Notations

We use the following notations

$$b\ a_1\ \ \dots\ \ a_n \ \text{ for } \ (\dots((b\ a_1)\ a_2)\dots)\ a_n$$

For instance $b\ a_1\ a_2$ for $(b\ a_1)\ a_2$

This is *Schönfinkel's notation* (1920)

$$\lambda x_1 \dots x_n.e \ \text{ for } \ \lambda x_1 \lambda x_2 \dots \lambda x_n.e$$

λ-notation is not common in mathematics but quite useful

$f : x \longmapsto e$ for $f = \lambda x.e$

λ-terms are used both for *terms* and *proofs* (cf. Church's higher-order logic)

Brouwer-Heyting-Kolmogorov's interpretation

Brouwer stressed the fact that there are non constructive proofs in mathematics (they appeared only in the late 19th century)

For instance

$$(\forall f \in \mathbb{N} \to \mathbb{N})(\exists n \in \mathbb{N})(\forall m \in \mathbb{N})\ f(n) \leq f(m)$$

We can *prove* this only in a non constructive way

The proof does not suggest (a priori) any way to compute n from f

Brouwer rejected the Law of Excluded Middle $A \vee \neg A$ and designed a new logic (1908)

General opinion in 1930: classical methods are needed, they are abstract and elegant, constructive proofs too restrictive

This has changed after Bishop's work (1966)

Kolmogorov's Calculus of Problems

Calculus of problem, for instance geometrical problems

$A \to B$ is read as: given a solution to the problem A solve problem B

If we have a solution to $A \to B$ and $B \to C$ we have a solution to $A \to C$

Kolmogorov observed the following remarkable fact (1933)

The calculus of problems coincides in form with the Brouwerian logic

Kolmogorov's Calculus of Problems

$A \wedge B$ denotes the problem "solve both problems A and B"

$A \vee B$ denotes the problem "solve at least one of the problems A or B"

$A \to B$ denotes the problem "reduce the problem B to the problem A"

$\neg A$ denotes the problem "assuming a solution to A derive a contradiction"

Write $\vdash A$ for "we can give a solution to the problem A"

We can then justify, uniformely in A, B

$$\vdash A \wedge B \to A \qquad \vdash A \to (B \to A)$$

Kolmogorov's Calculus of Problems

Notice that the problem

$$A \vee \neg A$$

has no uniform solution!!

The calculus of problems coincides in form with the Brouwerian logic

However we have

$$\vdash \neg\neg(A \vee \neg A)$$

Curry's remark

Haskell Curry was working in combinatory logic

$$S\ x\ y\ z = (x\ y)\ x\ z \qquad\qquad K\ x\ y = y$$

He noticed that the types of these combinators look like two axioms of propositional logic (Hilbert's system)

$$S \in (A \to B \to C) \to (A \to B) \to A \to C$$

$$K \in A \to B \to A$$

Here we write

$$A_1 \to \ldots \to A_n \ \text{ for } \ A_1 \to (A_2 \to (\ldots (A_{n-1} \to A_n) \ldots))$$

Curry-Howard Analogy

This suggests the following multiple reading of $a \in A$

- A is a set and a is an element of A
- A is a proposition and a is a proof of the proposition A (constructive logic)
- A is a problem (task) and a is a method of solving the problem (task) A
- A is a specification and a is a program which meets the specification A

Minimal Propositional Calculus

$A \to B$ set of functions from the set A to the set B

We can build uniformely elements in the sets

$$A \to A \qquad A \to B \to A$$

For instance $\lambda x \lambda y.x \in A \to B \to A$

The following map is essential in mathematics

$$\lambda x \lambda f.f\ x \in A \to ((A \to B) \to B)$$

We get in this way the *minimal propositional calculus*

Minimal Propositional Calculus

The formula $((A \to B) \to A) \to A$ is classically valid. However it has no uniform elements

$$\lambda f.f\ (\lambda x.??) \in ((A \to B) \to A) \to A$$

We can use the λ-calculus to show the decidability of the inhabitation problem

"is the proposition A provable"

for the minimal propositional calculus: it is enough to look for terms in *normal form*

One can show that this problem is P-space complete (hence likely to be more complex than classical propositional calculus)

Negation

The case of negation is extremely interesting

We introduce $\perp$ the "absurd proposition"

It can be read as the *empty set*, or as a problem with no solution

In set theory we have a *unique* element in $\perp \rightarrow A$

"Ex Falso Quodlibet" rule in logic

Then we can define $\neg A$ as $A \rightarrow \perp$

Double Negation Law

What does $\neg(\neg A) \to A$ means set theoretically??

It is exactly a uniform *choice function*, cf. Hilbert's ϵ-operator

$$\epsilon_X \in ((X \to \perp) \to \perp) \to X$$

since $(X \to \perp) \to \perp$ is empty iff X is empty!!

What is $\perp$?

A suggestive interpretation of $\perp$ is the type used for exceptions

One can even give plausible interpretations of $\neg\, A$ as the type of *continuations* over A and possible interpretations of

$$A \vee \neg\, A$$

This corresponds to "impure" features

See the papers of Jean-Louis Krivine

For instance "About classical logic and imperative programming"

Extending the Curry-Howard Analogy

$A \times B$ cartesian product corresponds to $A \wedge B$

$A + B$ disjoint sum corresponds to $A \vee B$

We get *intuitionistic logic*

$A + B \to C$ and $(A \to C) \times (B \to C)$ are isomorphic

In general $A \times B \to C$ and $(A \to C) + (B \to C)$ are not isomorphic

Extending the Curry-Howard Analogy

$(a, b) \in A \times B$ for $a \in A,\ b \in B$

$p(c) \in A,\ q(c) \in B$ if $c \in A \times B$

$i(a) \in A + B,\ j(b) \in A + B$ if $a \in A$ or $b \in B$

$E(d, u, v) \in C$ if $d \in A + B,\ u \in A \to C,\ v \in B \to C$

$$p((a, b)) = a \qquad q((a, b)) = b$$

$$E(i(a), u, v) = u\ a \qquad E(j(b), u, v) = v\ b$$

We need an *extension* of pure λ-calculus

Extending the Curry-Howard Analogy

It is remarkable that this analogy extends to quantifiers

In set theory, given a family of sets X_i over an index set I we can define

- the *dependent product* $\Pi_{i \in I} X_i$ which is the set of all functions $f \in I \to \bigcup_{i \in I} X_i$ such that $f\ i \in X_i$ for all $i \in I$
- the *dependent sum* $\Sigma_{i \in I} X_i$ which is the union $\bigcup_{i \in I} \{i\} \times X_i$ that is the set of pairs (i, x) with $x \in X_i$

In this way, we can interpret (intuitionistic) first-order logic set-theoretically (Howard, 1970)

Dependent Product

We can read $\Pi_{i \in I} X_i$ as $(\forall i \in I)\ X_i$

X_i is a proposition depending on $i \in I$

A proof of $(\forall i \in I)\ X_i$ is a method f which given an element $i \in I$ gives a proof $f\ i \in X_i$

If $t_i \in X_i\ (i \in I)$ we have $\lambda i.t_i \in (\forall i \in I)\ X_i$

If $f \in (\forall i \in I)\ X_i$ and $i \in I$ we have $f\ i \in X_i$

$(\Pi i \in I) X_i$ is $I \rightarrow J$ if $X_i = J$ for all $i \in I$

Dependent Sum

We can read $\Sigma_{i \in I} X_i$ as $(\exists i \in I) X_i$

X_i is a proposition depending on $i \in I$

A proof of $(\exists i \in I)\ X_i$ is a pair (i, x) with $i \in I$ and $x \in X_i$

If $i \in I$ and $x \in X_i$ we have $(i, x) \in (\exists i \in I)\ X_i$

If $c \in (\exists i \in I)\ X_i$ then $p(c) \in I$ and $q(c) \in X_{p(c)}$

$(\Sigma i \in I) X_i$ is $I \times J$ if $X_i = J$ for all $i \in I$

Strong Existential

This is not quite the usual elimination rule

$$((\forall i \in I).X_i \to Y) \quad \to ((\exists i \in I)\ X_i) \to Y$$

This is provable and a proof is

$$\lambda f \lambda c. f\ p(c)\ q(c)$$

We have a new existential quantifier!

The Axiom of Choice

The following form of the axiom of choice is provable

$$(\forall i \in I)(\exists j \in J)\ X_{i,j} \to (\exists f \in I \to J)(\forall i \in I)\ X_{i,f\ i}$$

A proof is

$$\lambda h.(\lambda i.p(h\ i), \lambda i.q(h\ i))$$

If $X_{i,j} = K$ we have a proof of

$$(I \to J \times K) \to (I \to J) \times (I \to K)$$

Notations for Proofs

A proof of

$$((\forall i \in I).A \to B_i) \to A \to (\forall i \in I)\ B_i$$

is

$$\lambda f \lambda x \lambda i.f\ \ i\ x$$

This is called *domain-free notation*

cf. G. Barthe, M. Sorensen "Domain-free pure type systems" J. Funct. Programming 10 (2000)

This style is intermediate between Curry style and Church style

Notations for Proofs

Curry style

$$\vdash \lambda x.x \in (\forall n \in \mathbb{N}).A(n) \to A(n)$$

Church style

$$\vdash (\lambda n \in \mathbb{N})(\lambda x \in A(n)).x \in (\forall n \in \mathbb{N}).A(n) \to A(n)$$

Domain-free style

$$\vdash \lambda n\ x.x \in (\forall n \in \mathbb{N}).A(n) \to A(n)$$

Notation for Proofs

The difference between these notations are relevant for decidability issues

Inhabitation problem: is A inhabited?? Clearly not decidable

Given A and a do we have $a \in A$? Only decidable in Church and domain-free style (for a in normal form)

Given a is there A such that $a \in A$? Only decidable in Church's style

We work mainly in *domain-free* style

In *extensional type theory* (such as NuPrl) the relation $a \in A$ is *not* decidable

Some differences: set theory/type theory

In set theory we can have $A \ni A_1 \ni A_2 \ni \dots$

In type theory we have only $A \ni a$

In type theory, we have a *definitional equality* or convertibility $t = u$ which is decidable

In set theory, the equality is extensional equality (same elements) which is not decidable

In classical set theory we can prove $(\exists x \in A)P(x)$ without giving a way to compute x

Notation for Proofs

In the calculus we shall consider the statement

$$a \in A$$

is decidable

This is a quite important point: *program verification* is *undecidable*, but *proof-checking* = type-checking is *decidable*

This is the basis for proof-carrying code for instance

Notation for Proofs

Around the same time (1968), de Bruijn designed the language *Automath*

Important contributions are the notions of *contexts*

$$x_1 \in A_1, x_2 \in A_2(x_1), \ldots, x_n \in A_n(x_1, \ldots, x_{n-1})$$

which was only implicit in Martin-Löf's early papers

This lists the assumptions *and* variables that are alive at some points in the proofs

This is well expressed by the *flag notation*, cf. Jutting's example

In usual presentations, the context list only the current assumptions

Notation for Proofs

This notion reflects well the practice in mathematics

"Let $n \in \mathbb{N}$ and $x \in \mathbb{R}^n$, assume that $n > 0$ and that $x_0 \neq 0$ then ..."

is represented by the context

$$n \in \mathbb{N}, x \in \mathbb{R}^n, h_1 \in (n > 0), h_2 \in (x_0 \neq 0)$$

If we want to *instantiate* this context, we need to provide

1. an element $p \in \mathbb{N}$
2. an element $u \in \mathbb{R}^p$
3. a proof that we have $p > 0$
4. a proof that we have $u_0 \neq 0$

Notation for Proofs

The same idea is potentially important for representing *comments* for programs: if we have

$$f \in (\Pi x \in A).P(x) \to B$$

then in ordre to get a program of type B from f we have to give both an element $a \in A$ *and* a proof p that the proposition $P(a)$ holds

We can only use f in a "safe" way

According to Automath people (de Bruijn and Jutting) this is close to informal practice of mathematicians: "if we define a function f for x real and $x > 1$ then the use of the value of the function at a point requires (i) that point (a real number), (ii) a proof that the real number is > 1"

Proof Notation for First-Order Logic

As an application of proof notations, we show, in a proof theoretic way, that the following first-order theory

$$(\forall x)\ R(x, f(x)) \qquad (\forall x) \neg\ R(x, x)$$

$$(\forall x, y, z)\ R(x, y) \to R(y, z) \to R(x, z)$$

is *consistent*

Notice that this theory has no *finite models*

Is it possible to prove its consistency without involving actual infinity??

We show then in a weak sense that "there is an infinite set" (Hilbert's notion of consistency = no contradiction)

Minimal Logic

Minimal logic: we have only the connective $\rightarrow$ and the quantifier $\forall$, and we have a special proposition X, which plays the role of $\perp$ but without the rule $\perp\rightarrow A$

We show how to reduce, in a syntactical way, classical first-order logic to minimal logic

This is known as Gödel-Gentzen negative translation

For instance

$$[A \vee B] = ([A] \rightarrow X) \rightarrow ([B] \rightarrow X) \rightarrow X$$

$$[A \wedge B] = ([A] \rightarrow [B] \rightarrow X) \rightarrow X$$

Minimal Logic

$$\frac{\Gamma \vdash}{\Gamma, x \in D \vdash} \qquad \frac{\Gamma \vdash A \text{ prop}}{\Gamma, x \in A \vdash}$$

$$\frac{}{\Gamma \vdash a \in D} \qquad \frac{\Gamma \vdash t \in D}{\Gamma \vdash f\ t \in D}$$

$$\frac{\Gamma, x \in D \vdash A \text{ prop}}{\Gamma \vdash (\Pi x \in D) A \text{ prop}} \qquad \frac{\Gamma \vdash A \text{ prop} \qquad \Gamma \vdash B \text{ prop}}{\Gamma \vdash A \to B \text{ prop}}$$

Minimal Logic

$$\frac{\Gamma \vdash t \in A \rightarrow B \qquad \Gamma \vdash u \in A}{\Gamma \vdash t\ u \in B} \qquad\qquad \frac{\Gamma \vdash t \in (\Pi x \in D)A \qquad \Gamma \vdash u \in D}{\Gamma \vdash t\ u \in A(u)}$$

$$\frac{\Gamma, x \in D \vdash t : A}{\Gamma \vdash \lambda x.t : (\Pi x \in D)A} \qquad\qquad \frac{\Gamma, x \in A \vdash t : B}{\Gamma \vdash \lambda x.t : A \rightarrow B}$$

Theorem: (subject reduction) If $\Gamma \vdash t \in A$ and $t \succ t'$ then $\Gamma \vdash t' \in A$

Theorem: (normalisation) If $\Gamma \vdash t \in A$ then t is normalisable

Minimal Logic

In this way, we are reduced to show that given two sets D and X and elements $a \in D, \quad f \in D \to D$ and

$A_1 = (\Pi x \in D)\ R\ x\ (f\ x)$

$A_2 = (\Pi x \in D)(\Pi y \in D)(\Pi z \in D).R\ x\ y \to R\ y\ z \to R\ x\ z$

$A_3 = (\Pi x \in D).R\ x\ x \to X$

we cannot find t such that $h_1 \in A_1, h_2 \in A_2, h_3 \in A_3 \vdash t \in X$

This illustrates well the strength of the normalisation theorem: we need only to look for proofs ($= \lambda$-terms) in normal form

Proofs and Programs

Kleene's realisability (1940)

The realisers are Turing machines/Gödel numbers

A proof of A is like a commented program

The program NuPrl (Cornell) presents several examples of this

We can use these comments to get better specialisations of a given program with parameters

This is the idea behing Goad's notion of *pruning*, where we formalise algorithms as proofs

Goad's Pruning

Suppose that we want to solve the problem: given x, y find z such that the following holds

$$\psi(x, y, z) = z \geq x + y \wedge z \geq xy$$

We use the following facts

$x \leq 1 \to \psi(x, y, y + 1)$

$y \leq 1 \to \psi(x, y, x + 1)$

$x > 1 \wedge y > 1 \to \psi(x, y, 2xy)$

We get the following algorithm

$z = if\ x \leq 1\ then\ y + 1\ else\ if\ y \leq 1\ then\ x + 1\ else\ 2xy$

Goad's Pruning

If we specialise $y = 0.5$ we get

$z = if\ x \leq 1\ then\ 1.5\ else\ x + 1$

With pruning (remembering the reason why the program is correct) we get $z = x + 1$

This is because the correctness for the answer $x + 1$ does not depende on the case analysis on $x \leq 1$ or $x > 1$

One pruning rule is

$E(d, \lambda h_1.t, \lambda h_2.u) \succ t$

if t does not depend on h_1

Some references

http://www.cs.chalmers.se/∼bengt/

See Programming in Martin-Lf's Type Theory and Constructivism.
A Computing Science Perspective

http://www.mathematik.uni-muenchen.de/∼schwicht/

Extensional Type Theory

http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html

See especially Constable's paper, for instance "Naive Type Theory"

Proof-carrying code http://flint.cs.yale.edu/flint/

and http://www.cs.berkeley.edu/∼necula/

On-going research on formalised documents (floating-point numbers)

http://www.ens-lyon.fr/∼daumas/SoftArith/Dagsthul.pdf