A Primer on Proofs and Types

Masako Takahashi

Department of Mathematical
and Computing Sciences
Tokyo Institute of Technology
masako@is.titech.ac.jp

Abstract

This article is intended to be an introduction (for real novices) to the area of proof systems and type systems. We provide very basic notions and their fundamental properties together with some informal ideas behind them.

We start by describing how 'logical reasoning' can be formulated into a proof system, and how structures of proofs can be studied based on this system. We then proceed to a concise exposition of the notion of computable functions, which emerged from proof theory in the early 1930's (during the development of the incompleteness theorems by Gödel). Next, after a short account of pure (type-free) λ-calculus in connection with computable functions, we discuss three typed λ-calculi (the simple type system, the second-order type system, and the calculus of constructions) as representatives of type systems, with emphasis on their relation to the corresponding intuitionistic proof systems (the implicational fragment, the second-order propositional logic, and the higher-order predicate logic, respectively).

1 Introduction

In the long history of mathematics, one might say mathematical logic started very late. Of course, certain patterns of logical inference, in particular those called syllogisms, were studied by Aristotle, but they are very restrictive and too weak to be considered as a foundation of mathematics,[1] and unfortunately the tradition of syllogistic logic lasted for more than two thousand years. It was in the middle of the nineteenth century that significant contributions in propositional logic were made by Boole, de Morgan, etc.

The propositional logic is a framework for investigating logical reasoning by taking propositions as basic units. Here by propositions we mean sentences which are either true or false. The main objective of propositional logic is then to study

[1] For instance, nested quantifiers are not considered in syllogisms, hence one cannot state Euclid's theorem; for any integer n there exists a prime number which is greater than n.

the role of propositional connectives such as *and*, *or*, *not*, *implies*, etc., from which the notion of Boolean algebras (i.e., complemented distributive lattices) naturally emerges.

Thus in propositional logic one is concerned exclusively with the relation between sentences, but of course this is not enough. The framework is naturally extended to that of predicate logic, in which one can consider predicates, i.e., sentences possibly with parameters, and quantifiers (*for all*, and *there exists*). Although a part of predicate logic had been discussed in ancient Greek times, it was C.S.Peirce and G.Frege in the late nineteenth century who started its systematic study as a foundation for mathematics.

When we look at mathematical developments in the late nineteenth century from the logical point of view, we note that Cantor developed transfinite set theory, and Dedekind studied fundamental properties of natural numbers, which were later formulated as Peano axioms. Stimuli from geometry, such as the axiomatization of projective geometry, are also worth mentioning.

Then in 1901 B.Russell discovered his famous paradox[2] in naive set theory (cf. [37]). This discovery, together with a number of other paradoxes discovered around the turn of the century, was a strong attack on the mathematical study of logic which was still in its infancy, but at the same time it was these paradoxes that gave incentives to subsequent attempts to search for rigorous logical systems on which mathematics could safely be based.

In the early twentieth century, the major objectives of studying logical foundations of mathematics may be classified as follows.

- To formulate and investigate purely logical reasoning, which is somewhat innate in human thought as natural languages.
- To find appropriate axiomatic systems for numbers, sets, and other basic mathematical entities, and prove their consistency and completeness.

As to the first objective, it was proved by Gödel that the goal is accomplished by the system which is essentially due to Frege and later refined by Whitehead-Russell. In section 2, we will explain the result together with its basic notions and the informal ideas behind them.

The system in Principia Mathematica by Whitehead-Russell was primarily intended to formalize a mathematical foundation, and so was the system by Hilbert-Ackermann. In addition to those systems, Zermelo's set theory and many others aimed at the second objective, and they were also successful in the sense that these systems had encouraged and given enormous influence on the later development of mathematical logic. However it was proved also by Gödel in two incompleteness theorems that the second objective cannot be fulfilled.

Gödel's first incompleteness theorem proved the following: Any consistent formal axiom system, say T, which includes elementary arithmetic is incomplete in the sense that there exists a sentence A in the language of T such that neither A nor

[2] Let Y be the set of all sets X such that X is not a member of itself; that is, $Y = \{X | X \notin X\}$. Then we have $Y \in Y \Longleftrightarrow Y \notin Y$, a contradiction.

its negation can be proved in T. This implies that there is no formal axiom system in which all and only the true sentences of elementary arithmetic are provable.

Gödel went on further and proved the second incompleteness theorem, which says: for any consistent formal system which includes elementary arithmetic, its consistency cannot be proved in the system itself. (See [14] for Gödel's works, and [13] and [51] for their historical background.)

By a formal system, we mean a system to generate theorems and their proofs by using a specified set of formulas as axioms and a specified set of rules of inference such that *one should be able to check by a finite procedure whether a given formula is one of the axioms or not, and likewise for the rules of inference.* The reason to require the existence of finite procedures is that otherwise a formidable system like the one containing all sentences which are true (under the standard interpretation) as axioms might be considered. But such a system would contribute nothing to our understanding of natural numbers and other mathematical entities.

General understanding of the notion of finite procedures mentioned in Gödel's paper was at that time still in embryo. But in the subsequent short period a number of proposals were made by Kleene [25], Church [7], Turing [49], etc. to formalize the notion, and they turn out to be equivalent each other in spite of totally different appearances. In section 3, we give a concise account of the notion both from a mathematical viewpoint (based on recursive functions) and from a computer-scientific view (based on simple languages for describing procedures). In section 4, we discuss another framework, called (type-free) λ-calculus, to express finite procedures or computation, and the discussion is extended in section 5 to a number of type structures built over the λ-calculus. We can observe that some type systems show a great affinity with (intuitionistic) logical systems discussed in section 2 and their extensions. We will focus our attention on three typical examples of such systems; the ones with simple types, with second-order types, and with higher-order dependent types.

The paper is intended to be a first guide to theories of proofs and types, and anyone who has moderate mathematical maturity should have no difficulty in understanding the contents.

2 Formal proof systems

Logical connectives commonly used in mathematics are; *and* ($\wedge$), *or* ($\vee$), *not* ($\neg$), *implies* ($\rightarrow$), *equivalence* ($\leftrightarrow$), *for all* ($\forall$), and *exists* ($\exists$). In the mathematical context, their meanings are fixed,[3] and by combining them with expressions specific to the field under consideration one can compose propositions. A proof in mathematics then can be considered as a process to derive a proposition by a series of valid inferences from some postulates such as axioms of the field, known results, assumptions, etc., which are also expressed as propositions.

[3] The standard (classical) meanings of propositional connectives $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$ are determined by the truth tables, and those of quantifiers $\forall, \exists$ are determined as the wordings 'for all' and 'there exists' suggest.

Here, the validity of inferences of course means logical validity. Typical examples of logically valid inferences are the argument by contraposition; to infer $A \to B$ from $\neg B \to \neg A$, and the argument by contradiction; to infer A from the fact that a contradiction follows from $\neg A$. We will denote a contradiction by $\bot$.

There exist a large variety of logically valid inferences, ranging from trivial ones to those which are too complicated to be appreciated. We will be concerned here with their totality from mathematical point of view. A device which will be useful for the purpose is a system to generate precisely all logically valid inferences. Indeed, a number of such systems have been discovered. The basic idea common to all of them is to decompose complicated patterns of inference to simpler ones. In effect, the systems specify (i) the simplest patterns of inference, and (ii) possible ways to construct complicated patterns out of simpler ones. We will call such a system a *proof system*. By adding appropriate axioms to such a proof system, one can obtain an axiom system suitable for a specific field under consideration.

2.1 Logical formulas

Before presenting the proof systems, let us first recall how propositions are constructed. They are composed of mathematical formulas (such as $(x \cdot y) \cdot z = x \cdot (y \cdot z)$, $x \cap y \subseteq z$, etc) combined with logical symbols. A mathematical formula used in this context consists of a predicate (in the above examples, $=$ and $\subseteq$) and some mathematical terms as its parameters ($x, y, x \cdot y, (x \cdot y) \cdot z, x \cap y$, etc. in the above examples), which are composed of variables and function symbols.

We will define *logical formulas* by extracting the syntactic aspect of propositions. The reason for doing so is that patterns of logical inference are syntactic by nature, and unless we distinguish the syntactic part from the semantic part, it is very difficult to focus our attention on purely logical relations among propositions. Indeed, in the long history of mathematics, the writing of propositions with logical symbols started only in the nineteenth century, and before then logical connectives had been expressed in natural languages. But usage of logical connectives such as 'or' and 'implies' in natural languages is context-dependent,[4] and one can imagine how difficult it is to study logical inferences in such situations. Thus one of the first important steps towards mathematical study of logic was to formulate propositions with syntax/semantics dichotomy.

2.1.1 Definition First, we define *terms* recursively,[5] as follows.

(t1) Variables $x_0, x_1, x_2, \ldots$ are terms.

(t2) If f is an n-ary function symbol where $n \geq 0$ and $t_1, t_2, \cdots, t_n$ are terms, then $f(t_1, t_2, \cdots, t_n)$ is a term. (0-ary function symbols may be called constant symbols.)

[4] For example, 'or' in English often means 'exclusive or' but not always.

[5] In a recursive definition of a notion, as in the case of proof systems one specifies the list of simplest items of the notion, and the list of possible ways of constructing complicated items out of simpler ones. In recursive definitions, we always assume that the specified lists are exhaustive.

Next, we define *logical formulas* recursively, as follows.

(f1) If p is an n-ary predicate symbol where $n \geq 0$ and $t_1, t_2, \cdots, t_n$ are terms, then $p(t_1, t_2, \cdots, t_n)$ is a logical formula.

(f2) $\bot$ is a logical formula.

(f3) If A, B are logical formulas and x is a variable, then $(A \wedge B)$, $(A \vee B)$, $(\neg A)$, $(A \to B)$, $(\forall x.A)$ and $(\exists x.A)$ are logical formulas.[6]

Logical formulas may be called simply *formulas*. The formulas of the form (f1) are called *atomic formulas*. When a logical formula has a subformula of the form $\forall x.A$ or $\exists x.A$, then occurrences of x in the subformula are said to be *bound*. Occurrences of variables which are not bound are said to be *free*.

As we claimed before, what we defined above is mere syntax of propositions. Then what we should do next in order to obtain actual propositions is to assign appropriate meanings to logical formulas.

For the purpose, we first specify what values the variables in (t1) can take (e.g., natural numbers, real numbers, etc), and then specify what functions are meant by function symbols f in (t2), and what predicates[7] are meant by predicate symbols p in (f1). Once we provide these items of information and actual values of variables, we can evaluate the 'value' of each atomic formula, which is either true or false. As for the logical formula $\bot$ in (f2), its 'value' is defined to be always false, as mentioned before. Finally, the 'values' of compound formulas in (f3) are determined as usual based on the standard (classical) meaning of logical connectives. The above mentioned information for specifying the semantic part of propositions is, as a whole, called an *interpretation* of logical formulas. In case where an interpretation is well-understood or immaterial to evaluate a certain formula, we may simply say a formula is true or false.

Note that all variables in our logical formulas range over a single domain; thus, in particular, we have no function/predicate variables. It is possible to relax the restriction, but for simplicity we stay in this section with this conventional definition, which is called the *first-order formulas*, and later in 5.3 we will consider more general framework.

2.2 The classical system NK

Among a number of proof systems known today, we first present the *classical calculus of natural deduction* NK introduced by Gentzen [15]. One of the advantages of the system is that it reflects, to a certain extent, the kind of inferences that human beings practise in mathematical reasoning and also in everyday life. Let us first take a close look at simple examples and see how we proceed with our

[6] When there is no ambiguity we may omit parentheses in logical formulas and the dots after $\forall x$ and $\exists x$. Also for simplicity we omit $A \leftrightarrow B$ from the definition of logical formulas and take it as an abbreviation of $(A \to B) \wedge (B \to A)$.

[7] A function that takes values in $B = \{\text{true}, \text{false}\}$ is called a *predicate*.

reasoning.

2.2.1 Examples

1. $((A \wedge B) \vee (A \wedge C)) \to (A \wedge (B \vee C))$. In order to verify this formula, suppose either $A \wedge B$ or $A \wedge C$ holds. In the first case, from $A \wedge B$ follow both A and B, and from B follows $B \vee C$. Thus in this case we have $A \wedge (B \vee C)$. Similarly, in the second case, we get $A \wedge (B \vee C)$. As the same conclusion $A \wedge (B \vee C)$ is drawn from both cases, it also follows from $(A \wedge B) \vee (A \wedge C)$. Thus we get $(A \wedge B) \vee (A \wedge C) \to A \wedge (B \vee C)$.
2. $\neg(\exists x.A(x)) \to \forall x.(\neg A(x))$. Assume $\neg(\exists x.A(x))$; that is, there exists no x such that $A(x)$ holds. In order to verify $\forall x.(\neg A(x))$, take any y. Then $\neg A(y)$ holds, because otherwise there is y such that $A(y)$ holds, which contradicts our assumption. Since y is arbitrary, we conclude $\forall x.(\neg A(x))$. Thus we get $\neg(\exists x.A(x)) \to \forall x.(\neg A(x))$.

One may observe that the above arguments can be decomposed into a number of small steps, and some of the steps follow the same pattern; for example, the last steps of the two examples do. It is indeed shown (cf. theorem 2.2.4 below) that there exist a finite number of patterns of inference, into which any purely logical reasoning can be decomposed. The following table illustrates the list of all rules of the system NK; these rules except the first two are divided into six 'introduction' rules ($\wedge$I), ($\vee$I), ($\neg$I), ($\to$I), ($\forall$I) and ($\exists$I), and six 'elimination' rules ($\wedge$E), ($\vee$E), ($\neg$E), ($\to$E), ($\forall$E) and ($\exists$E).

2.2.2 Rules of NK

$$A \qquad \dfrac{\begin{array}{c}[\neg A]\\ \vdots \\ \bot\end{array}}{A}\ (\bot)$$

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A_1 & A_2\end{array}}{A_1 \wedge A_2}\ (\wedge\mathrm{I}) \qquad \dfrac{\begin{array}{c}\vdots \\ A_1 \wedge A_2\end{array}}{A_i}\ (\wedge\mathrm{E}) \qquad \dfrac{\begin{array}{c}\vdots \\ A_i\end{array}}{A_1 \vee A_2}\ (\vee\mathrm{I}) \qquad \dfrac{\begin{array}{ccc}\vdots & [A_1] & [A_2] \\ A_1 \vee A_2 & \vdots & \vdots \\ & B & B\end{array}}{B}\ (\vee\mathrm{E})$$

$$\dfrac{\begin{array}{c}[A]\\ \vdots \\ \bot\end{array}}{\neg A}\ (\neg\mathrm{I}) \qquad \dfrac{\begin{array}{cc}\vdots & \vdots \\ A & \neg A\end{array}}{\bot}\ (\neg\mathrm{E}) \qquad \dfrac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B}\ (\to\mathrm{I}) \qquad \dfrac{\begin{array}{cc}\vdots & \vdots \\ A \to B & A\end{array}}{B}\ (\to\mathrm{E})$$

$$\dfrac{\begin{array}{c}\vdots \\ C(y)\end{array}}{\forall x.C(x)}\ (\forall\mathrm{I}) \qquad \dfrac{\begin{array}{c}\vdots \\ \forall x.C(x)\end{array}}{C(t)}\ (\forall\mathrm{E}) \qquad \dfrac{\begin{array}{c}\vdots \\ C(t)\end{array}}{\exists x.C(x)}\ (\exists\mathrm{I}) \qquad \dfrac{\begin{array}{cc}\vdots & [C(y)] \\ \exists x.C(x) & \vdots \\ & B\end{array}}{B}\ (\exists\mathrm{E})$$

In these rules, A, A_1, A_2, B and $C(x)$ range over arbitrary logical formulas, and i over $\{1, 2\}$. All the rules except the first one show how smaller derivations can be

combined to make larger ones, and these are called *inference rules.* In each of them, derivations already obtained are shown schematically over the horizontal line and the consequence of the current step is shown below the line. The first rule, which is the only rule without horizontal line, shows the simplest derivation; to derive a formula A from A itself. It is used as the base step of construction of derivations and we will call it the (start) rule.

The conjunction introduction rule ($\wedge$I) represents the following pattern of argument; if we have two derivations, for A_1 and for A_2 respectively, then we can combine them to make a derivation for $A_1 \wedge A_2$. Similarly for ($\wedge$E) and ($\vee$I). Next, the disjunction elimination rule ($\vee$E) shows the pattern of argument by cases:

> Suppose we have derivations for
>
> (0) $A_1 \vee A_2$,
>
> (1) B from assumptions possibly including A_1, and
>
> (2) B from assumptions possibly including A_2.
>
> Then we can conclude B.

Note that in this argument, the assumption A_1 is local to the derivation (1); it is no longer necessary as an assumption in the whole argument. Likewise, A_2 is local to the derivation (2). Note also that these are the only local assumptions, and all other assumptions in (0) $\sim$ (2), if any, remain to be the assumptions of the whole argument. To indicate the locality of A_1 and A_2, they are enclosed by square brackets in the table 2.2.2, and said to be *discharged* in this step. Throughout the table we follow the convention of indicating discharged assumptions by square brackets. Note that the number of occurrences of the discharged assumptions is not necessarily one, but could be any finite number (including 0).

The next rule ($\neg$I), called the negation introduction rule, claims that if a contradiction $\perp$ follows from A then $\neg A$ is derivable (under the same assumptions as before except A being discharged).

In the rules ($\forall$E) and ($\exists$I), we write $C(t)$ for the formula obtained from $C(x)$ by substituting[8] an arbitrary term t for each free occurrence of the variable x. Then the rule ($\forall$E) says that $C(t)$ follows from $\forall x.C(x)$, while the rule ($\exists$I) says $\exists x.C(x)$ follows from $C(t)$.

In ($\forall$I) and ($\exists$E), $C(y)$ stands for the result of substitution of a variable y for free occurrences of x in $C(x)$, but here y must be a variable not free in $\forall x.C(x)$ and subject to additional conditions specified below for the two rules separately. The rule ($\forall$I) says that

> if $C(y)$ follows from assumptions *in which y has no free occurrence,* then $\forall x.C(x)$ follows from the same assumptions.

The rule ($\exists$E) says

> if $\exists x.C(x)$ follows from assumptions, say Γ, and B follows from some assumptions, say Γ', such that y *has free occurrences neither in B nor in Γ' except in $C(y)$*, then we can derive B from $\Gamma \cup (\Gamma' - \{C(y)\})$.

[8] By substitution, we always mean replacement of variable occurrences with some expression under the condition that *the replacement does not create any new bound occurrence of variables*, in the same sense as we cannot substitute z for x in $\{z \in Z \mid x < z\} \neq \emptyset$.

Similar accounts for the rest of rules should be clear. The rule ($\bot$), called the classical absurdity rule, describes the pattern of argument by contradiction. The rule ($\to$E) is also known by the name of *Modus Ponens*.

We can apply these patterns consecutively to obtain a logically valid derivations, and the process is illustrated in the form of trees, which are called *proof trees*.

2.2.3 Examples The arguments in Example 2.2.1 can be illustrated in the following proof trees, in which labels $u, v, \ldots$ attached to discharged assumptions indicate the steps at which the assumptions are discharged.

(1) A proof tree for $(A \wedge B) \vee (A \wedge C) \to A \wedge (B \vee C)$.

$$\dfrac{[(A \wedge B) \vee (A \wedge C)]^v \qquad \dfrac{\dfrac{[A \wedge B]^u}{A}\,(\wedge\mathrm{E}) \quad \dfrac{\dfrac{[A \wedge B]^u}{B}\,(\wedge\mathrm{E})}{B \vee C}\,(\vee\mathrm{I})}{A \wedge (B \vee C)}\,(\wedge\mathrm{I}) \qquad \dfrac{\dfrac{[A \wedge C]^u}{A}\,(\wedge\mathrm{E}) \quad \dfrac{\dfrac{[A \wedge C]^u}{C}\,(\wedge\mathrm{E})}{B \vee C}\,(\vee\mathrm{I})}{A \wedge (B \vee C)}\,(\wedge\mathrm{I})}{A \wedge (B \vee C)}\,(\vee\mathrm{E})^u}{(A \wedge B) \vee (A \wedge C) \to A \wedge (B \vee C)}\,(\to\mathrm{I})^v$$

(2) A proof tree for $\neg(\exists x.A(x)) \to \forall x.(\neg A(x))$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[A(y)]^u}{\exists x.A(x)}\,(\exists\mathrm{I}) \qquad [\neg(\exists x.A(x))]^v}{\bot}\,(\neg\mathrm{E})}{\neg A(y)}\,(\neg\mathrm{I})^u}{\forall x.(\neg A(x))}\,(\forall\mathrm{I})}{\neg(\exists x.A(x)) \to \forall x.(\neg A(x))}\,(\to\mathrm{I})^v$$

The first proof tree shows that $(A \wedge B) \vee (A \wedge C) \to A \wedge (B \vee C)$ is derivable without assumption, because in the proof tree all formulas at the leaf nodes are discharged. Likewise the second proof tree shows that $\neg(\exists x.A(x)) \to \forall x.(\neg A(x))$ is derivable without assumption.

Now let us define two notions necessary to state Gödel's completeness theorem. We suppose A is a formula and Γ is a set of formulas. First, by writing

$$\Gamma \vdash_{\mathrm{NK}} A$$

we mean that A is derivable in NK from assumptions in Γ. More precisely, the notation means that there is a proof tree constructed by rules in 2.2.3 such that A is at the root node and all undischarged formulas at the leaf nodes belong to Γ. Note that it is a purely syntactic notion; that is, whether it holds or not depends, by definition, only on the forms of formula. The other notion is a semantic entailment relation

$$\Gamma \models A,$$

which means that to infer A from Γ is a valid inference. More precisely, it means that for any interpretation I, A is true under I if all members of Γ are true under I. Gödel's completeness theorem says that the two notions above coincide. In other words, all and only valid inferences can be attained by applying rules of NK.

2.2.4 Completeness Theorem for NK For a formula A and a set Γ of formulas,

$$\Gamma \vdash_{\mathrm{NK}} A \iff \Gamma \models A.$$

The $\Rightarrow$ direction of the theorem, which is sometimes called the soundness theorem of NK, can be verified by induction on the number of steps in the derivation of A from Γ. The more difficult part of the theorem is the $\Leftarrow$ direction, and its proof can be sketched as follows. First note that the proof can be reduced to the special case where A is $\bot$, because $\Gamma \vdash_{\mathrm{NK}} A$ is equivalent to $\Gamma \cup \{\neg A\} \vdash_{\mathrm{NK}} \bot$ by the rules ($\neg$E) and ($\bot$), and $\Gamma \models A$ is equivalent to $\Gamma \cup \{\neg A\} \models \bot$ by definition of $\models$. In the syntactic side, when $\Gamma \vdash_{\mathrm{NK}} \bot$ holds we say Γ is *inconsistent*, and otherwise Γ is *consistent*. In the semantic side, $\Gamma \models \bot$ means that there is no interpretation under which all members of Γ are true (since $\bot$ is false under any interpretation). In this case, we say Γ is *unsatisfiable*, and otherwise Γ is *satisfiable*. Now, in order to prove $\Gamma \models \bot \implies \Gamma \vdash_{\mathrm{NK}} \bot$ by contraposition, let us assume that Γ is consistent. Then it can be shown that there exists a consistent extension Γ' of Γ (i.e., $\Gamma \subseteq \Gamma'$ and $\Gamma' \nvdash_{\mathrm{NK}} \bot$) from which one can directly construct an interpretation such that all members of Γ' are true. Thus Γ is shown to be satisfiable; i.e., $\Gamma \not\models \bot$. For more details, see for example [50].

Now that the purely logical part of mathematics is formulated by NK, we can construct an axiom system for a certain field by specifying a set of appropriate formulas as axioms of the field. For example, an axiom system for group theory may be obtained by specifying

(G1) $\forall x \forall y \forall z [(x \cdot y) \cdot z = x \cdot (y \cdot z)]$.
(G2) $\forall x [e \cdot x = x]$.
(G3) $\forall x \exists y [y \cdot x = e]$.
(E1) $\forall x [x = x]$.
(E2) $\forall x \forall y [x = y \rightarrow (A(x) \rightarrow A(y))]$.

as axioms[9] where $=$ is a binary predicate symbol, e is a 0-ary function (or constant) symbol and $\cdot$ is a binary function symbol. Let Γ be the set of all these axioms. Then we know from the completeness theorem that $\Gamma \vdash_{\mathrm{NK}} A$ holds if and only if A is true for all groups.

The best-known axiom system for natural numbers is the Peano arithmetic (PA, for short). Here, by *natural numbers* we mean the nonnegative intergers $0, 1, 2, \ldots$

[9] (E1) and (E2) are general axioms for congruence relations; they guarantee, with the help of the rules of NK, the reflexivity, symmetry, transitivity, and congruency of $=$. In (E2), $A(x)$ and $A(y)$ are respectively the results of substitution of new variables x and y for free occurrences of z in an arbitrary formula $A(z)$. Thus (E2) is actually not one axiom, but an axiom scheme.

(including 0). In the axioms of PA below, the intended meanings of the symbols 0, s, $+$, $\times$, $=$ are the natural number 0, the successor function $s(x) = x + 1$, addition, multiplication, and the equality of natural numbers, respectively. The first six axioms specify basic properties of these functions. (PA7) states the induction principle, in which $A(x)$ is an arbitrary formula, and $A(t)$ for a term t (e.g., $0, \mathrm{s}(x)$) stands for the result of substitution of t for free occurrences of the variable x in $A(x)$. The equality axioms (E1) and (E2) are the same as before.

(PA1) $\forall x\,[\neg(\,\mathrm{s}(x) = 0\,)]$.
(PA2) $\forall x\,\forall y\,[\mathrm{s}(x) = \mathrm{s}(y) \;\rightarrow\; x = y\,]$.
(PA3) $\forall x\,[\,x + 0 = x\,]$.
(PA4) $\forall x\,\forall y\,[\,x + \mathrm{s}(y) = \mathrm{s}(x + y)\,]$.
(PA5) $\forall x\,[\,x \cdot 0 = 0\,]$.
(PA6) $\forall x\,\forall y\,[\,x \cdot \mathrm{s}(y) = x \cdot y + x\,]$.
(PA7) $A(0) \;\rightarrow\; [\forall x\,(\,A(x) \;\rightarrow\; A(\mathrm{s}(x))\,) \;\rightarrow\; \forall x\,A(x)]$.
(E1) $\forall x\,[\,x = x]$.
(E2) $\forall x\,\forall y\,[\,x = y \;\rightarrow\; (\,A(x) \;\rightarrow\; A(y)\,)\,]$.

It is for the system PA and arbitrary extensions of it that Gödel proved his incompleteness theorems mentioned in section 1. In general, if an axiom system T is obtained from NK by adding a set $\mathrm{Ax}(T)$ of axioms, we will write $\Gamma \vdash_T A$ for $\Gamma \cup \mathrm{Ax}(T) \vdash_{\mathrm{NK}} A$. Thus for example we have $\vdash_{\mathrm{PA}} \forall x \forall y\,[x + y = y + x]$, $\vdash_{\mathrm{PA}} \forall x\,[\neg(x = 0) \rightarrow \exists y\,(x = \mathrm{s}(y))]$, $\vdash_{\mathrm{PA}} 2 + 3 = 5$, etc. where 2 stands for $\mathrm{s}(\mathrm{s}(0))$ and similarly for other natural numbers.

2.3 The intuitionistic system NJ

A proof system weaker than NK, which is called the *intuitionistic calculus of natural deduction* NJ, is also introduced by Gentzen [15]. The system is obtained from NK by replacing the classical absurdity rule ($\perp$) with the intuitionistic absurdity rule

$$\dfrac{\begin{array}{c}\vdots\\ \perp\end{array}}{A}\;(\perp)'$$

which says that from a contradictioin any formula can follow (without changing the assumptions). The difference between the two rules is that in ($\perp$) one can discharge $\neg A$ from assumptions, but in $(\perp)'$ one cannot. A typical example to show the difference of the two systems is the law of the excluded middle $A \vee \neg A$. It is derivable[10] in NK as shown below

[10] When a formula is derivable in a system T without assumption, we simply say it is derivable in T.

$$\dfrac{\dfrac{\dfrac{[A]^u}{A \vee \neg A}\,(\vee\mathrm{I}) \qquad [\neg(A \vee \neg A)]^v}{\dfrac{\dfrac{\bot}{\neg A}\,(\neg\mathrm{I})^u}{A \vee \neg A}\,(\vee\mathrm{I})}\,(\neg\mathrm{E}) \qquad [\neg(A \vee \neg A)]^v}{\dfrac{\bot}{A \vee \neg A}\,(\bot)^v}\,(\neg\mathrm{E})$$

but in NJ, since we cannot discharge the assumption $\neg(A \vee \neg A)$ in the last step of the proof tree above, what we can get in this argument is only $\neg(A \vee \neg A) \vdash_{\mathrm{NJ}} A \vee \neg A$. Other examples which are derivable in NK but not in NJ are $\neg\neg A \to A$, $(\neg A \to \neg B) \to (B \to A)$, $((A \to B) \to A) \to A$ (Peirce's law), etc. In fact, in NJ logical connectives $\wedge, \vee, \to, \forall, \exists$ are independent of each other,[11] while in NK they are dependent; e.g., $\vdash_{\mathrm{NK}} (A \vee B) \leftrightarrow ((A \to \bot) \to B)$.

All of the above mentioned properties of NJ are proved by way of the normalization theorem for NJ (cf. [35] Ch.IV). The statement of the theorem itself is rather involved, but in effect it says that every formula derivable in NJ can be derived by a 'normal' (or 'detour-free') derivation. Here, a detour means an application of an introduction rule immediately followed by an application of the corresponding elimination rule in the following way

$$\dfrac{\dfrac{\begin{array}{c}[A]^u \\ \vdots\,\Pi_1 \\ B\end{array}}{A \to B}\,(\to\mathrm{I})^u \qquad \begin{array}{c}\vdots\,\Pi_2 \\ A\end{array}}{B}\,(\to\mathrm{E}) \qquad\qquad \dfrac{\begin{array}{c}\vdots\,\Pi_3 \\ A\end{array} \qquad \begin{array}{c}\vdots\,\Pi_4 \\ B\end{array}}{A \wedge B}\,(\wedge\mathrm{I})}{A}\,(\wedge\mathrm{E})$$

and they can be removed without changing the assumptions and the conclusions; for example, the detours above can be removed, as follows.

$$\begin{array}{c}\vdots\,\Pi_2 \\ A \\ \vdots\,\Pi_1 \\ B\end{array} \qquad\qquad \begin{array}{c}\vdots\,\Pi_3 \\ A\end{array}$$

The detour removal process may not necessarily decrease the size of proof trees, because in some cases it involves copying of subderivations.[12] Nevertheless, the detour removal process can always be terminated, resulting in a normal (or detour-free) derivation; this is what the normalization theorem for NJ claims.

Another important consequence of the normalization theorem for NJ is due to the 'subformula property' of the normal derivations; the normal derivation of $\Gamma \vdash_{\mathrm{NJ}} A$ contains only subformulas of A or of those in Γ, providing that we take $\neg A$

[11] That is, there is no $\wedge$-free formula which is equivalent in NJ to $A \wedge B$, and similarly for $\vee$, $\to$, $\forall$, and $\exists$. Note however that $\neg A$ is equivalent to $A \to \bot$ in NJ. That is, $\vdash_{\mathrm{NJ}} (\neg A) \leftrightarrow (A \to \bot)$.

[12] Recall that Π_1 in the left detour may contain more than one occurrences of the discharged assumption $[A]$, and the detour removal process replaces each occurrence of $[A]$ with Π_2.

as an abbreviation of $A \to \bot$.[13] Thus, for example, if $\Gamma \vdash_{\mathrm{NJ}} A$ and if formulas in Γ and A consist of only atomic formulas and $\to$, then we know that A is derivable from Γ by using only the rules (start), ($\to$I), ($\to$E). Likewise for other logical constants $\wedge, \vee, \to, \bot, \forall, \exists$, and for any combination of them. These facts can be summarized, as follows: NJ is a conservative extension[14] of its S-fragment[15] where S is any set of logical constants. As a special case, we obtain the consistency of NJ, i.e., $\nvdash_{\mathrm{NJ}} \bot$. In passing, we note that the consistency can also be proved semantically by using theorem 2.2.4. Indeed, from $\not\models \bot$ we know $\nvdash_{\mathrm{NK}} \bot$; hence $\nvdash_{\mathrm{NJ}} \bot$.

We have also a normalization theorem for NK (cf. [35] Ch.III), but because of the presence of the classical absurdity rule ($\bot$), the notion of normal derivation of NK is not so well-behaved as that of NJ, and as a consequence the theorem is less powerful.

We will come back to the (strong) normalization theorems for intuitionistic proof systems in section 5 in relation with the (strong) normalization theorems for type systems, because the latter can be stated and proved more easily than the former and the latter imply the former.

The system NJ is called 'intuitionistic' because it is formulated based on the intuitionistic (or constructive) view of logical formulas, in which a formula is considered to be true only if it has a constructive proof of the following nature.

- A proof of $A \wedge B$ is given by presenting a proof of A and a proof of B.
- A proof of $A \vee B$ is given by presenting either a proof of A or a proof of B together with information to tell which of A, B is the case.
- A proof of $A \to B$ is a method of transforming any proof of A to a proof of B.
- $\bot$ (contradiction) has no proof; $\neg A$ is taken as an abbreviation of $A \to \bot$.
- A proof of $\forall x.A(x)$ is a method of transforming a proof of $d \in D$, where D is the intended domain of variables, to a proof of $A(d)$.
- A proof of $\exists x.A(x)$ is given by presenting an element d of the intended domain D together with a proof of $A(d)$.

This view is known as the Brouwer-Heyting-Kolmogorov interpretation. It turns out that the law of the excluded middle $A \vee \neg A$ is not necessarily true in this view, because one may not be able to tell whether A is true or $\neg A$ is. Logical systems in which $A \vee \neg A$ is derivable are called 'classical'.

Historically, intuitionism was advocated by L.E.J.Brouwer and others in the early twentieth century from a certain philosophical point of view. Apart from

[13] See footnote 11.

[14] First, for any proof system T, when a formula A is derivable from Γ in the system T we write $\Gamma \vdash_T A$ as in the case of NK and NJ. For two proof systems T and T', if all T-formulas (i.e., formulas of the system T) are T'-formulas, and moreover $\Gamma \vdash_T A \implies \Gamma \vdash_{T'} A$ holds for every T-formula A and every set Γ of T-formulas, then we say T' is an *extension* of T. A *conservative extension* of a proof system T is an extension T' of T in which $\Gamma \vdash_T A \iff \Gamma \vdash_{T'} A$ holds for every T-formula A and every set Γ of T-formulas.

[15] In general, when S is a set of logical constants, the *S-fragment* of a proof system T means the subsystem of T obtained by restricting formulas to those consisting of atomic formulas and logical constants in S, and restricting inference rules accordingly.

their philosophical motivations, constructive logics have been attracting much attention recently both in mathematical logic and in theoretical computer science (cf. [29], [45], [48]). We will see in section 5 that intuitionistic logics can be embedded in some type systems very smoothly so that one might consider type theory as the theory of computation which is founded on the ground of intuitionistic logics. On the other hand, proof theory can be encouraged by and can benefit from development of type theory through intuitionistic logics. A typical contribution from type theory to proof theory is the proof by J.-Y.Girard [17] of the consistency of second-order Peano arithmetic through the consistency of its intuitionistic counterpart, second-order Heyting arithmetic. He proved the latter by introducing a second-order type system $\lambda 2$ and proving its strong normalization theorem. We will come back to the subject in 5.2.

We close this subsection with a remark on the sequent-style presentation of the systems NK and NJ. The two systems can be presented directly in terms of the derivability relation $\Gamma \vdash_T A$ where $T \in \{\mathrm{NK}, \mathrm{NJ}\}$, which in this context is called a *sequent*. For example, in terms of sequents, the rule $(\to\mathrm{I})$ can be stated as

$$(\to\mathrm{I}) \quad \Gamma \cup \{A\} \vdash B \quad \text{implies} \quad \Gamma \vdash A \to B$$

(that is, if B is derivable from $\Gamma \cup \{A\}$ then $A \to B$ is derivable from Γ), while the rules $(\to\mathrm{E})$ and (start) can be stated respectively as follows.

$$(\to\mathrm{E}) \quad \Gamma \vdash A \to B \quad \text{and} \quad \Gamma \vdash A \quad \text{implies} \quad \Gamma \vdash B,$$
$$(\text{start}) \quad \Gamma \vdash A \quad \text{if } A \in \Gamma.$$

2.4 Other proof systems

In addition to natural deduction systems, Gentzen [15] introduced two other proof systems LK and LJ which are defined only in the sequent style in a perfectly symmetric way, and they are called the 'sequent-style systems'. The system LK is equivalent[16] to NK and so it is a classical system, while LJ is equivalent to NJ hence it is intuitionistic. The sequent-style systems have some advantages for theoretical purposes: Indeed, the theorems for LK and LJ corresponding to the normalization theorems for natural deduction systems are stated in a clear way as the 'cut-elimination theorems' [15], and the latter inspired the former. On the other hand, natural deduction systems have advantages in other aspects, such as affinity with human deduction and affinity with many important type systems as we will see in section 5. For the proof theory of sequent-style systems, see the articles by H. Ono and M. Okada in this volume and their references. The article by H. Yokouchi in this volume discusses among others some type systems relevant to LK and LJ.

Other important proof systems are so-called 'Hilbert-style' systems. Such a system equivalent to NK (or NJ) can be obtained from NK (or NJ) by replacing

[16] If two proof systems have the same derivability relation, the two systems are said to be *equivalent*.

the rules except ($\rightarrow$E) and ($\forall$I) with appropriate axioms. Historically they are forerunners among all proof systems, and indeed Gödel proved his completeness theorem for one of these (classical) systems. Hilbert-style systems have an advantage that they are conceptually simple, because they are free from the 'discharge' mechanism. As standard textbooks on mathematical logic based on Hilbert-style systems, [38] and [30] are recommended. The $\{\rightarrow\}$-fragment of an intuitionistic Hilbert-style system has a close relation with a certain type system based on combinatory logic (cf. 4.1.4). See for example [21] Ch.14 and [47] §2.4 for this subject.

3 Computable functions

In this section, we will see what the notion of 'finite procedures' can mathematically be. For this purpose, first let us define a class of functions of natural numbers which are apparently computable in any meaning of the word. As we mentioned before, by *natural numbers* we mean non-negative integers 0,1,2,..., and we write $\mathbf{N}$ for their totality.

3.1 Recursive functions

3.1.1 Definition The following functions are called *primitive recursive functions.*

1. $\mathrm{zero}^n(x_1, x_2, \ldots, x_n) = 0$ for each $n, x_1, x_2, \ldots, x_n \in \mathbf{N}$.
2. $\mathrm{suc}(x) = x + 1$ for each $x \in \mathbf{N}$.
3. $\mathrm{p}_i^n(x_1, \ldots, x_n) = x_i$ for each $n, i, x_1, \ldots, x_n \in \mathbf{N}$ such that $1 \leq i \leq n$.
4. The function $f : \mathbf{N}^n \rightarrow \mathbf{N}$ defined by composition as

$$f(\vec{x}) = g(g_1(\vec{x}), \ldots, g_m(\vec{x})) \quad (\vec{x} \in \mathbf{N}^n)$$

   where $g : \mathbf{N}^m \rightarrow \mathbf{N}$ and $g_1, \ldots, g_m : \mathbf{N}^n \rightarrow \mathbf{N}$ are primitive recursive functions, and $n, m \in \mathbf{N}$.
5. The function $f : \mathbf{N}^{n+1} \rightarrow \mathbf{N}$ defined by *primitive recursion* as

$$f(\vec{x}, 0) = g(\vec{x}),$$
$$f(\vec{x}, y + 1) = h(\vec{x}, y, f(\vec{x}, y)) \quad (\vec{x} \in \mathbf{N}^n,\ y \in \mathbf{N})$$

   where $g : \mathbf{N}^n \rightarrow \mathbf{N}$ and $h : \mathbf{N}^{n+2} \rightarrow \mathbf{N}$ are primitive recursive functions, and $n \in \mathbf{N}$.

Thus the primitive recursive functions are precisely the functions of natural numbers which are obtained by composition and primitive recursion starting from the constant functions, the successor function, and the projections. We will write $g \circ (g_1, \ldots, g_m)$ for the function f in Definition 3.1.1.4, and $h * g$ for the function f in Definition 3.1.1.5.

3.1.2 Example Addition $\mathrm{add}(x,y) = x + y$ is a primitive recursive function, because $\mathrm{add}(x,0) = x$, $\mathrm{add}(x,y+1) = \mathrm{add}(x,y) + 1$, hence $\mathrm{add} = h * g$ where $h = \mathrm{suc} \circ \mathrm{p}_3^3$ and $g = \mathrm{p}_1^1$. The predecessor function $\mathrm{pred} : \mathbf{N} \to \mathbf{N}$ defined by

$$\mathrm{pred}(0) = 0, \quad \mathrm{pred}(y+1) = y$$

is also primitive recursive, since $\mathrm{pred} = \mathrm{p}_1^2 * \mathrm{zero}^0$. Likewise, $x \times y$, x^y, $x!$, the function $\mathrm{prime}(x)$ to return the x-th prime number, the function $\mathrm{sqrt}(x,y)$ to return the x-th digit of $\sqrt{y}$, etc. are shown to be primitive recursive.

Next we extend the notion of primitive recursive functions to that of recursive functions. While the domains of primitive recursive functions are of the form $\mathbf{N}^n$ for some n, the domains of recursive functions are in general subsets of $\mathbf{N}^n$.

3.1.3 Definition The following functions are called *recursive functions.*[17]

1. The function f defined by

$$f(\vec{x}) = y \iff y = \min\{z | g(\vec{x},z) = h(\vec{x},z)\} \quad (\vec{x} \in \mathbf{N}^n,\ y \in \mathbf{N})$$

   where $g, h : \mathbf{N}^{n+1} \to \mathbf{N}$ are primitive recursive functions.

2. The function f defined by

$$f(\vec{x}) = y \iff \text{there exist } y_1, \ldots, y_m \text{ such that } g_1(\vec{x}) = y_1, \ldots, g_m(\vec{x}) = y_m, g(y_1, \ldots, y_m) = y$$

   where $g, g_1, \ldots, g_m$ are recursive functions.

Note that for the function f in Definition 3.1.3.1 the value $f(\vec{x})$ for a given $\vec{x} \in \mathbf{N}^n$ is defined if and only if there is a $z \in \mathbf{N}$ such that $g(\vec{x},z) = h(\vec{x},z)$,[18] and if this is the case $f(\vec{x})$ denotes the minimum value among all such z's. We will write $g \downarrow h$ for the function f. The function f in Definition 3.1.3.2 will be denoted by $g \circ (g_1, \ldots, g_m)$, as before. Note however that here for some $\vec{x}$ the value of the function $g \circ (g_1, \ldots, g_m)$ may be undefined; it is defined if and only if $g_1(\vec{x}), \ldots, g_m(\vec{x})$ are all defined and moreover $g(g_1(\vec{x}), \ldots, g_m(\vec{x}))$ is defined.

Note also that every primitive recursive function $g : \mathbf{N}^n \to \mathbf{N}$ is a (total) recursive function, since $g(\vec{x}) = \min\{z | g(\vec{x}) = z\}$ for each $\vec{x}$.

3.2 Description of computation

Intuitively speaking, the definitions 3.1.1 and 3.1.3 themselves suggest how to compute values of recursive functions. To make the intuition precise, let us

[17] In the literature, recursive functions are sometimes called *partial* recursive functions to emphasize the fact that their domains are not necessarily $\mathbf{N}^n$ but its subsets. In this respect, a function whose domain is $\mathbf{N}^n$ (for some n) may be called a *total* function.

[18] As usual, we understand that nothing is the minimum element of the empty set.

introduce a simple language to describe computation processes. In this language, we will describe the processes by using *assignment statements* of the form

$$x := e$$

and *while statements* of the form

$$\textsf{while } e \neq e' \textsf{ do } [S; S'; \cdots; S''].$$

Here, x is a variable (in the sense of programming languages) which can keep a single natural number at a time, and e and e' are expressions to specify a natural number using the variables and certain functions (e.g., primitive recursive functions). In the while statement, $[S, S', \cdots, S'']$ stands for a finite sequence of assignment statements and/or while statements.

The assignment statements are used to change the values of variables, and the repetition of computation processes is specified by using the while statements. An execution of the assignment statement means to evaluate the expression e first by using current values of variables, and then assign the value to the variable x. An execution of the while statement is, as the wording itself suggests, to repeat the following process; first, evaluate the expressions e and e' and see whether the two values are equal or not, and if not execute the block of statements $S, S', \ldots, S''$ (sequentially from left to right). On the other hand, if the values of e and e' are equal, the execution of the while statement terminates, and the flow of control proceeds to the next statement.

A finite sequence of these statements enclosed by an *input statement* at the beginning and an *output statement* at the end in the following form makes a *program* in our language.

$$\textsf{input}(\vec{x}); S; S'; \ldots; S''; \textsf{output}(x)$$

Here $\vec{x}$ is a finite sequence of distinct variables, called the *input variables*, and x is the *output variable*.

When input data $\vec{m} \in \mathbf{N}^n$ are given to the input variables $\vec{x} = (x_1, x_2, \ldots, x_n)$ of the above program, say P, the program P executes statements $S, S', \ldots, S''$ one by one in this order, and if the control eventually reaches the output statement it outputs the current value, say m, of the output variable x. (If not, there will be no output.) The relation between the input $\vec{m} \in \mathbf{N}^n$ and the output $m \in \mathbf{N}$ so obtained defines a function, which we call the *function computed by* P. Note that input data for which the program does not terminate and so gives no output are not included in the domain of the function; therefore in general the domain is a subset of $\mathbf{N}^n$.

Let us assume, for the moment, computability of the primitive recursive functions, and use them in the expressions e and e' in assignment/while statements. Then the following is a program to compute the function $f = g \downarrow h$ defined in Definition 3.1.3.1.

$\textsf{input}(\vec{x});$
$y := 0;$
$\textsf{while } g(\vec{x}, y) \neq h(\vec{x}, y) \textsf{ do } [y := y + 1];$
$\textsf{output}(y)$

Likewise, one can easily write a program to compute the composition function $f = g \circ (g_1, \ldots, g_m)$ defined in Definition 3.1.3.2, assuming that programs to compute $g, g_1, \ldots, g_m$ are provided. Therefore by inductive argument we know that for every recursive function f one can write a program to compute f.

What is somewhat unexpected is the fact that the converse also holds; that is, all functions computed by our programs are recursive functions. The outline of the proof proceeds as follows. First, we verify that any function computed by a program (with primitive recursive functions in assignment/while statements) can be computed by a program of the form

$$\begin{array}{l}\mathsf{input}(\vec{x});\\ w := g_1(\vec{x});\\ \mathsf{while}\ g_2(w) \neq g_3(w)\ \mathsf{do}\ [w := g_4(w)];\\ y := g_5(w);\\ \mathsf{output}(y)\end{array}$$

where $g_1 \sim g_5$ are primitive recursive functions. In order to prove this, one heavily relies on the Gödel numbering technique to encode finite sequences of natural numbers by natural numbers using primitive recursive functions.[19] Next, the function f computed by the five-line program above is shown to satisfy

$$f(\vec{x}) = h_5(\vec{x}, (h_2 \downarrow h_3)(\vec{x}))$$

where h_i $(i = 2, 3, 5)$ are the primitive recursive functions defined by[20]

$$h_i = g_i \circ ((g_4 \circ p_3^3) * p_1^2) \circ (g_1 \circ (p_1^{n+1}, p_2^{n+1}, \ldots, p_n^{n+1}), p_{n+1}^{n+1})$$

Thus f is shown to be a recursive function.

By the argument above, we know that a function is recursive if and only if it is computed by a program (using primitive recursive functions in assignment/while statements).

As for the computability of primitive recursive functions, one can verify the following: Any primitive recursive function can be computed by a program which contains only assignment statements of the form

$$y := 0 \quad \text{or} \quad z := z + 1$$

and only while statements of the form

$$\mathsf{while}\ y \neq z\ \mathsf{do}\ [S; S'; \cdots; S'']$$

where y and z are arbitrary variables.

[19] This technique was developed and extensively used by Gödel in proving the incompleteness theorems. In his proofs, the technique was used for encoding not only finite sequences of natural numbers, but more general kinds of finitely representable objects such as logical formulas and proof trees, in order to internalize proof-theoretic arguments in PA.

[20] See 3.1.1 and 3.1.3 for the notations. Here n is the number of input variables in $\vec{x}$.

Let us call a program consisting of assignment statements and while statements of these restrictive forms a *low-level program*, and call a program using arbitrary primitive recursive functions in assignment/while statements a *high-level program*. Then we can summarize the previous arguments as follows.

3.2.1 Theorem For a function f from a subset of $\mathbf{N}^n$ to $\mathbf{N}$, the following conditions are equivalent. (i) f is a recursive function. (ii) f is computed by a high-level program. (iii) f is computed by a low-level program.

The descriptive power of our language, when it is compared with practical programming languages, appears extremely weak, because the only data structure available in our language is natural numbers and no structured data such as vectors, matrices, trees, graphs, functions, etc. are available. Also our language has only the while statement to control the flow of the program, and even (recursive or non-recursive) subprograms are not available. However it can be shown that various useful concepts available in traditional or modern programming languages can be 'implemented' in our language by using the Gödel numbering technique.

Based on this fact and other evidences that the notion of recursive functions coincides with each of several other notions of 'computable functions' that have been proposed, for example those by using Turing Machines [49], λ-calculus (cf. section 4), etc., we will call recursive functions simply *computable functions*, and by *finite procedures* we will understand any of the mutually equivalent computational models.

3.3 Non-computable functions and undecidable problems

Now a natural question arises; what functions are not computable? The mere existence of non-computable functions is easy to see by a cardinality argument: There are only countably many recursive functions while there are uncountably many functions over natural numbers, hence there should be (uncountably many) non-computable functions.

As a concrete example, we can exhibit a function f from a proper subset of $\mathbf{N}^2$ to $\mathbf{N}$ such that f itself is computable, but every total extension[21] of f is not.

The idea of how to construct such a function f is the following. First note that a low-level program in 3.2 is, from the syntactic viewpoint, described by a finite sequence over a finite set of symbols,[22] hence it can be encoded by means of a natural number using the Gödel numbering technique. Now let us concentrate on low-level programs with a single input variable, and write f_k for the function computed by such a program with code $k \in \mathbf{N}$. The function f is then defined by

$$f(k,x) = y \iff f_k(x) = y \text{ and } k \in K \quad (k,x,y \in \mathbf{N})$$

[21] I.e., a total function $g : \mathbf{N}^2 \to \mathbf{N}$ such that $f(\vec{x}) = g(\vec{x})$ holds whenever $f(\vec{x})$ is defined.

[22] The only trouble is the names of variables in programs, but for them we can use x, x', x'', x''', etc (consisting of symbols x and $'$).

where K stands for the set of all codes of low-level programs with one input variable. Then this function f is shown to be computable; indeed, it can be computed by an 'interpreter' of low-level programs with a single input variable, which, given $k \in K$ and $m \in \mathbf{N}$, outputs exactly when and what the program coded by k does for the input m, and such an interpreter can indeed be programmed in our language. The second property of f that every total extension of f is not recursive can be verified by the following diagonalization argument: Suppose a total extension $g : \mathbf{N}^2 \to \mathbf{N}$ of f were recursive. Then the function $g' : \mathbf{N} \to \mathbf{N}$ defined by $g'(x) = g(x, x) + 1$ would be a unary (total) recursive function. Hence $g' = f_k$ for some $k \in K$, and we have $g'(k) = f_k(k) = f(k, k) = g(k, k)$, while $g'(k) = g(k, k) + 1$, which is a contradiction.

From the function f above, we can obtain another non-computable function. Let $h : \mathbf{N}^2 \to \mathbf{N}$ be defined by

$$h(\vec{x}) = \begin{cases} 0 & \text{if } f(\vec{x}) \text{ is defined,} \\ 1 & \text{if } f(\vec{x}) \text{ is undefined.} \end{cases}$$

Then h is not computable, because otherwise one could easily write a program to compute a total extension of f based on programs to compute f and h, respectively. From the fact that h is not computable, we know that for some program P (indeed, for the one to compute the function f) no program can predict whether P terminates or not for given inputs. That is, the problem cannot be answered by finite procedures.

In general, for a predicate $p(\vec{x})$ of natural numbers (i.e., a function of natural numbers whose values are either true or false), if its characteristic function

$$\chi_p(\vec{x}) = \begin{cases} 0 & \text{if } p(\vec{x}) = \text{true,} \\ 1 & \text{if } p(\vec{x}) = \text{false.} \end{cases}$$

is computable, we say that the decision problem of p is *decidable* and otherwise *undecidable*. From the above argument, we know the undecidability of the problem asking for the above function f whether $f(\vec{x})$ for a given $\vec{x} \in \mathbf{N}^2$ is defined or not.

We can extend the domain of decision problems in a natural way from the set of natural numbers to a certain set of objects which can be described by (finite) words over a finite alphabet. For example, we can think of decision problems on (low-level or high-level) programs, those on recursive functions, those on logical formulas with a finite set of function/predicate symbols, and so forth. For example, immediately from the above argument, we get the undecidability of the 'halting problem' of programs which asks whether a given program eventually stops for a given input data or not.

As for the decision problems concerning logical formulas, it follows from the first incompleteness theorem by Gödel that the decision problem asking whether a given PA-formula A is true or not (under the standard interpretation[23]) is undecidable.

[23] By the standard interpretation of formulas of PA, we mean the interpretation over the set $\mathbf{N}$ of natural numbers in which 0, suc, $+$, $\times$ and $=$ are interpreted as the natural number 0, the successor function, addition, multiplication, and equality (between natural numbers).

It is also shown to be undecidable whether a given PA-formula is provable in PA or not. A decision problem related to pure proof systems (without axioms), which is also known to be undecidable, is the one asking whether a given logical formula A holds true always (i.e., $\models A$, or equivalently, $\vdash_{\mathrm{NK}} A$). Note that the corresponding problem for propositional logic (i.e., the $\{\wedge, \vee, \neg, \rightarrow\}$-fragment of NK) is decidable, because one can easily give the answer by writing truth tables, and the process of writing truth tables can be described in a program.

For more on the subject of this section, consult for example [12] and [32].

4 λ-calculus

A mathematical framework to study the mechanism of functional abstraction and application was introduced by Church [6] as a part of his logical system. Although his system was found to be inconsistent (cf. [36]), the framework of functional abstraction and application itself, which is called the λ-calculus, was shown to be one of the key notions to characterize computability (see Theorem 4.2.3 below). Besides, since the first functional programming language LISP was designed incorporating the idea of the λ-calculus and implemented in the late 1950's, the notion of λ-calculus has become popular among the computer science community, and its significance has gradually been recognized. In particular, the discovery by D.Scott [41] of models of λ-calculus initiated the domain theory; that is, mathematical studies of semantics of programming languages based on the models. In addition, more recently active studies on various type systems based on the λ-calculus has been going on (cf. section 5). These developments show that the notion of λ-calculus is much richer than one might have anticipated, and that it actually bridges between theory of computation and proof theory in a different way from what Church originally intended.

So far, there have been introduced a quite number of formal systems relating to Church's idea. As a basis for all such systems, in this section we take a look at the core system with no typing discipline nor built-in constants, and we call it the *type-free pure λ-calculus*, or simply the *λ-calculus*.

Before introducing the λ-calculus, let us first look at a useful notation for functions. Given a mathematical expression e and a variable x ranging over a set D, the function which assigns the value of e to the value of x is denoted by $\lambda x \in D.e$, or simply by $\lambda x.e$ if the domain D is well understood. For example, $\lambda n \in \mathbf{N}.n + 1$ denotes the successor function over $\mathbf{N}$, while $\lambda y \in \mathbf{R}.x^y$ may denote the function $f : \mathbf{R} \rightarrow \mathbf{R}$ such that $f(y) = x^y$. This notation, called the *λ-notation*, is particularly useful in representing higher-order functions. For example, when we write $(D \rightarrow D)$ for the set of unary functions over D, by $\lambda f \in (D \rightarrow D).f \circ f$ we mean the higher-order function (or functional) which assigns the composition $f \circ f$ to each f in $(D \rightarrow D)$, while $f \circ f$ can be expressed as $\lambda x \in D.f(f(x))$.

Although the λ-notation is primarily for unary functions, it can also express functions of any positive arity. For example, given a binary function $f : X \times Y \rightarrow Z$,

let us define $f' = \lambda x \in X.(\lambda y \in Y.f(x,y)) : X \to (Y \to X)$. Then for each $x \in X$ and $y \in Y$ we have $f'(x)(y) = f(x,y)$. In this way, any binary function f can be expressed by the (higher-order) unary function f', which we call the *Curried* function of f (after H.B.Curry, one of the pioneers in this field). The same argument applies to functions of the arity three or more. By convention, we will write arguments of Curried functions without parentheses as $f'xy$, instead of $f'(x)(y)$.

The λ-notation is also useful in defining functions recursively, which is very common in functional programming. For example, one can write a program to compute the factorial function $\mathrm{fact}(x) = x!$ in functional programming languages essentially as

$$\mathrm{fact} = \lambda x \in \mathbf{N}.(\ \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ \mathrm{fact}(x-1) \times x).$$

What this means is that the function fact is defined as the unique fixed point of the higher-order function

$$F = \lambda f \in (\mathbf{N} \to \mathbf{N}).(\lambda x \in \mathbf{N}.(\ \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(x-1) \times x));$$

that is, as the function f satisfying $f = F(f)$.

4.1 Definitions and examples

In the λ-calculus, the most basic notions are those of λ-terms and reduction between them. All well-formed expressions in the system, including functions and their arguments, are called λ-terms. Note that when higher-order functions are taken into account, functions may also be arguments of some functions.

4.1.1 Definition The set Λ of *λ-terms* is defined recursively, as follows.

1. Variables in $\mathrm{Var} = \{u_0, u_1, u_2, \ldots\}$ are λ-terms.
2. If M is a λ-term and $u \in \mathrm{Var}$, then $(\lambda u.M)$ is a λ-term.
3. If M and N are λ-terms, then (MN) is a λ-term.

We call a λ-term of the form $(\lambda u.M)$ an *abstraction* (thinking of M as being abstracted to a function assigning M to u), and a λ-term of the form (MN) an *application* (thinking of M as being a function applied to N). In this section, unless otherwise stated we assume $L, M, N, \ldots$ stand for arbitrary λ-terms, and $u, v, \ldots$ for distinct variables. In order to avoid too many parentheses in λ-terms, we will write $\lambda u_1 u_2 \cdots u_n.M$ for the λ-term $(\lambda u_1.(\lambda u_2.(\cdots(\lambda u_n.M)\cdots)))$, and write $LN_1N_2\cdots N_n$ for $((\cdots((LN_1)N_2)\cdots)N_n)$.

An occurrence of variable u in a λ-term is said to be *bound* if it is contained in a subterm of the form $\lambda u.M$, and otherwise it is *free*. When λ-terms L and L' differ only in the names of bound variables, we identify them and write $L \equiv L'$.

Next we define the notion of β-reduction of λ-terms, which represents computation steps of λ-terms.

4.1.2 Definition First let us denote by $M[u := N]$ the result of substitution of N for every free occurrence of u in M. When a λ-term L contains a subterm of the form $(\lambda u.M)N$, and L' is obtained from L by replacing the subterm with $M[u := N]$, we say L is contracted to L' by *one-step β-reduction*, and write $L \to_\beta L'$. Thus one-step β-reduction can schematically be written as

$$\ldots(\lambda u.M)N\ldots \underset{\beta}{\to} \ldots M[u := N]\ldots .$$

The λ-term $(\lambda u.M)N$ is called a *β-redex* (meaning a *β-reducible expression*).

The intuition behind $\to_\beta$ is the following: In β-redex $(\lambda u.M)N$, the subterm $\lambda u.M$ represents the function with u as the formal argument and M as the function body, while N represents an actual argument supplied to the function. On the other hand, $M[u := N]$ denotes the result obtained by substituting the actual argument N for the formal argument u in the body M of the function. Thus $\to_\beta$ can be considered as a formulation of a primitive step of evaluation of functional expressions.

Next, when we have a finite sequence of one-step β-reductions

$$L_0 \underset{\beta}{\to} L_1 \underset{\beta}{\to} L_2 \underset{\beta}{\to} \cdots \underset{\beta}{\to} L_n$$

where $n \geq 0$, we say L_0 is *β-reducible* to L_n, and write $L_0 \twoheadrightarrow_\beta L_n$. In other words, $\twoheadrightarrow_\beta$ is the reflexive and transitive closure of $\to_\beta$. The reflexive, symmetric and transitive closure of $\to_\beta$ (that is, the equivalence relation generated by $\to_\beta$) is denoted by $=_\beta$.

4.1.3 Examples (fixed point operator) Given a λ-term M and a variable u not free in M, let $M' \equiv \lambda u.M(uu)$. Then $M'M' \equiv (\lambda u.M(uu))M' \to_\beta M(M'M')$. Thus for every λ-term M there exists a λ-term X satisfying $X =_\beta MX$. We will call such a λ-term X a *fixed point* of M. Next, let $Y \equiv \lambda v.((\lambda u.v(uu))(\lambda u.v(uu)))$. Then

$$YM \underset{\beta}{=} (\lambda u.M(uu))(\lambda u.M(uu)) \equiv M'M' \underset{\beta}{=} M(M'M') \underset{\beta}{=} M(YM),$$

which means that YM is another fixed point of M. Thus Y plays the role of fixed point operator (to yield for each M a fixed point of M).

4.1.4 Examples (combinatory logic) Let $S \equiv \lambda uvw.uw(vw)$ and $K \equiv \lambda uv.u$. Then we have

$$SKK \underset{\beta}{\twoheadrightarrow} \lambda w.Kw(Kw) \underset{\beta}{\twoheadrightarrow} \lambda w.w.$$

More generally, we can prove that for any λ-term M there exists a λ-term L such that $L \twoheadrightarrow_\beta M$ and L is constructed by application from the λ-terms S and K above and variables in $\mathrm{FV}(M)$.[24] This suggests a close relationship between the structure

[24] Variables which have free occurrences in M are called *free variables* of M, and their totality is denoted by $\mathrm{FV}(M)$.

of the set Λ of λ-terms and an algebraic structure associated with the combinatory logic (cf. [3] §7.3, [21] Ch.9). First, let CL be the set of the first-order terms, called CL-*terms*, constructed by binary operator $\cdot$ from two constants s, k and variables, and $=_{\mathrm{w}}$ be the congruence relation over CL generated by[25] $sabc = ac(bc)$ and $kab = a$ for every $a, b, c \in CL$. Under this definition, it can be shown that the quotient structure[26] $(\Lambda/_{=_\beta}, S, K, \cdot)$ of Λ is a homomorphic image of the quotient structure $(CL/_{=_{\mathrm{w}}}, s, k, \cdot)$, called the free combinatory algebra. Moreover, when we consider, instead of $=_\beta$, the extensional congruence relation $=_{\beta\eta}$, i.e., the weakest congruence relation containing $=_\beta$ and satisfying the extensionality condition

$$\forall N[MN \underset{\beta\eta}{=} M'N] \implies M \underset{\beta\eta}{=} M',$$

and likewise replace the relation $=_{\mathrm{w}}$ between CL-terms to the weakest extensional congruence $=_{\mathrm{w}\eta}$, the two quotient structures $(\Lambda/_{=_{\beta\eta}}, S, K, \cdot)$ and $(CL/_{=_{\mathrm{w}\eta}}, s, k, \cdot)$ are shown to be isomorphic. Roughly speaking, this means that the role played by the λ-abstraction in λ-calculus can almost be confined to the two λ-terms S and K, and if the extensionality is assumed this is so in a strict sense. See for example [42] and [21] for combinatory logic and its relation to λ-calculus.

4.2 Fundamental properties of λ-calculus

A λ-term which contains no β-redex is said to be in *β-normal form* (*β-nf*, for short), and when $L \twoheadrightarrow_\beta M$ and M is in β-nf, we say L is *normalizing* and M is a *β-nf of L*. Note that some λ-terms are not normalizing; take for example the λ-term LL where $L \equiv \lambda u.uu$ or $L \equiv \lambda u.uuu$. The fixed point operator Y in 4.1.3 is another such example. But it is guaranteed by the next theorem that if a λ-term is normalizing then its β-nf is unique (up to renaming of bound variables).

4.2.1 Church-Rosser theorem for λ-calculus If $L \twoheadrightarrow_\beta M_i$ $(i = 1, 2)$, then there exists N such that $M_i \twoheadrightarrow_\beta N$ $(i = 1, 2)$.

The theorem can easily be extended, as follows: If $M_1 =_\beta M_2$, then there exists N such that $M_i \twoheadrightarrow_\beta N$ $(i = 1, 2)$. As a consequence, we know that if $M_1 =_\beta M_2$ and M_2 is in β-nf, then $M_1 \twoheadrightarrow_\beta M_2$.

It is possible for some λ-terms M that, even though M is normalizing, there exists an infinite β-reduction sequence starting from M. That is, a clever choice of β-reduction steps can lead M to a result (i.e., a λ-term in β-nf), but a poor choice may lead M to an infinite computation. For example, consider $(\lambda uv.v)(LL)$ where $L \equiv \lambda u.uu$. The following theorem tells us what strategy we should take in order

[25] As in the case of the application operator for λ-terms, the binary operator $\cdot$ for CL-terms is assumed to be associated to the left and usually suppressed.

[26] By abuse of notation, here we indicate by S and K the equivalence classes containing the λ-terms S and K, respectively, and $\cdot$ stands for the operator on equivalence classes induced from the application operator on Λ. We practise similar abuses of notations for other quotient structures in this subsection for the sake of simplicity.

to avoid infinite computation if possible.

4.2.2 Normalization theorem for λ-calculus If M is normalizing, then M is led to its β-nf by the leftmost β-reduction. Here the leftmost β-reduction means the sequence of β-reduction steps in which always the outermost leftmost β-redexes are contracted.

The theorems 4.2.1 and 4.2.2 can be proved by induction based on the notion of 'parallel β-reduction', which is originally due to Tait and Martin-Löf (cf. [44]).

As mentioned before, the λ-calculus can be considered as a computational model. For this purpose, first we note that natural numbers can be represented by the λ-terms

$$\overline{n} \equiv \lambda u.\lambda v.\overbrace{v(v(\cdots(v}^{n} u)\cdots)) \quad (n = 0, 1, 2, \ldots)$$

called *numerals*. The idea behind it is the following; in the pure λ-calculus, which has only functional abstraction and application as the basic constructors, it is natural to represent a natural number n by n times application $L(L(\cdots(LM)\cdots))$ of a function L to an argument M, which we write $L^n(M)$. But, since there is no reason to pick up particular λ-terms L and M, we take the abstraction $\overline{n} \equiv \lambda u.\lambda v.v^n(u)$, from which one can obtain $L^n(M)$ as $\overline{n}LM \twoheadrightarrow_\beta L^n(M)$.

Once we fix the representation of natural numbers, next we define the representation of numerical functions in λ-calculus. We say a function f from a subset of $\mathbf{N}^n$ to $\mathbf{N}$ is *represented* by a λ-term L if the following holds for each $m_1, \ldots, m_n, m \in \mathbf{N}$;

$L\overline{m_1}\,\overline{m_2}\cdots\overline{m_n} \twoheadrightarrow_\beta \overline{m}$ if $f(m_1, \ldots, m_n) = m$,
$L\overline{m_1}\,\overline{m_2}\cdots\overline{m_n}$ has no β-nf if $f(m_1, \ldots, m_n)$ is undefined.

In this case, the λ-term L is said to be a *λ-representation* of f. Under this definition, it can be proved that all recursive functions (defined in 3.1.3) are λ-representable (that is, represented by a λ-term). In fact, given a recursive function f one can effectively construct a λ-representation of f possibly by using a fixed point operator to solve defining equations of recursive functions. We can also prove the converse; one can write a program to compute a λ-representable function, based on the leftmost β-reduction strategy mentioned in theorem 4.2.2. Thus we have the following.

4.2.3 Theorem A numerical function f is computable if and only if it is λ-representable.

As a consequence, we get an undecidability result: It is undecidable whether a λ-term is normalizing or not, since otherwise the halting problem (cf. 3.3) would become decidable. More generally, for any non-trivial subset of Λ which is closed under $=_\beta$, the question of its membership is shown to be undecidable.

In passing, we note that the extensional equivalence relation $=_{\beta\eta}$ discussed in 4.1.4 can be generated by the union $\rightarrow_{\beta\eta}$ of the one-step β-reduction $\rightarrow_\beta$ and the

one-step *η-reduction* $\to_\eta$ which is schematically defined as

$$\ldots \lambda u.Mu \ldots \underset{\eta}{\to} \ldots M \ldots \qquad (u \notin \mathrm{FV}(M)).$$

We also note that the theorems 4.2.1 $\sim$ 4.2.3 hold true when we replace the β-reduction $\twoheadrightarrow_\beta$ in the theorems with the *βη-reduction* $\twoheadrightarrow_{\beta\eta}$, i.e., the reflexive transitive closure of $\to_{\beta\eta}$, and replace the notion of β-nf and that of leftmost β-reduction accordingly (cf. e.g. [44]).

A characteristic feature of λ-calculus is that any λ-term can be applied as a function to any λ-term, so that a function can even be applied to itself. Thus the functions denoted by λ-terms are 'type-free'. In other words, the functions share the set of all denotations of λ-terms as their domains.

Then a natural question arises; what kind of mathematical structure does the set of all denotations of λ-terms have? The question had been open until D.Scott [41] proved that the inverse limit of certain continuous lattices provides such a structure. Since then a number of other ways have been discovered to provide such structures, which are called *λ-models*. Here we only mention that in algebraic terms, the notion of λ-models can be identified with the algebraic structure $(D, s, k, e, \cdot)$ where $\cdot$ is a binary operator and s, k are constants satisfying the two axioms mentioned in 4.1.4 (i.e., $sabc = ac(bc)$ and $kab = a$ for each $a, b, c \in D$) and e is a constant satisfying for each $a, b, c \in D$ the axioms $eab = ab$ and $\forall c \in D(ac = bc) \Longrightarrow ea = eb$. In categorical terms, the notion can be identified with that of well-pointed cartesian closed categories endowed with a reflexive object. See the article by M. Dezani-Ciancaglini et al. in this volume for an account of λ-models. See also the standard texts on the λ-calculus (e.g., [3], [21], [43]).

5 Type systems

In this section, we look at three major type structures built upon the (type-free) λ-calculus, and see what one can do with types. Our emphasis is on proof theoretic aspects of type systems.

5.1 Simple type system $\lambda_\to$

First, we consider the *simple type system* $\lambda_\to$ introduced by Church [8]. What it does is to restore the usual notion of domains and ranges of functions which are ignored in the (type-free) λ-calculus in the previous section. This system provides a basis for all other type systems.

5.1.1 Definition The types of the system $\lambda_\to$, called *simple types*, are expressions constructed from atomic types by functional type constructor $\to$. More precisely, simple types are defined recursively, as follows.

- Atomic types $\alpha_0, \alpha_1, \alpha_2, \ldots$ are simple types.

- If A and B are simple types, then so is $(A \to B)$.

The terms dealt with in $\lambda_{\to}$, which are called *simple terms*, are either variables, abstractions, or applications, as in the case of (type-free) λ-terms. However the difference is that simple terms are always associated with types. The formation rules of simple terms are as follows.

- If u is a variable of type A, then u is a simple term of type A.
- If u is a variable of type A and M is a simple term of type B, then $(\lambda u : A.M)$ is a simple term of type $A \to B$.
- If M is a simple term of type $A \to B$ and N is a simple term of type A, then (MN) is a simple term of type B.

Note that types of simple terms depend on types of their free variables,[27] and in the last case above the types of the free variables in M and N must be consistent. In order to make the point explicit, we have to state the formation rules of simple terms by specifying type declarations (or contexts) $\Delta \subseteq \{u_0 : A_0,\ u_1 : A_1,\ \ldots\}$, as follows.[28]

$$\frac{}{\Delta \vdash u : A}\ (\text{start}) \quad \text{where} \quad (u : A) \in \Delta$$

$$\frac{\Delta, u : A \vdash M : B}{\Delta \vdash (\lambda u : A.M) : A \to B}\ (\to\text{I})$$

$$\frac{\Delta \vdash M : A \to B \quad \Delta \vdash N : A}{\Delta \vdash (MN) : B}\ (\to\text{E})$$

By applying these rules repeatedly, we can generate simple terms together with their types, as illustrated in the following figure.[29]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta \vdash u : (A \to B \to C)}\ (\text{start}) \quad \dfrac{}{\Delta \vdash w : A}\ (\text{start})}{\Delta \vdash (uw) : (B \to C)}\ (\to\text{E}) \quad \dfrac{\dfrac{}{\Delta \vdash v : (A \to B)}\ (\text{start}) \quad \dfrac{}{\Delta \vdash w : A}\ (\text{start})}{\Delta \vdash (vw) : B}\ (\to\text{E})}{\Delta \vdash (uw(vw)) : C}\ (\to\text{E})}{u : A \to B \to C, v : A \to B \vdash (\lambda w : A.uw(vw)) : (A \to C)}\ (\to\text{I})}{u : A \to B \to C \vdash (\lambda v : A \to B.\lambda w : A.uw(vw)) : ((A \to B) \to A \to C)}\ (\to\text{I})}{\vdash (\lambda u : A \to B \to C.\lambda v : A \to B.\lambda w : A.uw(vw)) : ((A \to B \to C) \to (A \to B) \to A \to C)}\ (\to\text{I})$$

Here we assume that $\Delta = \{u : A \to B \to C,\ v : A \to B,\ w : A\}$. In this figure, $\lambda u : A \to B \to C.\ \lambda v : A \to B.\ \lambda w : A.uw(vw)$ is shown to be a simple term of type $(A \to B \to C) \to (A \to B) \to A \to C$ under the empty type declaration; in

[27] We naturally extend the notion of free variables of λ-terms to that of simple terms.

[28] In a type declaration $\Delta = \{u : A,\ u' : A', \ldots\}$, variables $u, u', \ldots$ (which are distinct from each other) are declared to be of types $A, A', \ldots$, respectively. The type declaration $\Delta \cup \{u : A\}$ is often abbreviated as $\Delta, u : A$ as in the rule $(\to\text{I})$.

[29] We abbreviate $A_1 \to (A_2 \to (\cdots \to (A_{n-1} \to A_n) \cdots))$ as $A_1 \to A_2 \to \cdots \to A_n$.

notation,[30] $\vdash_{\lambda_\to} (\lambda u : A_1.\lambda v : A_2.\lambda w : A.uw(vw)) : (A_1 \to A_2 \to A \to C)$ where $A_1 = A \to B \to C$ and $A_2 = A \to B$. The upper half of the figure shows $uw(vw)$ is a simple term of type C under the type declaration Δ.

It is interesting to note a similarity between the formation rules of simple terms and the inference rules of natural deduction systems in section 2. In fact, when we look at only the type part of the formation rules of simple terms, we find precisely the sequent-style presentation of the rules (start), ($\to$I) and ($\to$E) of the natural deduction systems NK and NJ (cf. 2.3). Based on this observation, we get the following correspondence between formation of simple terms and derivability of formulas in the $\{\to\}$-fragment $\mathrm{NJ}_\to$ of the intuitionistic system NJ.

> For simple types $A_1, A_2, \cdots, A_n$ and B, the following two conditions are equivalent.
>
> 1. $u_1 : A_1, u_2 : A_2, \ldots, u_n : A_n \vdash_{\lambda_\to} M : B$ for some simple term M.
> 2. $A_1, A_2, \cdots, A_n \vdash_{\mathrm{NJ}_\to} B$.[31]

We can replace $\mathrm{NJ}_\to$ above with NJ, because NJ is a conservative extension of $\mathrm{NJ}_\to$ (cf. 2.3). However we cannot replace it with NK, because NK is not conservative over $\mathrm{NJ}_\to$.[32]

What is even more interesting is the fact that the simple term M in the condition 1 above precisely describes the history of the derivation in $\mathrm{NJ}_\to$ mentioned in the condition 2; each occurrence of variables, abstraction, and application in the former corresponds to an application instance of the rules (start), ($\to$I), and ($\to$E) in the latter, respectively. For example, the above example shows that in $\mathrm{NJ}_\to$ the formula $(A \to B \to C) \to (A \to B) \to A \to C$ is derivable, and a compact description of its derivation is given by the λ-term $M \equiv \lambda u : A \to B \to C.\lambda v : A \to B.\lambda w : A.uw(vw)$.

This correspondence between $\mathrm{NJ}_\to$ and $\lambda_\to$ is the simplest case among various correspondences between intuitionistic proof systems and type systems, and they are known as the 'formulas-as-types principle' or the 'Curry-Howard correspondence'. It is first noticed by Curry (cf. [11] §9E) in the case of simply typed combinatory logic, and extended to the case of first-order logic by Howard (cf. [22]). Then the idea has been further extended to many other cases; we will see some of them in the following subsections.

As in the case of λ-terms, we define the notion of β-reduction between simple terms: One-step β-reduction $\to_\beta$ is schematically defined as

$$\ldots(\lambda u : A.M)N \ldots \underset{\beta}{\to} \ldots M[u := N] \ldots$$

[30] As in the case of proof systems, for a type system T we use the notation $\Delta \vdash_T M : A$ (or $u : B, u' : B', \ldots, u'' : B'' \vdash_T M : A$) to mean that M is a legal term in T of the type A under the type declaration Δ (or $\{u : B, u' : B', \ldots, u'' : B''\}$). The subscript T of $\vdash_T$ may be omitted.

[31] Here $A_1, \ldots, A_n, B$ are considered as formulas of $\mathrm{NJ}_\to$.

[32] Recall that Peirce's law $((A \to B) \to A) \to A$ is derivable in NK, but not derivable in NJ, hence not in $\mathrm{NJ}_\to$.

and it is extended to $\twoheadrightarrow_\beta$ and $=_\beta$ as in 4.1.2. Then the types of simple terms are shown to be preserved by β-reduction (i.e., if $\Delta \vdash_{\lambda\to} M : A$ and $M \twoheadrightarrow_\beta N$ then $\Delta \vdash_{\lambda\to} N : A$), which is called the subject-reduction theorem.[33] The Church-Rosser theorem

$$L \underset{\beta}{\twoheadrightarrow} M_i \ (i = 1, 2) \implies M_i \underset{\beta}{\twoheadrightarrow} N \ (i = 1, 2) \text{ for some } N$$

also holds for simple terms.

A typed or untyped λ-term M is said to be *strongly normalizing* if there is no infinite β-reduction sequence starting from M. As we have seen in section 4, some λ-terms are not normalizing; hence not strongly normalizing. Therefore the following theorem reveals a big contrast between the type-free λ-calculus and the simple type system.

5.1.2 Strong normalization theorem for $\lambda\to$ All simple terms are strongly normalizing.

There is a number of ways to prove the theorem. We will outline one of them, which is extensible to higher-order type systems. The proof is originally due to Tait and Girard (cf. [18] Ch.6 and [4] §4.3).

Let us write $|M|$ for the (type-free) λ-term obtained from a simple term M by erasing all type information. Then clearly M is strongly normalizing if and only if so is $|M|$. Therefore in order to prove the theorem it suffices to show

$$\Delta \vdash_{\lambda\to} M : A \implies |M| \in \mathrm{SN}$$

where SN stands for the set of λ-terms which are strongly normalizing. We can prove the statement by introducing a set $[\![A]\!]$ of λ-terms associated with the simple type A, which is shown to intervene between $|M|$ and SN as

(1) $\Delta \vdash_{\lambda\to} M : A \implies |M| \in [\![A]\!] \subseteq \mathrm{SN}$.

The set $[\![A]\!]$ ($\subseteq \Lambda$) is defined recursively on the structure of A, as follows:

- $[\![\alpha]\!] = \mathrm{SN}$, if α is an atomic type,
- $[\![A \to B]\!] = \{M \in \Lambda \mid MN \in [\![B]\!] \text{ for each } N \in [\![A]\!]\}$.

In order to prove (1), we define the notion of saturated sets. A subset X of SN is said to be *saturated* if the following two conditions are satisfied.

- $uN_1N_2 \cdots N_n \in X$ for each $u \in \mathrm{Var}$, $n \geq 0$, and $N_1, \cdots, N_n \in \mathrm{SN}$.
- For each $u \in \mathrm{Var}$, $M \in \Lambda$, $n \geq 0$, and $N, N_1, \cdots, N_n \in \mathrm{SN}$, if we have $M[u := N]N_1N_2 \cdots N_n \in X$ then $(\lambda u.M)NN_1N_2 \cdots N_n \in X$.

First by induction on the structure of A we prove[34]

[33] The name comes from the tradition that, in the expression $M : A$, M is called the 'subject', and A is the 'predicate'.

[34] That is, we prove (2) assuming that it holds for all types properly contained in A.

(2) $[\![A]\!]$ is saturated.

As a consequence, we obtain $[\![A]\!] \subseteq \mathrm{SN}$, which shows the second half of (1). Next, in order to prove

(3) $\Delta \vdash_{\lambda_\to} M : A \implies |M| \in [\![A]\!]$,

we verify a stronger statement

(3)′ $\Delta \vdash_{\lambda_\to} M : A \implies \Delta \models M : A$

where $\Delta \models M : A$ is a shorthand for the statement

(3)″ $\forall \theta : \mathrm{Var} \to \Lambda \ [\forall (u : B) \in \Delta \ (\ \theta(u) \in [\![B]\!]\ ) \implies |M|\theta \in [\![A]\!]]$.

Here $|M|\theta$ stands for the result of simultaneously substituting $\theta(u)$ for free occurrences of variable $u \in \mathrm{Var}$ in $|M|$. One can prove (3)′ by induction on the structure of the derivation[35] of $\Delta \vdash_{\lambda_\to} M : A$, by using the fact (2). Once (3)′ is verified, we get immediately (3) and hence (1), because $|M| \in [\![A]\!]$ is a special case of $\Delta \models M : A$ where θ in (3)″ is the identity function $\theta(u) \equiv u$ for each $u \in \mathrm{Var}$. (Note that by definition all variables belong to saturated sets.) □

From the strong normalization theorem and the subject reduction theorem above mentioned, we know that if $\Delta \vdash_{\lambda_\to} M : A$ then there exists a simple term N in β-nf such that $\Delta \vdash_{\lambda_\to} N : A$. Then this implies through the Curry-Howard correspondence that if $\Gamma \vdash_{\mathrm{NJ}_\to} A$ then this fact can be derived by a normal derivation; that is, a derivation without detours (cf. 2.3). Thus as a corollary to theorem 5.1.2 we know that derivations in $\mathrm{NJ}_\to$ are (strongly) normalizing.

Also from the strong normalization theorem, one can easily see that there is no simple term M such that $|M|$ being a fixed point operator.

As another corollary to the strong normalization theorem, we get a decidability result for $\lambda_\to$. The decision problem of asking for arbitrary simple terms M, N whether $M =_\beta N$ or not is decidable, because by the Church-Rosser theorem $M =_\beta N$ holds if and only if $\beta\text{-nf}(M) \equiv \beta\text{-nf}(N)$ where $\beta\text{-nf}(M)$ stands for the β-nf of M, which indeed exists and can effectively be obtained by theorem 5.1.2. Compare this result with the undecidability of the corresponding problem for λ-terms (cf. 4.2).

The expressive power of the simple type system is very weak. To see this, first we define the notion of $\lambda_\to$-representable functions as before (cf. 4.2) except that the λ-term is replaced with the simple term, the numeral with the simple term $\overline{n} \equiv \lambda u : \alpha . \lambda v : \alpha \to \alpha . v^n(u)$ where α is a fixed atomic type, and the β-reduction $\twoheadrightarrow_\beta$ with the $\beta\eta$-reduction $\twoheadrightarrow_{\beta\eta}$ (cf. 4.2). Then the mapping $n \mapsto [\overline{n}]_{=_{\beta\eta}}$ gives a bijection between the set of natural numbers and that of equivalence

[35] That is, we prove (3)′ assuming that it holds for all derivations which are properly contained in the derivation of $\Delta \vdash_{\lambda_\to} M : A$.

classes of closed[36] simple terms of type $\alpha \to (\alpha \to \alpha) \to \alpha$ modulo $=_{\beta\eta}$. Under this definition, it has been proved (cf. [39]) that $\lambda_{\to}$-representable functions are precisely the 'extended polynomials', i.e., functions obtained by composition from polynomials and the conditional function; if $x = 0$ then y else z.

A slight extension of $\lambda_{\to}$ has an interesting application. Gödel [19] gave a consistency proof of the Peano arithmetic PA by introducing a type system T for primitive recursive functionals of finite (i.e., simple) types.

5.1.4 Type system T Types of the system T are like simple types but constructed from an atomic type o (intended to be the type of natural numbers) by the functional type constructor $\to$. The T-*terms*, terms of the system T, are constructed by abstraction and application from (typed) variables and constants $\overline{0}$ (for the natural number 0), $\overline{\mathrm{suc}}$ (for the successor function), and R_A (for recursion operator) respectively of types o, $\mathrm{o} \to \mathrm{o}$, and $A \to (\mathrm{o} \to A \to A) \to \mathrm{o} \to A$ for each T-type A. The constants $\overline{0}$ and $\overline{\mathrm{suc}}$ serve to represent natural numbers as $\overline{n} \equiv \overline{\mathrm{suc}}^n(\overline{0})$. The recursion operator R_A serves to construct T-terms representing primitive recursive functions over natural numbers when A is a type of the form $\mathrm{o} \to \mathrm{o} \to \cdots \to \mathrm{o}$, and in general their higher-order counterparts, with the help of the reduction $\to_R$ which is defined schematically by

$$\begin{aligned} \cdots R_A L M \overline{0} \cdots &\to_R \cdots L \cdots, \\ \cdots R_A L M \overline{n+1} \cdots &\to_R \cdots M\overline{n}\,(R_A L M \overline{n}) \cdots \end{aligned}$$

for each natural number n and T-terms L and M of types A and $\mathrm{o} \to A \to A$, respectively. We write $\to_{\beta\eta R}$ for the union of $\to_{\beta\eta}$ and $\to_R$, and extend it to $\twoheadrightarrow_{\beta\eta R}$ and $=_{\beta\eta R}$ as before.

Then the strong normalization theorem and other fundamental theorems (w.r.t. $\to_{\beta\eta R}$) can be proved by modifying the proofs for $\lambda_{\to}$. As a consequence, besides the decidability of $=_{\beta\eta R}$, we know that the type o indeed behaves like the set of natural numbers; a closed T-term M is of the type o if and only if $M \twoheadrightarrow_{\beta\eta R} \overline{n}$ for a natural number n. When T-*representable* functions are defined as in 4.2, by replacing λ-terms with T-terms, numerals with $\overline{n} \equiv \overline{\mathrm{suc}}^n(\overline{0})$, and $\twoheadrightarrow_\beta$ with $\twoheadrightarrow_{\beta\eta R}$, they are shown to be precisely the total recursive functions whose totality can be proved in PA (cf. [18] §7.4).

The consistency proof of Peano arithmetic proceeds roughly as follows. First, we note that the proof can be reduced to that of its intuitionistic counterpart, called Heyting arithmetic (HA, for short).[37] Indeed, when we define the transformation $A \mapsto A^g$ where A^g is the formula obtained from A by replacing all atomic formulas B in A with $\neg\neg B$ and eliminating logical symbols $\vee$ and $\exists$ by means of de Morgan's laws, it satisfies

$$\Gamma \vdash_{\mathrm{NK}} A \iff \{B^g \mid B \in \Gamma\} \vdash_{\mathrm{NJ}} A^g,$$

[36] As in the (type-free) λ-calculus, the notion of free variables of typed terms is defined, and a (typed or untyped) term without free variables is said to be *closed*.

[37] Heyting arithmetic is defined by replacing the classical proof system NK in Peano arithmetic with NJ. Thus $\Gamma \vdash_{\mathrm{HA}} A \iff \Gamma \cup \mathrm{Ax(PA)} \vdash_{\mathrm{NJ}} A$, while $\Gamma \vdash_{\mathrm{PA}} A \iff \Gamma \cup \mathrm{Ax(PA)} \vdash_{\mathrm{NK}} A$.

and moreover $\vdash_{\mathrm{HA}} A^g$ holds for all axioms A of PA (cf. [48] §3.3). Thus we get $\vdash_{\mathrm{PA}} \bot \iff \vdash_{\mathrm{HA}} \bot^g (\equiv \bot)$.

The main part of the consistency proof is to define another transformation of formulas B of HA to certain meta-statements B^* on T, and verify the following: if $\vdash_{\mathrm{HA}} B$ then B^* holds. The statement B^* is a $\exists\forall$-closure of an equality (with respect to $=_{\beta\eta R}$) between T-terms, and the verification of the above property of B^* heavily relies on the strong normalizability of T-terms. For more details, see [2], [21] Ch.18, [45] Ch.III, [46].

The consistency proof cannot be carried out in the system PA, because otherwise it conflicts with the second incompleteness theorem by Gödel. In fact, when we write down the consistency proof, it requires a uniform proof of the strong normalizability of all T-terms, and this cannot be carried out in the framework of PA.

5.2 Second-order type system $\lambda 2$

Another way to make the simple type system more powerful is to introduce 'polymorphic' functions which can take types A as their arguments and return objects depending on the types, say the identity function $\lambda u : A.u$ of type $A \to A$. Then, it is reasonable to introduce abstraction of terms by type variables, such as $\lambda\alpha.(\lambda u : \alpha.u)$, and also introduce universal quantification over types, as $\forall\alpha.(\alpha \to \alpha)$. In this subsection, we discuss an extension of $\lambda_{\to}$ in this direction, which is called the *second-order* (or *polymorphic*) *type system* $\lambda 2$.

5.2.1 Definition The types of the system $\lambda 2$, called $\lambda 2$-*types*, are defined recursively, as follows.

- Type variables $\alpha_0, \alpha_1, \alpha_2, \ldots$ are $\lambda 2$-types.
- If A and B are $\lambda 2$-types, then so is $(A \to B)$.
- If A is a $\lambda 2$-type and α is a type variable, then $(\forall\alpha.A)$ is a $\lambda 2$-type.

The terms of the system $\lambda 2$, called $\lambda 2$-*terms*, are constructed by the following formation rules.[38]

$$\frac{}{\Delta \vdash u : A}\ (\text{start}) \quad \text{where} \quad (u : A) \in \Delta$$

$$\frac{\Delta, u : A \vdash M : B}{\Delta \vdash (\lambda u : A.M) : A \to B}\ (\to\text{I})$$

$$\frac{\Delta \vdash M : A \to B \quad \Delta \vdash N : A}{\Delta \vdash (MN) : B}\ (\to\text{E})$$

$$\frac{\Delta \vdash M : A}{\Delta \vdash (\lambda\alpha.M) : (\forall\alpha.A)}\ (\forall\text{I}) \quad \text{where } \alpha \notin \mathrm{FV}(\Delta)$$

$$\frac{\Delta \vdash M : \forall\alpha.A}{\Delta \vdash (MB) : A[\alpha := B]}\ (\forall\text{E})$$

[38] As before, Δ stands for contexts which assign types of this system to variables, and $\mathrm{FV}(\Delta)$ stands for the set of type variables in Δ which are free in the same sense as in logical formulas.

Thus $\lambda 2$-terms are either variables $u_0, u_1, u_2, \ldots$, abstractions $\lambda u : A.M$, applications MN, type-abstractions $\lambda\alpha.M$, or type-applications MA. For example, the $\lambda 2$-term $\lambda\alpha.\lambda u : \alpha.\lambda v : (\alpha \to \alpha).v(u)$ of the type $\forall\alpha.(\alpha \to (\alpha \to \alpha) \to \alpha)$ is constructed by applying these rules, as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta \vdash v : (\alpha \to \alpha)}\,(\text{start}) \quad \dfrac{}{\Delta \vdash u : \alpha}\,(\text{start})}{\Delta \vdash v(u) : \alpha}\,(\to\text{E})}{u : \alpha \vdash (\lambda v : (\alpha \to \alpha).v(u)) : ((\alpha \to \alpha) \to \alpha)}\,(\to\text{I})}{\vdash (\lambda u : \alpha.\lambda v : (\alpha \to \alpha).v(u)) : (\alpha \to (\alpha \to \alpha) \to \alpha)}\,(\to\text{I})}{\vdash (\lambda\alpha.\lambda u : \alpha.\lambda v : (\alpha \to \alpha).v(u)) : (\forall\alpha.(\alpha \to (\alpha \to \alpha) \to \alpha))}\,(\forall\text{I})$$

Here Δ stands for the context $\{u : \alpha, v : \alpha \to \alpha\}$.

Since we introduced type abstraction and type application to incorporate polymorphic functions into the system $\lambda 2$, it is natural to introduce a new reduction relation between $\lambda 2$-terms besides the familiar notion of β-reduction. First we extend the one-step β-reduction $\to_\beta$ to $\lambda 2$-terms as

$$\ldots(\lambda u : A.M)N\ldots \underset{\beta}{\to} \ldots M[u := N]\ldots,$$

and define a new notion of one-step β'-reduction between $\lambda 2$-terms by[39]

$$\ldots(\lambda\alpha.M)A\ldots \underset{\beta'}{\to} \ldots M[\alpha := A]\ldots.$$

We write $M \to_{\beta 2} N$ to mean $M \to_\beta N$ or $M \to_{\beta'} N$ and extend it to $\twoheadrightarrow_{\beta 2}$ and $=_{\beta 2}$ as before.

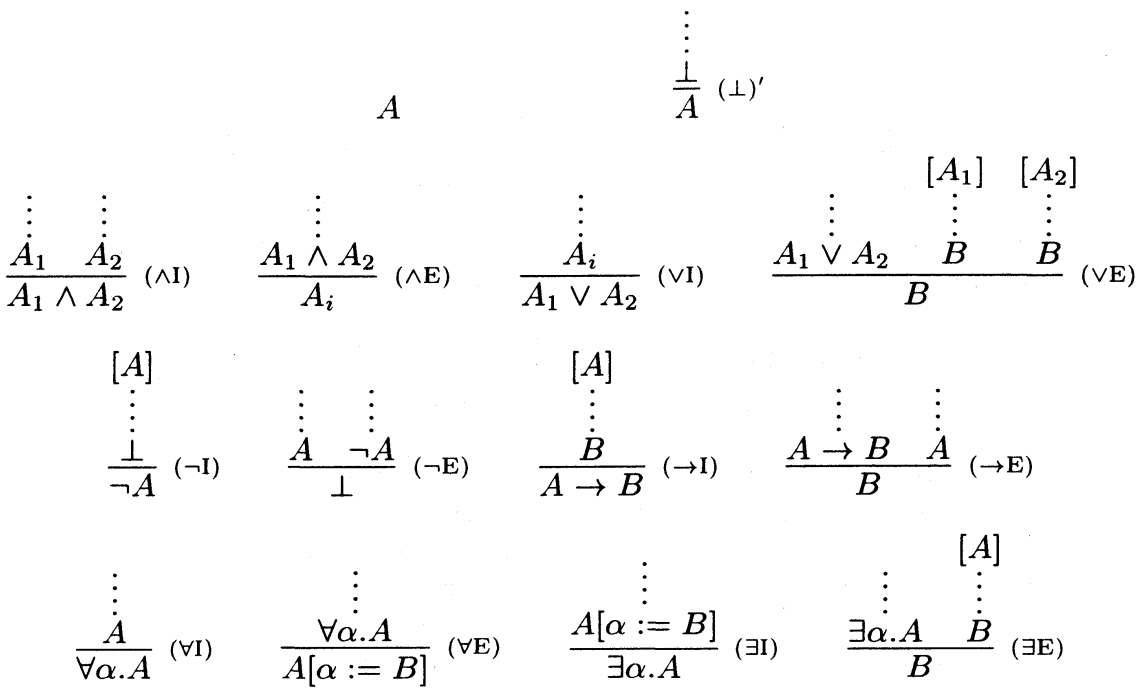
The Curry-Howard correspondence between $\lambda_\to$ and $\mathrm{NJ}_\to$ mentioned in 5.1 can be extended to the correspondence between $\lambda 2$ and a certain second-order proof system. In order to get it in perspective, let us first introduce the *second-order intuitionistic propositional calculus* PROP2.

A PROP2-*formula* is either a propositional variable[40] in $\{\alpha_0, \alpha_1, \ldots\}$, or the propositional constant $\bot$, or a compound formula of the form $A \wedge B$, $A \vee B$, $\neg A$, $A \to B$, $\forall\alpha.A$, or $\exists\alpha.A$, where A, B are PROP2-formulas, and α is a propositional variable. The system is 'propositional' because atomic formulas are only propositional variables (i.e., 0-ary predicate symbols), and it is 'second-order' because it has quantification over the second-order objects, propositions. It is 'intuitionistic' because the intuitionistic absurdity rule $(\bot)'$ is used but not the classical one.

5.2.2 Rules of PROP2 The rules of PROP2 are the same as those of NJ in 2.3, except that A, A_1, A_2, B here stand for PROP2-formulas and quantification is over propositional variables.

[39] $M[\alpha := A]$ stands for the result of substituting the type A for the free occurrences of α in M.

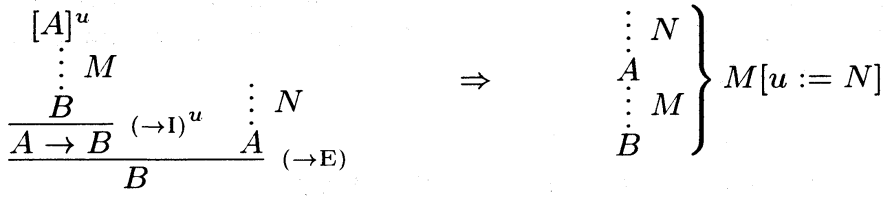
[40] We use the same symbols for propositional variables in PROP2 and type variables in $\lambda 2$, but this is just for convenience; we could use other symbols as well.



Side conditions: ($\forall$I) is applicable only when α is not free in assumptions of the derivation of A, and ($\exists$E) is applicable only when α is free neither in B nor in assumptions of the derivation of B except for A.

Let us write $\mathrm{PROP2}_{\to,\forall}$ for the $\{\to,\forall\}$-fragment of PROP2, and write $\mathrm{PROP2}_{\to}$ for the $\{\to\}$-fragment. Then the latter is equivalent to $\mathrm{NJ}_{\to}$, because both PROP2 and NJ are shown to be conservative over their own $\{\to\}$-fragments; that is, a formula of the $\{\to\}$-fragment is derivable in $\mathrm{PROP2}_{\to}$ (or in $\mathrm{NJ}_{\to}$) if and only if it is derivable by means of the rules (start), ($\to$I), and ($\to$E).

It is clear that $\lambda 2$ corresponds to $\mathrm{PROP2}_{\to,\forall}$ in the same way as $\lambda_{\to}$ corresponds to $\mathrm{NJ}_{\to}$; that is, $u_1 : A_1, \ldots, u_n : A_n \vdash_{\lambda 2} M : B$ holds for some M if and only if $A_1, \ldots, A_n \vdash_{\mathrm{PROP2}_{\to,\forall}} B$ holds, and the $\lambda 2$-term M in the former derivation in $\lambda 2$ gives a compact description of the proof tree of the latter derivation in $\mathrm{PROP2}_{\to,\forall}$. Moreover the $\beta 2$-reduction of $\lambda 2$-terms naturally induces a proof reduction to remove 'detours' in proof trees: The one-step β-reduction $(\lambda u : A.M)N \to_{\beta} M[u := N]$ for $\lambda 2$-terms induces the $\to$-reduction



of proof trees, while the one-step β'-reduction $(\lambda\alpha.M)A \to_{\beta'} M[\alpha := A]$ induces the $\forall$-reduction

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ M\\ \vdots\\ A\end{array}}{\forall\alpha.A}\,(\forall\mathrm{I})}{A[\alpha := B]}\,(\forall\mathrm{E}) \quad \Rightarrow \quad \begin{array}{c}\vdots\\ M[\alpha := B]\\ \vdots\\ A[\alpha := B]\end{array}$$

of proof trees. This correspondence is called the Curry-Howard correspondence between $\lambda 2$ and $\mathrm{PROP2}_{\to,\forall}$.

Second-order type system $\lambda 2$	Proof system $\mathrm{PROP2}_{\to,\forall}$
$\lambda 2$-type	formula
type variable	propositional variable
functional type constructor	implication
universal type constructor	universal quantifier
$\lambda 2$-term	derivation
term variable	(start) rule
abstraction	($\to$I) rule
application	($\to$E) rule
type-abstraction	($\forall$I) rule
type-application	($\forall$E) rule
$\lambda 2$-term in $\beta 2$-nf	normal derivation
$\beta 2$-reduction	proof reduction
β-reduction	$\to$-reduction
β'-reduction	$\forall$-reduction

It is interesting to note that $\mathrm{PROP2}_{\to,\forall}$ is quite expressive, so that for any PROP2-formula there exists an equivalent $\mathrm{PROP2}_{\to,\forall}$-formula. Indeed,

$$\begin{aligned}
\bot &\leftrightarrow \forall\alpha.\alpha,\\
\neg A &\leftrightarrow A \to \bot,\\
A \wedge B &\leftrightarrow \forall\alpha.((A \to B \to \alpha) \to \alpha),\\
A \vee B &\leftrightarrow \forall\alpha.((A \to \alpha) \to (B \to \alpha) \to \alpha),\\
\exists\alpha'.A &\leftrightarrow \forall\alpha.(\forall\alpha'.(A \to \alpha) \to \alpha)
\end{aligned}$$

are derivable in PROP2 where $\alpha \notin \mathrm{FV}(A) \cup \mathrm{FV}(B)$. Moreover, when we write $\bot, \neg A, A \wedge B, A \vee B, \exists\alpha.A$ in $\mathrm{PROP2}_{\to,\forall}$-formulas as abbreviations of their equivalent $\mathrm{PROP2}_{\to,\forall}$-formulas mentioned above, we can 'simulate' with the five rules (start), ($\to$I), ($\to$E), ($\forall$I) and ($\forall$E) of $\mathrm{PROP2}_{\to,\forall}$ the other inference rules of PROP2 in the sense that

($\bot$)′ $\Gamma \vdash \bot$ implies $\Gamma \vdash A$,

($\wedge$I) $\Gamma \vdash A$ and $\Gamma \vdash B$ implies $\Gamma \vdash A \wedge B$,

etc. where $\vdash$ stands for $\vdash_{\mathrm{PROP2}_{\to,\forall}}$. This means that in effect $\mathrm{PROP2}_{\to,\forall}$ has precisely the same power as PROP2. Because of this, in $\lambda 2$-types we also write

$\bot, \neg A, A \wedge B, A \vee B, \exists\alpha.A$ as abbreviations. For example, we write

$$\vdash_{\lambda 2} (\lambda u : \bot.uA) : (\bot \to A)$$

where $\bot$ is the abbreviation of the $\lambda 2$-type $\forall\alpha.\alpha$.

As before, we can prove that $\beta 2$-reduction between $\lambda 2$-terms is well-behaved; the strong normalization theorem and other fundamental theorems can be proved for $\lambda 2$, which imply that any derivation in $\mathrm{PROP2}_{\to,\forall}$ is reducible to a unique normal derivation. From this result follow the conservativity of $\mathrm{PROP2}_{\to,\forall}$ over its fragments and the consistency of PROP2. The proof of the strong normalization theorem outlined below is a slight modification of the one in [4] §4.3, which is based on [18] Ch.14.

5.2.3 Strong normalization theorem for $\lambda 2$ All $\lambda 2$-terms are strongly normalizing (w.r.t. the $\beta 2$-reduction).

In extending the proof for $\lambda_{\to}$ in 5.1.2 to $\lambda 2$, the only difficulty is how to define the set $[\![A]\!]$ for $\lambda 2$-types A with the universal type constructor $\forall$. A naive definition might be $[\![\forall\alpha.A]\!] = \bigcap_B [\![A[\alpha := B]]\!]$ where B ranges over all $\lambda 2$-types. But then the definition would get circular; for example, $[\![\forall\alpha.\alpha]\!]$ is defined based on $[\![B]\!]$ for all $\lambda 2$-types B. A way to avoid the circularity is to make the definition of $[\![A]\!]$ parameterized by type variables which are free in A.

Let us call a mapping $\rho : \{\alpha_0, \alpha_1, \ldots\} \to \mathrm{Sat}$ a *type environment* where Sat stands for the set of all saturated subsets (cf. 5.1.2) of Λ. As before, we write $|M|$ for the type-erasure of $\lambda 2$-term M, that is, the λ-term obtained from M by erasing all type information including the type-abstractors and type-arguments.[41] Then, we can verify that $|M|$ is strongly normalizing w.r.t. β-reduction if and only if so is M w.r.t. $\beta 2$-reduction. Now in order to prove the theorem for $\lambda 2$ by way of

(1) $\Delta \vdash M : A \implies |M| \in [\![A]\!] \subseteq \mathrm{SN}$,

we first define the set $[\![A]\!]\rho$ ($\subseteq \Lambda$) for $\lambda 2$-types A and type environments ρ by

- $[\![\alpha]\!]\rho = \rho(\alpha)$ if α is a type variable,
- $[\![A \to B]\!]\rho = \{M \in \Lambda \mid MN \in [\![B]\!]\rho \text{ for each } N \in [\![A]\!]\rho\}$,
- $[\![\forall\alpha.A]\!]\rho = \bigcap_{X \in \mathrm{Sat}} [\![A]\!]\rho(\alpha := X)$.

Here, $\rho(\alpha := X)$ stands for the type environment which is the same as ρ except the value for α being X. Then we define $[\![A]\!] = [\![A]\!]\rho_{\mathrm{SN}}$ where ρ_{SN} is the type environment such that $\rho_{\mathrm{SN}}(\alpha) = \mathrm{SN}$ for each α.

Under this definition, by extending the argument for $\lambda_{\to}$, we can verify

(2) $[\![A]\!]\rho$ is saturated,

[41] More precisely, $|M|$ is defined recursively as follows; $|u| \equiv u$, $|\lambda u{:}A.M| \equiv \lambda u.|M|$, $|MN| \equiv |M||N|$, $|\lambda\alpha.M| \equiv |M|$, *and* $|MA| \equiv |M|$.

hence $[\![A]\!] = [\![A]\!]\rho_{SN} \subseteq SN$ for each $\lambda 2$-type A. Next we verify

$$(3)' \quad \Delta \vdash M : A \implies \Delta \models M : A$$

where the notation $\Delta \models M : A$ now stands for

$$(3)'' \quad \forall \rho : \{\alpha_0, \alpha_1, \ldots\} \to \mathrm{Sat},\ \forall \theta : \{u_0, u_1, \ldots\} \to \Lambda,$$
$$[\ \forall (u : B) \in \Delta\ (\ \theta(u) \in [\![B]\!]\rho\ ) \implies |M|\theta \in [\![A]\!]\rho\ ].$$

Then, as a special case of (3)'' with $\rho = \rho_{SN}$ and θ being the identity, we obtain

$$(3) \quad \Delta \vdash M : A \implies |M| \in [\![A]\!],$$

hence (1) and the theorem. □

In the system $\lambda 2$, the closed $\lambda 2$-terms

$$\overline{n} \equiv \lambda\alpha.\lambda u : \alpha.\lambda v : \alpha \to \alpha.v^n(u) \quad (n = 0, 1, 2, \ldots)$$

of the type $\mathcal{N} \equiv \forall\alpha.(\alpha \to (\alpha \to \alpha) \to \alpha)$ play the role of natural numbers, and $\mathcal{N}$ stands for their totality. The strong normalization theorem guarantees that a closed $\lambda 2$-term M is of type $\mathcal{N}$ if and only if $M \twoheadrightarrow_{\beta\eta 2} \overline{n}$ for some n where $\twoheadrightarrow_{\beta\eta 2}$ is the reflexive transitive closure of the union of $\to_{\beta 2}$ and $\to_\eta$ (cf. 4.2). Moreover there is a closed $\lambda 2$-term R of type $\forall\alpha.(\alpha \to (\mathcal{N} \to \alpha \to \alpha) \to \mathcal{N} \to \alpha)$ which satisfies

$$RALM\overline{0} \twoheadrightarrow_{\beta\eta 2} L, \quad \text{and} \quad RALM\overline{n+1} \twoheadrightarrow_{\beta\eta 2} M\overline{n}\,(RALM\overline{n})$$

for any $\lambda 2$-type A, natural number n, and $\lambda 2$-terms L and M of types A and $\mathcal{N} \to A \to A$, respectively. Thus the $\lambda 2$-term RA plays the role of the recursion operator R_A in the system T (cf. 5.1.4), hence all T-representable functions are $\lambda 2$-representable.[42] It is proved by Girard [17] that $\lambda 2$-representable functions are precisely total recursive functions whose totality is provable in the second-order PA (cf. [18] Ch.15).

5.3 Higher-order type systems

The first-order proof systems discussed in section 2 are restrictive in the sense that all variables range over a single domain, hence no higher-order variables (such as function variables and predicate variables) are available. On the other hand, the proof system PROP2 in 5.2 is restrictive in the sense that it does not take into account predicates, and quantification is only over the domain of propositions. In this subsection, we first introduce a proof system called the *higher-order intuitionistic predicate calculus* PREDω, which contains in effect both NJ and PROP2.

Since the system PREDω is a predicate logic, formulas may contain terms; and since PREDω is a higher-order logic, terms may contain formulas. Therefore we

[42] We extend the notion of $\lambda{\to}$-representable functions to that of $\lambda 2$-representable functions in an obvious way.

have to define terms and formulas simultaneously.

5.3.1 Definition of PREDω[43] First, *domains* of PREDω are defined recursively, as follows.

- Prop and $X_0, X_1, X_2, \ldots$ are domains.
- If D and D' are domains, then so is $(D \to D')$.

We will call $X_0, X_1, X_2, \ldots$ *basic domains.* Note that a domain is either of the form[44] $D_1 \to D_2 \to \cdots \to D_n \to X$ or $D_1 \to D_2 \to \cdots \to D_n \to$ Prop where $n \geq 0$, $D_1, \cdots, D_n$ are arbitrary domains of PREDω, and X is a basic domain. We will call domains in the first form *functional domains* and those in the second form *predicate domains.*

Next, the PREDω*-terms*, which include both terms and formulas in the ordinary sense, are defined together with their domains recursively, as follows.

1. For each domain D, variables $x_0^D, x_1^D, \ldots$ are terms of the domain D.
2. If M and N are terms of the domains $D \to D'$ and D respectively, then (MN) is a term of the domain D'.
3. If x is a variable of the domain D and M is a term of the domain D', then $(\lambda x : D.M)$ is a term of the domain $D \to D'$.
4. If A and B are terms of the domain Prop, then so is $(A \to B)$.
5. If x is a variable of the domain D and A is a term of the domain Prop, then $(\forall x : D.A)$ is a term of the domain Prop.

PREDω-terms of the domain Prop are called PREDω*-formulas.* The clause 1 says that we have function variables (ranging over functional domains) and predicate variables (ranging over predicate domains), among which individual variables (ranging over basic domains) and propositional variables (ranging over Prop) are included. The clause 3 says functions and predicates can be constructed within the system by abstracting terms, which may be used as functions or predicates as well as their arguments.

The derivations in PREDω are constructed by using the following six rules.

$$\begin{array}{ccccc}
 & [A] & & & \\
 & \vdots & & \vdots \quad \vdots & \\
A & \dfrac{B}{A \to B}\ (\to\text{I}) & & \dfrac{A \to B \quad A}{B}\ (\to\text{E}) & \\
\vdots & & \vdots & & \vdots \\
\dfrac{A}{\forall x : D.A}\ (\forall\text{I}) & & \dfrac{\forall x : D.A}{A[x := M]}\ (\forall\text{E}) & & \dfrac{A}{B}\ (=_\beta)
\end{array}$$

[43] Our PREDω is slightly different from PREDω in the literature (e.g., [4]), but the type system CC introduced later based on PREDω is the same as CC in the literature.

[44] As before, we omit parentheses in $D_1 \to (D_2 \to (D_3 \to \cdots (D_n \to D_{n+1}) \cdots))$ and write simply $D_1 \to D_2 \to \cdots \to D_{n+1}$.

The first five rules are the same as before except that A, B and $\forall x : D.A$ are PREDω-formulas and M in ($\forall$E) is a PREDω-term of the domain D. As before, we impose the side condition on ($\forall$I) that x cannot be free in the assumptions. The last rule ($=_\beta$) is applicable only when A and B are formulas such that $A =_\beta B$ where $=_\beta$ is defined as before.[45]

By definition, PREDω-formulas contain as logical symbols only the implication $\to$ and the universal quantifier $\forall$. But we can introduce other logical symbols and the logical constant $\bot$ as abbreviations as in the case of PROP2$_{\to,\forall}$ in 5.2; namely, we write

$$\begin{array}{rll} \bot & \text{for} & \forall\alpha : \mathsf{Prop}.\,\alpha, \\ \neg A & \text{for} & A \to \bot, \\ A \wedge B & \text{for} & \forall\alpha : \mathsf{Prop}.\,((A \to B \to \alpha) \to \alpha), \\ A \vee B & \text{for} & \forall\alpha : \mathsf{Prop}.\,((A \to \alpha) \to (B \to \alpha) \to \alpha), \\ \exists x : D.A & \text{for} & \forall\alpha : \mathsf{Prop}.\,(\forall x : D.(A \to \alpha) \to \alpha) \end{array}$$

where α is a propositional variable which is free neither in A nor in B.

Then as in the case of PROP2$_{\to,\forall}$, appropriate rules for $\bot, \neg, \wedge, \vee, \exists$ are shown to be derivable in PREDω. Moreover, when we define for each domain D the 'Leibniz equality' $=_D$ (of the domain $D \to D \to \mathsf{Prop}$) by

$$M \underset{D}{=} N \iff_{\mathrm{def}} \forall L : D \to \mathsf{Prop}.(LM \to LN),$$

the equality axioms (E1) and (E2) in 2.2 are easily derivable in PREDω. Note that we can obtain a classical counterpart of PREDω by adding the formula $\forall\alpha : \mathsf{Prop}.(\alpha \vee \neg\alpha)$ (or equivalently $\forall\alpha : \mathsf{Prop}.(\neg\neg\alpha \to \alpha)$) to PRED$\omega$ as an axiom.

Next we will extend the proof system PREDω to a type system λPREDω. In λPREDω, derivations in PREDω are described by typed terms as in $\lambda_\to$ and $\lambda 2$, but moreover the language part of the system PREDω can also be described in the typing mechanism, by writing

$$\begin{array}{rll} D : \mathsf{fd} & \text{for} & \text{“}D \text{ is a functional domain”}, \\ D : \mathsf{pd} & \text{for} & \text{“}D \text{ is a predicate domain”}, \\ M : D & \text{for} & \text{“}M \text{ is a term of the domain } D\text{”}. \end{array}$$

The rules of λPREDω are classified into four groups. The first group consists of the domain-formation rules, which are of the form $\Delta \vdash D : s$ where $s \in \{\mathsf{fd}, \mathsf{pd}\}$, meaning that D is a domain of the sort s under the context Δ. The second group consists of the term-formation rules, which are of the form $\Delta \vdash M : D$ where $\Delta \vdash D : s$ for some s, and this means M is a legal term of the domain D under the context Δ. Third, the inference rules are of the form $\Delta \vdash P : A$ where $\Delta \vdash A : \mathsf{Prop}$, and this means A is a formula which is derivable in PREDω under

[45] Namely, the one-step β-reduction relation $\to_\beta$ is extended to PREDω-terms in a natural way, and $=_\beta$ is the equivalence relation generated by $\to_\beta$.

the context Δ, and its derivation is described by P. The last group consists of the weakening rules, which are necessary for bookkeeping, due to the fact that the contexts of this system are not mere sets but sequences (of language definitions and labelled assumptions). Throughout the rules, s and s' range over the set $\{\mathsf{pd}, \mathsf{fd}\}$, and $\mathrm{FV}(\Delta)$ stands for the set of free variables of either terms or types in Δ.

5.3.2 Rules of λPREDω

1. Domain-formation rules

$$\frac{}{\Delta, X : \mathsf{fd} \vdash X : \mathsf{fd}} \text{ (d1)} \quad \text{where } X \notin \mathrm{FV}(\Delta)$$

$$\frac{}{\Delta \vdash \mathsf{Prop} : \mathsf{pd}} \text{ (d2)}$$

$$\frac{\Delta \vdash D : s \quad \Delta \vdash D' : s'}{\Delta \vdash (D \to D') : s'} \text{ (d3)}$$

2. Term-formation rules

$$\frac{\Delta \vdash D : s}{\Delta, x : D \vdash x : D} \text{ (t1)} \quad \text{where } x \notin \mathrm{FV}(\Delta)$$

$$\frac{\Delta \vdash M : (D \to D') \quad \Delta \vdash N : D}{\Delta \vdash MN : D'} \text{ (t2)}$$

$$\frac{\Delta, x : D \vdash M : D' \quad \Delta \vdash (D \to D') : s}{\Delta \vdash (\lambda x : D.M) : (D \to D')} \text{ (t3)}$$

$$\frac{\Delta \vdash A : \mathsf{Prop} \quad \Delta \vdash B : \mathsf{Prop}}{\Delta \vdash (A \to B) : \mathsf{Prop}} \text{ (t4)}$$

$$\frac{\Delta \vdash D : s \quad \Delta, x : D \vdash A : \mathsf{Prop}}{\Delta \vdash (\forall x : D.A) : \mathsf{Prop}} \text{ (t5)}$$

3. Inference rules for derivations

$$\frac{\Delta \vdash A : \mathsf{Prop}}{\Delta, u : A \vdash u : A} \text{ (start)} \quad \text{where } u \notin \mathrm{FV}(\Delta)$$

$$\frac{\Delta, u : A \vdash P : B \quad \Delta \vdash (A \to B) : \mathsf{Prop}}{\Delta \vdash (\lambda u : A.P) : (A \to B)} \text{ ($\to$I)}$$

$$\frac{\Delta \vdash P : (A \to B) \quad \Delta \vdash Q : A}{\Delta \vdash PQ : B} \text{ ($\to$E)}$$

$$\frac{\Delta, x : D \vdash P : A \quad \Delta \vdash (\forall x : D.A) : \mathsf{Prop}}{\Delta \vdash (\lambda x : D.P) : (\forall x : D.A)} \text{ ($\forall$I)}$$

$$\frac{\Delta \vdash P : (\forall x : D.A) \quad \Delta \vdash M : D}{\Delta \vdash PM : A[x := M]} \text{ ($\forall$E)}$$

$$\frac{\Delta \vdash P : A \quad \Delta \vdash B : \mathsf{Prop}}{\Delta \vdash P : B} \text{ ($=_\beta$)} \quad \text{where } A =_\beta B$$

4. Weakening rules[46]

$$\frac{\Delta \vdash U : V}{\Delta, X : \mathsf{fd} \vdash U : V}\ (\mathrm{w1}) \quad \text{where } X \notin \mathrm{FV}(\Delta)$$

$$\frac{\Delta \vdash D : s \quad \Delta \vdash U : V}{\Delta, x : D \vdash U : V}\ (\mathrm{w2}) \quad \text{where } x \notin \mathrm{FV}(\Delta)$$

$$\frac{\Delta \vdash A : \mathsf{Prop} \quad \Delta \vdash U : V}{\Delta, u : A \vdash U : V}\ (\mathrm{w3}) \quad \text{where } u \notin \mathrm{FV}(\Delta)$$

There are a number of possible ways to improve the system $\lambda\mathrm{PRED}\omega$. First, in view of similarities between the rules, we can unify some of them. For example, the three rules (d1), (t1) and (start) for introducing new variables are similar, and they can be unified. Likewise, the three weakening rules may be unified, and so forth. More significantly, when we rewrite implicational formulas $A \to B$ by $\forall u : A.B$ where $u \notin \mathrm{FV}(B)$, the rules for implication can be merged to those for the universal quantifier.

Another possibility is to increase the expressive power of the system. We can introduce the direct product $\Pi x : D.D'$ of functional domains where D' possibly depends on x. To do so, we have to introduce a new notion of 'parameterized functional domains' of type $D_1 \to \cdots \to D_n \to \mathsf{fd}$ in the same way as predicates are considered as 'parameterized propositions' of type $D_1 \to \cdots \to D_n \to \mathsf{Prop}$. But how can we do this without making the system too big?

There is a clever way to reconcile the two ideas; to simplify the system and to increase the expressive power. The point is to take advantage of the similarity between the notions of direct products $\Pi x : D.D'$ and universal formulas $\forall x : D.A$. We can introduce the notion of direct products of functional domains $\Pi x : D.D'$ in the same way as universal formulas $\forall x : D.A$ are introduced, so that the two notions can share the same rules. In addition, once the product $\Pi x : D.D'$ is introduced, by rewriting the functional domain construct $D \to D'$ as $\Pi x : D.D'$ where $x \notin \mathrm{FV}(D')$, we can still further economize the number of rules.

By combining all these ideas, we can obtain the type system called the *Calculus of Constructions* (CC, for short). In CC, Prop and fd in $\lambda\mathrm{PRED}\omega$ are merged into a constant $*$, while pd and another new sort necessary to incorporate the direct product construct are merged into another constant $\Box$.

5.3.3 Rules of CC In the rules below, s and s' range over the set of constants $\{*, \Box\}$, and U, V, V', W, W' range over legal expressions of CC. The symbol Π stands for both the universal quantifier and the direct product constructor.

$$\frac{}{\vdash * : \Box}\ (\text{axiom})$$

$$\frac{\Delta \vdash U : s}{\Delta, x : U \vdash x : U}\ (\text{var}) \quad \text{where } x \notin \mathrm{FV}(\Delta)$$

[46] $\Delta \vdash U : V$ stands for any legal sequent of the system.

$$\frac{\Delta \vdash U : s \quad \Delta, x : U \vdash W : s'}{\Delta \vdash (\Pi x : U.W) : s'} \ (\Pi)$$

$$\frac{\Delta, x : U \vdash V : W \quad \Delta \vdash (\Pi x : U.W) : s}{\Delta \vdash (\lambda x : U.V) : (\Pi x : U.W)} \ (\lambda)$$

$$\frac{\Delta \vdash V : (\Pi x : U.W) \quad \Delta \vdash V' : U}{\Delta \vdash VV' : W[x := V']} \ (\text{app})$$

$$\frac{\Delta \vdash V : W \quad \Delta \vdash W' : s}{\Delta \vdash V : W'} \ (=_\beta) \quad \text{where } W =_\beta W'$$

$$\frac{\Delta \vdash U : s \quad \Delta \vdash V : W}{\Delta, x : U \vdash V : W} \ (\text{weak})$$

It is interesting to note that the simple type system $\lambda_{\to}$ in 5.1 can be identified with the subsystem of CC in which both s and s' in the rule (Π) are fixed to $*$. Likewise the system $\lambda 2$ in 5.2 can be identified with the subsystem of CC in which the s' in (Π) is fixed to $*$, while the s ranges over $\{*, \Box\}$. On the other hand, when we fix the s in the rule (Π) to be $*$, we obtain the type system corresponding to the first-order predicate logic with product domains.

The type system CC was introduced by Th. Coquand [9], and he proved mathematical properties of the system, including the strong normalization theorem. Since $\lambda\text{PRED}\omega$ is mapped to CC by a renaming of constants, the theorem immediately implies the strong normalization theorem for $\lambda\text{PRED}\omega$ and $\text{PRED}\omega$, which in turn implies the strong normalization theorem for NJ because NJ can be embedded in $\text{PRED}\omega$. Conservativity results among subsystems of CC have been studied in [16]. In particular, CC is conservative over $\lambda 2$, while $\lambda 2$ is conservative over $\lambda_{\to}$. However the system CC is not conservative over $\text{PRED}\omega$, in the sense that there exist a $\text{PRED}\omega$-formula A and a context Δ of $\lambda\text{PRED}\omega$ such that A is not derivable in $\lambda\text{PRED}\omega$ under Δ (that is, $\Delta \vdash_{\lambda\text{PRED}\omega} A : \mathsf{Prop}$ and $\Delta \not\vdash_{\lambda\text{PRED}\omega} P : A$ for any P) but the translation A' of A is derivable in CC under the translation Δ' of the context Δ (that is, $\Delta' \vdash_{\text{CC}} P : A'$ for some P). This means that the conversion of $\lambda\text{PRED}\omega$ to CC has some side effects. But it is not a destructive one; indeed, the strong normalization theorem for CC guarantees the consistency of the system in the sense that $\not\vdash_{\text{CC}} P : \bot$ for any P where $\bot$ stands for $\Pi\alpha : *.\alpha$. Extensions of CC with certain axioms such as $\Pi\alpha : *.(\alpha \vee \neg\alpha)$ are also shown to be consistent.

On the other hand, it is known (as Girard's paradox) that a unifying process of notions in a type system could be destructive; indeed, if we unify the two constants $*$ and $\Box$ of CC, the resulting system becomes inconsistent (i.e., $\Delta \vdash P : \bot$ for some P). For more details on CC and related topics including Girard's paradox, see [9], [10], [4] §5, [16], among which [4] is the standard text of the subject area of this section.

It is possible to extend CC further, for example, by adding the direct sum construct $\Sigma x : D.D'$. The type system ECC, the Extended Calculus of Constructions, by Z.Luo [27] is such an extension, which is very interesting mathematically and

also from the viewpoint of computer science. In fact, an interactive proof development system (proof checker), called Lego, has been implemented based on ECC by R.Pollack [28]. The implementation enhanced the use and understanding of type theory among people who are interested in logical aspects of computer science, and it still does.

For more on the subject of this section, see other articles in this volume and [4], [20], [23], [24], [31], [33], [34], [40]. Also good guides for relating subjects will be found in [1], [5], [26].

I am very grateful to people who encouraged and helped me to write this article. In particular, discussions with Roger Hindley were very useful. He also read a preliminary version of the draft and made detailed comments to improve my English exposition.

References

[1] S. Abramsky et al. (eds.) (1992,1994,1995). *Handbook of Logic in Computer Science*, Vol.1-4, Oxford University Press.

[2] J. Avigad and S. Feferman (1998). Gödel's functional ("Dialectica") interpretation, in [5] 338-405.

[3] H. Barendregt (1984). *The Lambda Calculus*, 2nd edition, North-Holland.

[4] H. Barendregt (1992). Lambda calculi with types, in [1] Vol.2, 117-309.

[5] S. R. Buss (ed.) (1998) *Handbook of Proof Theory*, Elsevier Science B.V.

[6] A. Church (1932). A set of postulates for the foundation of logic. Annals of Math. 33, 346-366.

[7] A. Church (1936). An unsolvable problem of elementary number theory. American J. Math. 58, 345-363. (Also in [12] 88-107.)

[8] A. Church (1940). A formalization of the simple theory of types. J. Symbolic Logic 5, 56-68.

[9] Th. Coquand (1985). *Une théorie des constructions*, Thèse de troisième cycle, Université Paris VII.

[10] Th. Coquand and G. Huet (1988). The calculus of constructions. *Information and Computation*, 76, 95-120.

[11] H. B. Curry and R. Feys (1958). *Combinatory Logic*, Vol.I, North-Holland.

[12] M. Davis (ed.) (1965). *The Undecidable*, Raven Press.

[13] J. W. Dawson (1997). *Logical Dilemmas, the Life and Work of Kurt Gödel*, AK Peters.

[14] S. Feferman et al. (eds.) (1986, 1990). *Kurt Gödel Collected Works*, Vol.I-II, Oxford University Press.

[15] G. Gentzen (1935). Untersuchungen über das logische Schliessen I,II, Mathematische Zeitschrift 39, 176-21, 405-431. (English translation in M. E. Szabo (ed.) (1969). *The Collected Papers of Gerhard Gentzen*, 68-131. North-Holland.)

[16] J. H. Geuvers (1993). *Logics and type systems*, Ph.D thesis, University of Nijmegen.

[17] J. -Y. Girard (1972). *Interprétation fonctionelle et elimination des coupures de l'arithmétique d'ordre supérieur*, Ph.D thesis, Université Paris VII.

[18] J. -Y. Girard, Y. Lafont and P. Taylor (1988). *Proofs and Types*, Cambridge University Press.

[19] K. Gödel (1958). Über eine bisher noch night benützte Erweiterung des finiten Standpunktes. Dialectica 12, 280-287. (English translation in J. Philos. Logic 9 (1980) 133-142 and in [14] Vol.II, 240-251.)

[20] J. R. Hindley and J. P. Seldin (eds.) (1980). *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press.

[21] J. R. Hindley and J. P. Seldin (1986). *Introduction to Combinators and λ-Calculus*, Cambridge University Press.

[22] W. A. Howard (1980). The formulae-as-types notion of construction, in [20] 479-490,

[23] G. Huet (ed.) (1990). *Logical Foundations of Functional Programming*, Addison Wesley.

[24] G. Huet and G. Plotkin (eds.) (1991). *Logical Frameworks*, Cambridge University Press.

[25] S. C. Kleene (1936). General recursive functions of natural numbers, Mathematische Annalen 112, 727-742. (Also in [12] 236-253.)

[26] J. van Leeuwen (ed.) (1990). *Handbook of Theoretical Computer Science*, Vol.A-B, Elsevier.

[27] Z. Luo (1994). *Computation and Reasoning*, Oxford University Press.

[28] Z. Luo and R. Pollack (1992). *LEGO Proof Development System: User's Manual*, LFCS Report ECS-LCFS-92-211, Department of Computer Science, University of Edinburgh.

[29] P. Martin-Löf (1984). *Intuitionistic Type Theory*, Bibliopolis.

[30] E. Mendelson (1997). *Introduction to Mathematical Logic*, fourth edicion, Chapman & Hall.

[31] R. P. Nederpert, J. H. Geuvers, and R. C. de Vrijer (eds.) (1994). *Selected Papers on Automath*, North-Holland.

[32] P. Odifreddi (1989). *Classical Recursion Theory*, North-Holland.

[33] P. Odifreddi (ed.) (1990). *Logic and Computer Science*, Academic Press.

[34] A. Pitts and P. Dybjer (eds.) (1997). *Semantics and Logics of Computation*, Cambridge University Press.

[35] D. Prawitz (1965). *Natural Deduction*, Almqvist and Wiksell.

[36] J. B. Rosser (1982). Highlights of the history of the lambda-calculus, in *Proceedings of Symp. on LISP and Functional Programming*, 216-225, ACM.

[37] B. Russell (1903). *The Principles of Mathematics*, Allen and Unwin. (1992 edition: Routledge and Kegan Paul.)

[38] J. R. Shoenfield (1967). *Mathematical Logic*, Addison Wesley.

[39] H. Schwichtenberg (1976). Definierbare Funktionen im λ-Kalkül mit Typen. Archiv Math. Logik 17, 113-114.

[40] H. Schwichtenberg (ed.) (1995). *Proof and Computation*, NATO ASI Series, Springer.

[41] D. S. Scott (1972). Continuous lattices, Lecture Notes in Math. Vol.274, 97-136, Springer.

[42] S. Stenlund (1972). *Combinators, λ-terms and Proof Theory*, D.Reidel.

[43] M. Takahashi (1991). *Theory of Computation – Computability and λ-Calculus* (in Japanese), Kindai Kagakusha.

[44] M. Takahashi (1995). Parallel reductions in λ-calculus, *Information and Computation*, Vol.118, 120-127.

[45] A. S. Troelstra (ed.) (1973). *Mathematical Investigations of Intuitionistic Arithmetic and Analysis*, Lecture Notes in Math. Vol.344, Springer.

[46] A. S. Troelstra (1990). Introductory note to 1958 and 1972, in [14] Vol.II, 217-241.

[47] A. S. Troelstra and H. Schwichtenberg (1996). *Basic Proof Theory*, Cambridge University Press.

[48] A. S. Troelstra and D. van Dalen (1988). *Constructivism in Mathematics*, North-Holland.

[49] A. Turing (1936). On computable numbers, with an application to the entscheidungsproblem, in Proceedings of the London Math. Society Ser.2, 42, 230-265. Corrections, Ibid, 43, 544-546. (Also in [12] 115-154.)

[50] D. van Dalen (1994). *Logic and Structure*, 3rd edition, Springer-Verlag.

[51] H. Wang (1987). *Reflections on Kurt Gödel*, MIT Press.