G-derivations and a class of G-judgments. A G-judgment J is said to be *provable* (or, *derivable*) if there is a G-derivation whose conclusion is J. For example, in `NatNum`, which is the first definition game we teach in our course, we can derive the `NatNum`-judgment `s(s(0)) : Nat` as follows.

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

The second defining mechanism is called *derivation games*. While definition games are used to derive judgments without hypotheses, derivation games are used to derive *hypothetical judgments* which are judgments that depend on several hypotheses. Derivation games have built-in mechanisms to introduce and discharge hypotheses. Using derivation games we can define several natural deduction systems in a uniform way.

The paper is organized as follows. In section 2, we explain *expressions* which provide us with an abstract way of uniformly presenting syntactic objects we teach in the course. The same abstract syntax is used to implement the CAL system, so that we do not have the discrepancy usually found in the implementations of syntactic objects using de Bruijn notation. In section 3, we define definition games and derivation games, and give some examples of them. Section 4 explains the CAL system from the implementation point of view, and section 5 concludes the paper.

2 Theory of Expressions

In informal mathematics, we often explain concepts using "dots". For example:

Natural numbers are defined as $0, s(0), s(s(0)), \cdots$.

While this description helps readers' intuitive understanding, it does not constitute a precise definition, and there is always possibility of misunderstanding by the readers. In fact, students can easily get lost in manipulating formal objects, and the reason is often that they could not understand the definitions correctly since they were presented to the students in an informal and ambiguous way. In particular, we believe that the concepts of *variable binding* and *substitution* are the most fundamental concepts we have to teach in any logic course; however, they are rarely taught rigorously for the technical reasons we explain later in this section.

Based on this observation, we had decided to design our lecture course so that each and every mathematical concept will be presented to students in a rigid and formal manner. A trade-off of the decision is that we must use a great number of formal inductive definitions when giving each formal system, and we were naturally led to introduce two *definition mechanisms* of formal definitions and a *theory of expressions* which provides us with a natural and convenient data structure for defining the concepts of variable binding and substitution.

The two definition mechanisms we introduced are *definition games* and *derivation games*. Definition games are used to define operations and relations on syntactic objects in a manner similar to that of structured operational semantics. For example, expressions and substitution operation on expressions are defined by the definition games `Exp` and `Subst`, respectively. Derivation games are used to define various logical systems and we will explain them in the next section.

The above design decision enabled us to present various formal systems uniformly and naturally. This design decision also brought us the following good side-effects. Namely, the course note we prepared served not only as a text book but also both as the user's manual for students when they use the CAL system and as the formal specification of the CAL system which had to be implemented by us.

2.1 Definition Games

A definition game G consists of a finite number of *rule schemata*. Each schema is of the following form where k is a non-negative integer, $C, P_1, \cdots, P_k$ are expressions possibly containing schematic variables ranging over sequences of words[1] and N is a word:

$$\frac{P_1 \quad \cdots \quad P_k}{C}\ N$$

Here, N, C, P_i are called the *name*, the *conclusion*, and the *assumptions* of the rule schema, respectively. For example, the definition game `NatNum` is determined by the following two rule schemata, where n is a schematic variable.

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

Given a definition game G, a *G-derivation* and *its conclusion* is defined as follows. Let $\mathcal{D}_1, \ldots, \mathcal{D}_k$ $(k \geq 0)$ be G-derivations whose conclusions are $P_1, \ldots, P_n$ respectively and

$$\frac{P_1 \quad \cdots \quad P_k}{C}\ N$$

be an instance of a rule schema of G, then

$$\frac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_k}{C}\ N$$

is a G-derivation whose conclusion is C. As an example, we already gave a derivation of $\mathtt{s(s(0))} : \mathtt{Nat}$ in the introduction.

[1] We assume that "sequences of words" are appropriately defined.

2.2 Expressions

In CAL, we introduce the concept of *expression* to uniformly represent formal syntactic objects we teach. The set of expressions is freely generated from constants and variable references[2] by means of the two operations[3] of *list formation* ($\mathsf{list}(\_)$) and *abstraction* ($\mathsf{abs}_x(\_)$) of an expression with respect to a variable x.

Two expressions are defined to be *definitionally equal* if they are generated in exactly the same way by these operations. We write $E \equiv F$ if E and F are definitionally equal. Thus, if $E_1, \ldots, E_n$ and $F_1, \ldots, F_m$ are expressions, then $\mathsf{list}(E_1, \ldots, E_n) \equiv \mathsf{list}(F_1, \ldots, F_m)$ iff $m = n$ and $E_i \equiv F_i$ for each i $(1 \le i \le m)$, and if x, y are variables and E, F are expressions, then $\mathsf{abs}_x(E) \equiv \mathsf{abs}_y(F)$ iff $x \equiv y$ and $E \equiv F$.

We can then define the operations of *substitution* ($\mathsf{subst}(\_, \_, \_)$) and *instantiation* ($\mathsf{inst}(\_, \_)$) in such a way that they enjoy, among others, the equality:

$$\mathsf{inst}(\mathsf{abs}_x(E), F) \equiv \mathsf{subst}(F, x, E).$$

Existing methods of constructing a structure on which abstraction and substitution operations are properly defined are not at all easy and therefore very difficult to teach. For example, the classical way of teaching the syntax and the β-reduction rule of the untyped λ-calculus is like this. First, expressions are introduced as *strings* of characters, where a character is either a variable or a constant like 'λ', and a string is of the form $c_1 \cdots c_n$ $(n \ge 0)$ where each c_i is a character. Two strings $c_1 \cdots c_n$ and $d_1 \cdots d_m$ are defined to be *definitionally equal* if $m = n$ and c_i and d_i are the same characters. The concatenation ($\mathsf{conc}(\_, \_)$) of two expressions is defined by juxtaposition and $\mathsf{conc}(E, F)$ is simply written as EF. The abstraction operation is then defined by putting $\mathsf{abs}_x(E) \equiv \mathsf{conc}(x, \mathsf{conc}(\text{'.'}, E)) \equiv x.E$. The operation of substituting F for x in E is defined as the operation of textual replacement of each free occurrence of x in E with F. But, this operation works well only when no free variables in E become bound after the substitution. It is therefore necessary to re-define the definitional equality relation by introducing the concept of α-equivalence. Even if we could successfully introduce the syntax in this way, we must continue to use concrete syntax when we actually write down λ-terms. Therefore, we must define substitution operation on terms in concrete syntax, but this definition requires us to check that the operation respects α-equivalence.

There are other methods by de Bruijn [3] and by us [7, 9]. They both remove the variable x from E when $\mathsf{abs}_x(E)$ is constructed and do not rely on the concept of α-equivalence when defining expressions. But these methods are not suitable for teaching for the following reasons. First of all, if we write concrete λ-terms using these notations, then they are almost unreadable. We can make them readable again by introducing traditional syntax with the definitional equality as strings and define a translation mapping from the terms in traditional syntax

[2] We will explain *variable references* later.

[3] The second operation is actually a schema of operations since it contains x as a parameter.

to the terms in de Bruijn notation or in our notation [7, 9]. But, if we do so, we must again define substitution operation on terms written in concrete syntax and must show that the operation is well-defined.

Although de Bruijn notation is not suitable for humans, it is often used in the implementations of logical systems because of the simplicity of the substitution operation. But such implementations have the following drawback. When the system prints out an expression, it must use the external syntax, but since bound variables have no names in de Bruijn notation, the system must choose an arbitrary name like X3141.

Our approach uses the expressions which we informally introduced at the beginning of this subsection. It is a hybrid of the traditional approach and de Bruijn's approach and it inherits only good points of these approaches. That is, our syntax is human readable and substitution can be defined without relying on the concept of α-equivalence. Moreover, unlike de Bruijn notation, λ-terms in traditional syntax can be expressed in our syntax without any modifications. Because of these good properties, our expressions are implemented by the CAL system as Lisp symbolic expressions, and the implementation establishes an isomorphism between expressions we teach and their representations by Lisp symbolic expressions. So, we could implement substitution on a computer in an isomorphic way as we teach substitution in the lecture course.

Having this mechanism in mind, we can define expressions and substitution over them as definition games. We first assume that there are countably many *variables* denoted by x, y, z (possibly with subscripts) and *constants* denoted by c. A *variable reference* is a pair (n, x) of a natural number n called *reference counter* and a variable x, and we will write $\texttt{\#}^n x$ for the variable reference (n, x). A reference counter plays a similar role as a de Bruijn index. A *declaration* is a finite sequence of variables $x_1, \cdots, x_n$ $(n \geq 0)$ denoted by Γ where duplication of a variable is allowed.

The definition game `Exp` consists of the following four rule schemata:

$$\frac{}{\Gamma \ \texttt{exp} \ \texttt{\#}^n x} \ \texttt{var} \qquad \frac{}{\Gamma \ \texttt{exp} \ c} \ \texttt{const}$$

$$\frac{\Gamma \ \texttt{exp} \ E_1 \quad \cdots \quad \Gamma \ \texttt{exp} \ E_n}{\Gamma \ \texttt{exp} \ \mathsf{list}(E_1, \ldots, E_n)} \ \texttt{struct} \qquad \frac{\Gamma, x \ \texttt{exp} \ E}{\Gamma \ \texttt{exp} \ \mathsf{abs}_x(E)} \ \texttt{abs}$$

where in the `var` rule, n is a non-negative integer and Γ must contain at least $n+1$ occurrences of the variable x. In the `var` rule, $\texttt{\#}^n x$ refers to the $n+1$-st occurrence of x in Γ counting from the right. We used dots in the `struct` rule schema, but since the meaning of these dots can be explained rigorously by induction on n, the usage is permitted. If there is an `Exp`-derivation whose conclusion is Γ `exp` E, we say that E is an *expression under* Γ. Note that in this case any variable reference $\texttt{\#}^n x$ occurring in E refers either to a unique occurrence of x in Γ or to the parameter x of a unique binder $\mathsf{abs}_x(\_)$ that contains the $\texttt{\#}^n x$ in its scope. We say that the occurrence of $\texttt{\#}^n x$ in E is *free* in the first case, and *bound* in the second case.

Similarly, substitution can be defined by a definition game. However, due to lack of space, we explain substitution here informally. Let E be an expression under Γ, and Δ be a declaration. Then, we can construct an expression F under Γ, Δ as follows. For each occurrence of variable references $\texttt{#}^n x$ in E, we compute a non-negative integer k as follows. If the occurrence is bound in E, then we put $k = 0$. If it is free in E, then we put k to be the number of occurrences of x in Δ. We then replace the occurrence of $\texttt{#}^n x$ in E with $\texttt{#}^{n+k} x$. F is obtained from E in this way. It is easy to check that F is an expression under Γ, Δ. We will call F the result of *pushing E through Δ*, and write $E \uparrow \Delta$ for F. For example, if x and y are distinct variables and $E \equiv \mathsf{abs}_x(\mathsf{list}(\texttt{#}^0 x, \texttt{#}^1 x, \texttt{#}^2 x, \texttt{#}^0 y))$, then E is an expression under x, x, y and we have:

$$F \equiv E \uparrow x, x \equiv \mathsf{abs}_x(\mathsf{list}(\texttt{#}^0 x, \texttt{#}^3 x, \texttt{#}^4 x, \texttt{#}^0 y)),$$

which is an expression under x, x, y, x, x.

Now, given a variable reference $\texttt{#}^n x$ and expressions E and F, we define an expression $[\texttt{#}^n x := F](E)$ inductively as follows.

1. $[\texttt{#}^n x := F](\texttt{#}^m y) \equiv \begin{cases} \texttt{#}^m y & \text{if } y \not\equiv x, \\ \texttt{#}^m y & \text{if } y \equiv x \text{ and } m < n, \\ F & \text{if } y \equiv x \text{ and } m = n, \\ \texttt{#}^{m-1} y & \text{if } y \equiv x \text{ and } m > n. \end{cases}$
2. $[\texttt{#}^n x := F](c) \equiv c$.
3. $[\texttt{#}^n x := F](\mathsf{list}(E_1, \ldots, E_m)) \equiv \mathsf{list}([\texttt{#}^n x := F](E_1), \ldots, [\texttt{#}^n x := F](E_m))$.
4. $[\texttt{#}^n x := F](\mathsf{abs}_y(E)) \equiv \mathsf{abs}_y([\texttt{#}^n x \uparrow y := F \uparrow y](E))$.

We define the substitution operation by putting $\mathsf{subst}(F, x, E) \equiv [\texttt{#}^0 x := F](E)$. For example, we have:

$$\mathsf{subst}(\texttt{#}^1 y, x, \mathsf{abs}_x(\mathsf{list}(\texttt{#}^0 x, \texttt{#}^1 x))) \equiv \mathsf{abs}_x(\mathsf{list}(\texttt{#}^0 x, \texttt{#}^1 y))$$

and

$$\mathsf{subst}(\texttt{#}^2 x, x, \mathsf{abs}_x(\mathsf{list}(\texttt{#}^0 x, \texttt{#}^1 x))) \equiv \mathsf{abs}_x(\mathsf{list}(\texttt{#}^0 x, \texttt{#}^3 x)).$$

Note that, if there are no name clashes in variables (as is usual when we write mathematical expressions), all variable references are in the form $\texttt{#}^0 x$, so we do not have to care about the reference counters.

In the concrete syntax of CAL, we write `<`$E_1, \ldots, E_n$`>` for $\mathsf{list}(E_1, \ldots, E_n)$ and `(`x`)[`E`]` for $\mathsf{abs}_x(E)$, and $\texttt{#}^n x$ is written by prefixing n #'s in front of x.

3 Derivation Games

Derivation games are the second defining mechanism in CAL, and are used to derive hypothetical judgments. As advocated in Martin-Löf's type theory, we view a derivation in object logics as a derivation of hypothetical judgements, hence, all object logics (and computational systems) in CAL will be defined by derivation games. This applies not only to the derivability but also the formations of propositions and terms in arithmetic.

In this section we first introduce the general mechanism of derivation games over expressions, and after that we quickly overview the definitions of several logical systems in terms of derivation games.

3.1 Derivation Games in General

A *hypothesis* is an expression $x : E$[4] where x is a variable and E is an expression. A *hypothesis sequence*, denoted by Γ, is a finite sequence of hypotheses. Then a *hypothetical judgment* is an expression $\Gamma \vdash E$ where E is an expression.

A *derivation rule* is in the following form:

$$\frac{E_1 \quad \cdots \quad E_n}{E}\ R$$

where $E, E_1, \cdots, E_n$ are expressions and R is a constant.

A *derivation game* G is determined in two steps. First, we specify a set of hypotheses in this derivation game, which we call G-hypotheses. Then, we specify, for each G-hypothesis-sequence, a set of derivation rules which can be used in this derivation game.

One may consider this definition is too complex, but the dependency of the second step on G-hypothesis-sequence cannot be eliminated in our setting. Let us take a simple example. Suppose we want to define the notion of provability of propositions in propositional logic as a derivation game. Then the hypotheses we may use in this game are either $A : \texttt{Prop}$ (A is a proposition) or $x : P$ (x is a proof of P). Under the hypothesis sequence $\Gamma \equiv \texttt{A} : \texttt{Prop}, \texttt{x} : \texttt{A}$, we can apply the implication-introduction rule to prove $\texttt{A} \supset \texttt{A}$, but not $\texttt{B} \supset \texttt{A}$, since the latter is not a proposition under Γ[5]. So the applicability of a derivation rule depends on the hypothesis sequence under which we apply the rule.

Given a derivation game G, a *G-derivation under* a G-hypothesis-sequence and its *conclusion* are defined by the following three clauses:

1. If $x : E$ is in Γ, then x is a G-derivation under Γ, and its conclusion is E.
2. If
$$\frac{E_1 \quad \cdots \quad E_n}{E}\ R$$
is a derivation rule in G, and $\mathcal{D}_i$ is a G-derivation under Γ whose conclusions are E_i (for $1 \le i \le n$), then
$$\frac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_n}{E}\ R$$
is a G-derivation under Γ whose conclusion is E.
3. If $\mathcal{D}$ is a G-derivation under Γ, Δ whose conclusion is E, then
$$\texttt{(}\Delta\texttt{)}\,\texttt{[}\mathcal{D}\texttt{]}$$
is a G-derivation under Γ and its conclusion is $\texttt{(}\Delta\texttt{)}\,\texttt{[}E\texttt{]}$.

[4] This is actually an abbreviation of the expression $\mathsf{list}(`{:}', x, E)$ where ':' is a constant. We will use similar abbreviations without explaining what expressions they will stand for, since such details will not concern us. In CAL, even abbreviations are formally defined using definition games.

[5] We cannot show that $\texttt{B} : \texttt{Prop}$ from Γ.

The first clause is a start-rule like one in the natural deduction style logic. Namely, assuming A, we have a proof A with conclusion A. The second is an ordinary inference rule. The third is a rule for discharging hypotheses. We shall show examples to illustrate their usage in the following subsections.

If there is a G-derivation $\mathcal{D}$ under Γ whose conclusion is E, then we say that $\mathcal{D}$ is a *derivation* of the hypothetical judgment $\Gamma \vdash E$ *in* G.

3.2 Propositional Logic

To define propositional logic, we need two[6] derivation games, `PropFrmn` for the *formation* of propositions and `PropProof` for the *provability* of propositions.

In the derivation game `PropFrmn`, an expression of the form A : `Prop` is allowed as a hypothesis where A is a variable, and `Prop` is a constant. Instead of enumerating all the rules of this game, we show an example of `PropFrmn`-derivation under the hypothesis sequence $\mathtt{A : Prop, B : Prop}$:

$$\dfrac{\mathtt{A} \quad \dfrac{\mathtt{B} \quad \mathtt{A}}{\mathtt{B \supset A : Prop}}\ \mathtt{imp\_frmn}}{\mathtt{A \wedge (B \supset A) : Prop}}\ \mathtt{and\_frmn}$$

This is a `PropFrmn`-derivation of the following hypothetical judgment:

$$\mathtt{A : Prop, B : Prop} \vdash \mathtt{A \wedge (B \supset A) : Prop}$$

For the derivation game `PropProof`, a hypothesis can be either of the form A : `Prop` or of the form $x : P$ where A and x are variables and P must be a proposition. Again, we avoid enumerating all the derivation rules, and only show a few cases of `PropProof`-derivations.

Let Γ be a `PropProof`-hypothesis-sequence, P, Q be propositions under Γ and x be a variable. Then, the following two rules are the introduction and elimination rules of the implication under Γ.

$$\dfrac{(x : P)\,[Q]}{P \supset Q}\ \mathtt{imp\_intro} \qquad \dfrac{P \supset Q \quad P}{Q}\ \mathtt{imp\_elim}$$

The hypothesis $x : P$ in the `imp_intro` rule is discharged as in the ordinary natural deduction system. A characteristic point in our formulation is that we may choose *any* variable for x while in the ordinary formulation, the label for the discharged assumption should be different from other labels, although this condition is often left implicit.

As an example, here is a `PropProof`-derivation of the hypothetical judgment $\mathtt{A : Prop, B : Prop} \vdash \mathtt{A \supset B \supset A}$.

$$\dfrac{\mathtt{(x : A)} \left[\dfrac{\mathtt{(y : B)\,[x]}}{\mathtt{B \supset A}}\ \mathtt{imp\_intro}\right]}{\mathtt{A \supset B \supset A}}\ \mathtt{imp\_intro}$$

[6] Besides the derivation game for abbreviations.

3.3 Simply Typed Lambda Calculus

It is now straightforward to define the simply typed lambda calculus by means of a derivation game. To pick up the corresponding rules to propositional logic, we have the following rules:

$$\frac{(x : P)\,[M : Q]}{\lambda(x : P)\,[M] : P \supset Q}\ \texttt{lambda\_frmn} \qquad \frac{M : P \supset Q \quad N : P}{M\,(N) : Q}\ \texttt{apply\_frmn}$$

where $\lambda(x : P)\,[M]$ is λ-abstraction and $M\,(N)$ is functional application.

3.4 Computational Aspect of Logic

One of the main goals of our lecture course is to establish the Curry-Howard isomorphisms between certain logical systems and computational systems. In section 1, we listed 6 items we teach in the course. Item 3 establishes the Curry-Howard isomorphism between the intuitionistic propositional logic and the simply typed lambda calculus we teach in 1 and 2 respectively. In item 6, we extend this isomorphism between the Heyting arithmetic and the dependently typed lambda calculus we teach in 4 and 5 respectively. We show that the isomorphisms are not only bijective but also compatible with the reduction relations defined on the logical systems and computational systems. In this way, we can show students that proving and programming can be seen as essentially the same mental activities under a suitable setting.

More precisely, in item 3, we introduce two definition games `LambdaTermRed` and `PropProofRed`. The former defines the reduction relation on lambda terms given in 3.3, and the latter defines that of derivations of propositional logic given in 3.2. Here, we pick up the β-reduction rule as a typical example. In `LambdaTermRed`, let Γ be a hypothesis sequence, let P be a type and M be a simply typed lambda term under Γ, let x be a variable and L be a simply typed lambda term under $\Gamma, x : P$, and let $N \equiv \mathsf{subst}(M, x, L)$. Then we have the following rule schema:

$$\frac{}{\lambda(x : P)\,[L]\,(M)\ \ \texttt{red}\ \ N}\ \texttt{beta}$$

In `PropProofRed`, let Γ be a hypothesis sequence, let P and R be propositions under Γ, let $\mathcal{D}$ be a `PropProof`-derivation under Γ whose conclusion is P, and let x be a variable and $\mathcal{D}_R$ be a `PropProof`-derivation under $\Gamma, x : P$ whose conclusion is R. Then letting $\mathcal{E}$ be the following derivation:

$$\frac{\dfrac{(x : P)\,[\mathcal{D}_R]}{P \supset R}\ \texttt{imp\_intro} \qquad \mathcal{D}}{R}\ \texttt{imp\_elim}$$

we have the following rule schema:

$$\frac{}{\mathcal{E}\ \ \texttt{red}\ \ \mathsf{subst}(\mathcal{D}, x, \mathcal{D}_R)}\ \texttt{imp}$$

These two rule schemata are identical (except the names of rule schemata) if we identify the construction of lambda terms and derivations of propositional logic. Although the Curry-Howard isomorphism is not easy to understand, students can gradually grasp the essence by repeatedly solving problems given in both games.

We note that the reduction rule schema `imp` uses the substitution operation defined for expressions. In contrast to this, in most logical systems, some special substitution operation for derivations is necessary to define the reduction of derivations.

3.5 Heyting Arithmetic

Heyting arithmetic is defined as the derivation game `ArithProof` in CAL. Among the rule schemata in `ArithProof`, a typical rule schema is ∃-elimination:

$$\frac{\exists(x)\,[P] \quad (k:\mathtt{Nat}, X:\mathsf{inst}((x)\,[P]\!\uparrow\! k, k))\,[R\!\uparrow\! k]}{R}\ \mathtt{exist\_elim}$$

where Γ is a hypothesis sequence, x, k, X are variables, P is an arithmetical proposition under $\Gamma, x:\mathtt{Nat}$, and R is an arithmetical proposition under Γ.

We note that, in the traditional presentation of first-order logic, a side condition (the so-called eigen variable condition) must be satisfied to apply the ∃-elimination rule. Since the eigen variable condition is given as a side condition and it is difficult to check, one often writes a derivation in which the condition is violated. In contrast to this, our rule schema has no side conditions as is shown above. Instead of having meta-level side conditions, we internalized the hidden conditions using the push-operation.

4 CAL system

As we have explained briefly in the introduction, the CAL system is implemented entirely in Emacs Lisp and it consists of the three components: the user interface, the parser and the checker. In this section we describe each of these three components.

4.1 User interface

The user interface provides an interactive command interpreter in a special buffer created by the CAL system. When a user starts up the CAL system, the user interface first evaluates the user's record file. The record file contains useful values such as the list of problems the user solved in the previous CAL sessions, the list of theorems the user proved so far, or the list of news already read by the user. The interface then checks if there is any unread news, and if so issues the message saying that there are some unread news. After that, the interface prompts the user for a command by issuing the prompt `CAL<1>`.

The user can now type in a CAL command. Each CAL command is one of the following forms:

⟨*command*⟩`[`⟨*arg*⟩`]`

⟨*command*⟩`[`⟨*arg1*⟩`]` `[`⟨*arg2*⟩`]`

A command in the first form takes a single argument and a command in the second form takes two arguments. Currently about 30 CAL commands are available. Most of them are commands for checking the correctness of derivations, but there are several commands that help users such as `help` or `news`. The `help` command is a command that takes a key word as its argument. For example, `help[help]` explains the `help` command itself, and `help[key]` explains the key bindings local to the CAL buffer. The `news` command is used to read news provided by us. By these commands, users can learn the usage of the CAL system.

For each game we have a command that checks whether a given derivation is a correct derivation in the given game. For example, `PropProof[`$\mathcal{D}$`]` checks if $\mathcal{D}$ is a correct derivation in the derivation game `PropProof`. When inputting $\mathcal{D}$, the user interface provides a number of supports. The local key bindings are set up so that special symbols: $\vdash$ (proves), $\supset$ (implication), $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\forall$ (all) and $\exists$ (exists) can be input by typing `/p`, `/i`, `/c`, `/d`, `/n`, `/a` and `/e`, respectively. Also, since a derivation has a tree like shape, the interface automatically inserts appropriate number of blank characters so that the text will be indented properly. The interface then sends the derivation $\mathcal{D}$ to the parser. The parser parses the character string $\mathcal{D}$ and translates it into an expression, say, E. The parser will report a syntax error if $\mathcal{D}$ cannot be parsed successfully. If the parser returns an expression E, the user interface then sends E to the checker and asks to check if E is a correct derivation of the derivation game `PropProof` or not. The checker returns an error if E is not a correct derivation and just returns `nil` (a Lisp constant) if E is a correct derivation. The user interface prints out the result of the proof checking into the CAL buffer.

The most important pair of CAL commands is `Problem` and `Answer`. There are about 220 problems CAL provides and the students are expected to solve most of them. The usage of these command is best explained by listing part of a sample CAL session[7]. In the following listing, underlined texts are inserted by the system and other texts are inserted by the user.

```
CAL <1> Problem[41]
Construct a derivation of the following hypothetical judgment
in the derivation game PropProof:
A:Prop,B:Prop,x:¬A,y:¬B ⊢ ¬(A ∨ B)
CAL <2> Answer[41][
A:Prop, B:Prop, x:¬ A, y:¬B ⊢ ¬(A∨B) in PropProof since
¬(A∨B) by imp_intro {
  (z:A∨B)[
    ⊥ by or_elim {
      z;
```

[7] English sentences in the example below are actually Japanese.

```
    (zl:A)[⊥ by imp_elim {x; zl}];
    (zr:B)[⊥ by imp_elim {y; zr}]
  }]}]
Correct!
```

To solve problems about Heyting arithmetic, the `Definition` command and the `Theorem` command are indispensable. For example:

```
CAL <3> Definition[LessThanOrEq][
x:Nat, y:Nat ⊢ ∃(z)[x + z=y]:Prop in ArithPropFrmn since
∃(z)[x + z=y]:Prop by exist_frmn {
  (z:Nat)[x + z=y:Prop by eq_frmn {}]
}]
LessThanOrEq defined as a 2-ary predicate.
```

defines `LessThanOrEq` as a binary predicate. Note that the user must supply a proof that `∃(z)[x+z=y]` is indeed a proposition of Heyting arithmetic under the hypotheses `x:Nat` and `y:Nat`. The `Theorem` command can be used to prove a new theorem, give it a name, and use it later as shown below. Note also the usage of the defined predicate `LessThanOrEq`.

```
CAL <4> Theorem[LessThanOrEqIsReflexive][
x:Nat ⊢ LessThanOrEq(x, x) in ArithProof since
LessThanOrEq(x, x) by def_intro {
  ∃(z)[x + z=x] by exist_intro {
    x + 0=x by plus_0 {}
  }}]
You have just proved a new theorem: LessThanOrEqIsReflexive.
CAL <5> ArithProof[
⊢ LessThanOrEq(0, 0) in ArithProof since
LessThanOrEq(0, 0) by LessThanOrEqIsReflexive {}
]
Yes, that is certainly a correct derivation in ArithProof!
```

Definitions made so far and theorems proved so far can be listed by the commands `list-definition` and `list-theorem`.

The user can terminate the CAL session by typing in the command `bye[]`. The user interface, then updates the record file and also writes out the session log in a file. The log file provides us with useful information as to the points where typical users make mistakes.

4.2 Parser

When the CAL parser receives a character string S from the user interface, it first decomposes it into a list of tokens W, and then it tries to construct a CAL expression E such that the judgment W `denotes` E is derivable in the definition game `Abbrev`. Thus the CAL parser may be regarded as an automatic theorem prover for `Abbrev`-judgments. The definition game `Abbrev` is a union of about 20 sub-definition games and each sub-definition game defines a syntactic class. (Each sub-definition game usually consists of several sub-sub-definition games, and they also define corresponding syntactic classes.) Therefore, whenever the user interface calls the parser, the UI sends not only S but also the name of the syntactic class to which the result of parsing S is expected to belong. For example,

```
(ParseString "s(s(0))" 'NatNum-exp)
```

asks the parser to parse the string `s(s(0))` into an expression in the syntactic class `NatNum-exp`, where the class is defined by the context free grammar:

```
NatNum-exp ::= "0" | "s" "(" NatNum-exp ")"
```

This grammar is represented in a slightly modified form in CAL as the following macro definition.

```
(defclass NatNum-exp
  (or
   "0"
   (prefix "s" (paren "(" NatNum-exp))
   (throw "Not a valid NatNum-exp!") ))
```

The parser tries to match the list of tokens W against the pattern which is the body of the definition. A pattern of the form `(prefix "s"` P`)` matches W if the first element of W is `s` and the tail (`cdr` in Lisp) of W matches the pattern P. The pattern `(paren "(" NatNum-exp)` matches W if the first and the last elements of W is a pair of parentheses '(' and ')' and the list of the remaining tokens matches `NatNum-exp`. There is a special file that defines all the CAL syntactic classes, and this file is loaded by the UI when the user starts up the CAL system. So the parser can get the necessary syntactic definitions from the Lisp memory. In the present case, the parser returns the CAL expression `<s, <s, 0>>` as the result of parsing W into an expression in `NatNum-exp`.

4.3 Checker

The CAL checker is a simple proof-checker; it receives a derivation and checks if it is a correct derivation in the specified game. If it is correct, it just returns a non-significant value. Otherwise, the checker analyzes the sources of incorrectness and returns an error message indicating a source of the incorrectness. Since the CAL system is used by students, it is important to give helpful error messages, and we have tried to give effective suggestions to students. Yet, much more are still left to be done. In the current implementation, the checker traverses each derivation tree in the bottom-up (from leaves to the root), left-to-right direction.

5 Conclusion

We started our lecture course, Logic and Computation, in 1998 and we have been using the CAL system since then. Students attended the course liked the CAL system very much and used CAL intensively. They also gave us useful suggestions which greatly contributed to improve the system.[8] By utilizing such feedbacks from students we could gradually polish our system into the current form. Although the system is not perfect, our system has proven to be a good system to complement our lecture course. Through the teaching and implementing experiences we now strongly feel that education of formal logics should always be supported by computer environments like the CAL system.

There are a number of proposals which aim to give frameworks for presenting logical systems, and they are called *logical frameworks* [5]. CAL is one such framework, and we compare below our system with other frameworks.

Among such frameworks, Martin-Löf's theory of expressions [6] is based on the concept of *arity* and our system is strongly influenced by his theory. The theory is used as an underlying system, and an object logic (Intuitionistic Type Theory in this case) is presented on top of it, but it is general enough to be able to implement other logical systems on top of it. The theory has a built-in abstraction mechanism and instantiation mechanism, and the definitional equality relation on the expressions is defined by a set of reduction rules on expressions. The reduction relation is Church-Rosser and strongly normalizing and hence the equality is decidable. In Martin-Löf's theory of expressions, α-equivalent expressions are definitionally equal.

Our theory of expressions has stronger notion of definitional equality in the sense that we can distinguish two α-equivalent and indistinguishable expressions in Martin-Löf's theory. However, it is easy to *add* a weaker notion of equality on our expressions by identifying α-equivalent expressions, and under this weak equality, our theory and Martin-Löf's theory become isomorphic.

The reason why our theory is much simpler than Martin-Löf's and yet more expressive than his theory is that we defined our expressions only by using constructors (introduction rules) while Martin-Löf's theory has both constructors and destructors for abstracts and lists. In case of abstracts, Martin-Löf's theory has both an abstraction rule which constructs (introduces) abstracts and an application rule which destructs (eliminates) abstracts. In our theory, we have only an introduction rule for abstracts and elimination of abstracts is defined not within the theory but as a meta-level operation of instantiation. Our philosophy is that expressions must be designed as a data structure and we showed that it is possible to do so naturally by using constructors only as, for example, Lisp symbolic expressions are constructed from atoms only by means of the cons operation.

The reason why we could develop our theory as a type-free theory is again that our theory has only introduction rules. Martin-Löf's theory uses the notion of arity to avoid self application which might make the theory not strongly

[8] We would like to acknowledge them sincerely.

normalizing. Our theory is strongly normalizing and Church-Rosser in the very trivial sense that we have no notion of reduction in our theory. Martin-Löf's theory requires a reduction relation to define the notion of definitional equality, but we can define definitional equality in our theory without the notion of reduction as we saw in section 2, and this was possible since our theory has only introduction rules. The notion of reduction, that is instantiation, is introduced as a meta-level concept only after we defined the theory.

Edinburgh Logical Framework [4] is a powerful framework for representing various logics in the type-theoretic setting, and is considered as a generalization of Martin-Löf's theory. But precisely because it is a generalization based on type theory, it cannot serve as a *foundational* framework for building logical theories on top of it. A foundational framework on top of which logical systems and type theories can be built must itself be built *without* using type theories. Our theory of expressions is such a foundational framework.

References

1. Barendregt, H. P., *The Lambda Calculus, Its Syntax and Semantics*, North-Holland, 1981.
2. Barwise, J. and J. Etchemendy, *Tarski's World*, CSLI Lecture Notes, No. 25, CSLI Publications, Cambridge University Press, 1994.
3. 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.
4. Harper, R., F. Honsell, and G. Plotkin, A Framework for Defining Logics, *Journal of the Association for Computing Machinery*, Vol. 40, No. 1, pp. 143–184, 1993.
5. Huet, G., and G. Plotkin eds., *Logical Frameworks*, Cambridge University Press, 1991.
6. Nordström, B., K. Petersson, and J. M. Smith, *Programming in Martin-Löf's Type Theory*, Oxford University Press, 200 pages, 1990.
7. Sato, M., and M. Hagiya, Hyperlisp, in de Bakker, van Vliet eds., *Algorithmic Languages*, North-Holland, pp. 251–269, 1981.
8. Sato, M., Theory of Symbolic Expressions, II, *Publ. of Res. Inst. for Math. Sci.*, Kyoto Univ., **21**, pp. 455–540, 1985.
9. 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.
10. Sato, M., T. Sakurai and R. Burstall, Explicit Environments, *Fundamenta Informaticae* 45, pp. 79–115, 2001.
11. Sato, M., T. Sakurai and Y. Kameyama, A Simply Typed Context Calculus with First-Class Environments, Proc. Fifth International Symposium on Functional and Logic Programming (FLOPS), *Lecture Notes in Computer Science* 2024, pp. 359–374, 2001.