Estratto da

R. Ferro e A. Zanardo (a cura di), *Atti degli incontri di logica matematica* Volume 3, Siena 8-11 gennaio 1985, Padova 24-27 ottobre 1985, Siena 2-5 aprile 1986.

Disponibile in rete su http://www.ailalogica.it

MARTIN-LÖF'S TYPE THEORY AS A PROGRAMMING LOGIC

BENGT NORDSTRÖM
Programming Methodology Group
Chalmers University of Technology and
University of Göteborg (Sweden)

I. BACKGROUND

Type theory was developed in the 70's by the swedish logician Per Martin-Löf [8]. His motives have been to clarify the syntax and semantics of the mathematical language. For him, the only way to do this is to analyze constructive mathematics. This was mainly developed by Brouwer during the beginning of this century. The basic difference between classical and constructive mathematics is that the classical meaning of a proposition is its truth value (which is either true or false) while the constructive meaning is spelled out in terms of what a proof of it is. For instance, $A \vee \neg A$ is in general not constructively valid since we have no method which can be applied to an arbitrary proposition and yield a proof of it or its negation.

The explanation in type theory of what a mathematical proposition is fits well with Heyting's [6] who explains it by explaining what counts as a proof of it. It also fits well with Kolmogorof's [7] who looks at a proposition as a problem and explains a problem by explaining what counts as a solution to it. But it is not only the notion of proposition which has to be given a precise meaning in type theory. Also concepts like *set*, *function*, *equality*, *element of*, *assumption*, *natural number* have to be analyzed.

A. The relevance of constructive mathematics for Computer Science

It is not necessary to take a standpoint in the foundational discussion about mathematics to see the advantages of constructive mathematics for computer programming:

- The notions of *computation* and *method* is basic for constructive mathematics. For instance, the function concept in constructive mathematics is exactly the same as in Computer Science; it is a method which when applied to an argument of the right kind will output something of the right kind. The function concept in classical mathematics (a subset of a cartesian product with certain properties) is not what programmers use!
- From a constructive proof of a proposition it is possible to construct a program which computes relevant information from the proof. For instance, a proof of an existential proposition $\exists x.P(x)$ will yield a program which computes an object a which has the desired property $P(a)$.

Other researchers who have proposed constructive mathematics as a basis for programming are: Bishop [1], Constable [2, 16], Goto [5], Sato [13] and Takasu [15]. Goad [4] has used computational information in constructive proofs to obtain more efficient programs.

B. The relevance of type theory for Computer Science

For me, type theory is a good conceptual framework for Computer Science. We are trying to build a new science and it is extremely important to start with a firm foundation. Type theory is a part of this. We have to be precise and must have analyzed basic concepts like program, type, specification, equality, evaluation, abstract data types, etc.

Type theory can be seen as a programming logic, a logic for the process where programmers write a program for a certain task and give arguments why the program is correct. It is an important open problem in Computer Science whether it is feasible to use the computer not only for editing, storing and executing programs but also to check that the programs are correct. To do that means that we must be able to write the task of the program in some formal language. This specification language must be powerful enough to allow the programmer to express problems, without having any idea how to solve them. We must be able to say *what* without knowing *how*. So a programming language will not do. Therefore it must also be possible to formalize the arguments why a program meets the specification so that the computer can check that the arguments are correct. Type theory suggests one way for doing this.

II. SYNTAX AND SEMANTICS OF TYPE THEORY

A. How to understand type theory

The basis of type theory is a little theory of expressions which explains what an expression is and when two expressions are (syntactically) equal. The semantics of type theory is *not* described using denotational semantics or in some other model theoretic way. Remember that the purpose of type theory was to be a clarification of the syntax and semantic of mathematics. It would only lead to a vicious circle to try to define it in terms of mathematical objects like sets and functions. Instead, these notions must be explained in some other, more direct, way. The semantics of type theory is explained using the notion of *computation*; the purely mechanical way of finding the value of an expression. There are *canonical* and *noncanonical* expressions. A canonical expression is already computed, it needs no further evaluation. Examples of canonical expressions are: $\lambda x.x+2$, 3, **true**, <3,5> and examples of noncanonical ones are: **apply** $(\lambda x.x, 3)$, **if** x **then** 3 **else** 4, 5+12.

There are four different judgement forms in type theory:

$$\begin{gathered} A \text{ set} \\ A = B \\ A \in A \\ a = b \in A \end{gathered}$$

and these are explained using the primitive notion of computation.

Then there are different set forming operations which are defined using the meaning of the judgement A set. The formal rules of type theory are justified using the semantics of the different forms of judgements. Type theory is inherently an open system. It is possible to extend it with new program forming expressions and new set forming operations.

B. The Syntax of Type Theory

The traditional way of looking at the syntax of programming languages is that programs are formed from primitive n-ary constants and variables using application. Not distinguishing between different ways of writing application (like prefix, infix, postfix and mixfix) a program is then always to be considered to be of the form $c(e_1, ..., e_n)$ where $n \geq 0$, c is a primitive constant and $e_1, ..., e_n$ are expressions built up in the same way. This is the way programs are dealt with in syntax oriented editors and it was also how expressions were built up in earlier versions of type theory. A problem with looking at expressions in this way, is that variable binding operations are not treated in a uniform way.

In Siena 1983, Martin-Löf suggested that we should look at expressions as formed by abstraction and application from variables and primitive constants. But not in the way this is done in Combinatory Logic [3] or untyped λ-calculus where any expression can be applied to another expression. This would lead to serious consequences. One is that it would be possible to apply a function like *sin* to arbitrary many arguments and form expressions like $sin(1,2,3)$, although the *sin*-function has only one argument place. It would also be possible to form expressions like $sin(sin)$. Self application, together with the defining equation for abstraction

$$(x.d)(e) \equiv d[x{:=}e]$$

where $x.d$ is the abstraction of d with respect to x and $d[x{:=}e]$ denotes the result of substituting e for all free occurrences of x in d, leads to expressions in which definitions cannot be eliminated (for instance $(x.x(x))(x.x(x)) \equiv (x.x(x))(x.x(x)) \equiv \ldots$). A third consequence of this view is that it would not be decidable if two expressions are definitionally equal or not. This will have serious consequences for the usage of formal proof rules since it must be mechanically decidable if a proof rule is properly applied. If it is possible to make explicit definitions, then the same meta-variable in the proof rules stand for definitionally equal expressions, for instance in the rule $\frac{A \supset B \quad A}{B}$ we mean that the implicand of the first premise must be definitionally equal to the second premise. So definitional equality must be decidable and definitions must be eliminable.

Instead of having just one syntactical category of expressions, as in Combinatory Logic, the expressions are divided into several categories according to which syntactical operation is applicable. There is an arity associated with each expression, showing the "functionality" of the expression. Expressions like 0 and $sin(x)$ which cannot be applied to an expression are called *saturated*, they have arity $\boldsymbol{0}$. The *unsaturated* expressions have different arities depending on what arity their argument must have and depending on what arity the expression applied to its argument has. For instance, *sin* has arity $\boldsymbol{0} \twoheadrightarrow \boldsymbol{0}$ since it expects a saturated argument and when it is applied to a saturated argument, the resulting expression is saturated. The arities are inductively defined by:

— $\boldsymbol{0}$ is an arity

— if α and β are arities, then $\alpha \twoheadrightarrow \beta$ is an arity.

Expressions with arity $\alpha \twoheadrightarrow \beta$ are expressions which only can be applied to expressions of arity α.

Each variable and constant have a unique arity associated with it. I will write $a : \alpha$ to mean that a is an expression of arity α.

We allow abbreviatory definitions (macros) of the form:

$$c \equiv e$$

where c is a unique identifier and e is an expressions without any free variables. In a definition, the left hand side is called the *definiendum* and the right hand side is the *definiens*.

Examples:

$$0 : \boldsymbol{0}$$
$$x : \boldsymbol{0}$$
$$sin : \boldsymbol{0} \twoheadrightarrow \boldsymbol{0}$$
$$+ : \boldsymbol{0} \twoheadrightarrow (\boldsymbol{0} \twoheadrightarrow \boldsymbol{0})$$
$$\int : \boldsymbol{0} \twoheadrightarrow (\boldsymbol{0} \twoheadrightarrow ((\boldsymbol{0} \twoheadrightarrow \boldsymbol{0}) \twoheadrightarrow \boldsymbol{0}))$$

In general, expressions are built up in the following way:

1. If x is a variable of arity α then x is an expression of arity α.
2. If c is a primitive constant of arity α then c is an expression of α
3. If c is a definiendum with definiens e of arity α then c is an expression of arity α.
4. If $f : \alpha \twoheadrightarrow \beta$ and $a : \alpha$ then $f(a) : \beta$.
5. If x is a variable of arity α and $e : \beta$, then $x.e : \alpha \twoheadrightarrow \beta$.

The following syntactical conventions will be used:

$\alpha \twoheadrightarrow \beta \twoheadrightarrow \gamma$ will be written instead of $\alpha \twoheadrightarrow (\beta \twoheadrightarrow \gamma)$
$f(a)(b)$ will be written instead of $(f(a))(b)$
$f(a,b)$ will be written instead of $f(a)(b)$
$x.y.c$ will be written instead of $x.(y.c)$
$(x)e$ will be written instead of $x.e$

I will use the notation $a \equiv b : \alpha$ for a and b are equal expressions of arity α. The rules for equality of expressions of a certain arity are:

1. *Variables*. If x is a variable of arity α, then
$$x \equiv x : \alpha$$
2. *Constants*. If c is a constant of arity α, then
$$c \equiv c : \alpha$$
3. *Definiens* $\equiv$ *Definiendum*. If a is a definiendum with definiens b of arity α, then
$$a \equiv b : \alpha$$
4. *Application 1*. If $a \equiv a' : \alpha \twoheadrightarrow \beta$ and $b \equiv b' : \alpha$, then
$$a(b) \equiv a'(b') : \beta$$
5. *Application 2*. *(β-rule)*. If x is a variable of arity α, a an expression of arity α and b an expression of arity β, then
$$(x.b)(a) \equiv b[x{:=}a] : \beta$$

6. *Abstraction 1. (ξ-rule).* If x is a variable of arity α and $b \equiv b' : \beta$, then

$$x.b \equiv x.b' : \alpha \twoheadrightarrow \beta$$

7. *Abstraction 2. (α-rule).* If x and y are variables of arity α and b an expression of arity β without occurrences of y, then

$$x.b \equiv y.b[x:=y] : \alpha \twoheadrightarrow \beta$$

8. *Abstraction 3. (η-rule).* If x is a variable of arity β and b is an expression of arity $\alpha \twoheadrightarrow \beta$, then

$$x.(b(x)) \equiv b : \alpha \twoheadrightarrow \beta$$

9. *Reflexivity.* If $a : \alpha$, then $a \equiv a : \alpha$.
10. *Symmetry.* If $a \equiv b : \alpha$, then $b \equiv a : \alpha$.
11. *Transitivity.* If $a \equiv b : \alpha$ and $b \equiv c : \alpha$, then $a \equiv c : \alpha$.

From a formal point of view, this is similar to typed λ-calculus with one ground type.

C. Computation

As I mentioned before, the program expressions are separated into canonical and noncanonical expressions. A closed saturated expression is always definitionally equal to an expression of the form:

$$c(e_1, e_2, ..., e_n)$$

where $n \geq 0$, c is a primitive constant and $e_1, ..., e_n$ are expressions. In type theory, the distinction between canonical and noncanonical expressions can always be made from the constant c. Therefore, also the primitive constants are divided into canonical and noncanonical. The canonical constants are called *constructors* in Computer Science and the noncanonical constants are called *selectors*. To each selector in type theory there is a computation rule explaining how a program formed by applying the selector to its arguments is computed.

The general strategy for computing expressions is lazy evaluation, i.e. expressions are computed from without and the computation process will continue until an expression which starts with a constructor is reached. So, an expression is considered to be evaluated when it is on the form $c(e_1, ..., e_n)$ where c is a constructor, $n \geq 0$ and $e_1, ..., e_n$ are expressions (not necessarily canonical). For instance, **succ**(2+3) and **cons**(3, *append* (**cons**(4, **nil**), **nil**)) are considered to be evaluated.

It may seem a little counter-intuitive that an expression is considered to be evaluated even if the parts of it are not evaluated. One reason is that when variable binding operations are introduced, it is sometimes impossible to evaluate parts of an expression. For instance, to compute the body of $\lambda x.e$ would be like trying to compute a program which expects input to continue executing it without giving it

any input. There are examples of well defined functions where the computation of the body of the function would not even terminate. Another reason for this kind of evaluation is that no extraneous computation is performed.

In order to have a notion that more closely corresponds to what one normally mean with a value and an evaluated expression, a closed saturated expression is called *fully evaluated* when it is on the form $c(e_1, e_2, ..., e_n)$ where c is a constructor and all the parts $e_1, ..., e_n$ which are saturated are also fully evaluated.

D. Different judgement forms and their semantics

The judgements are the things we prove. For instance, the forms of judgements in predicate logic is "A is true" and in Hoare logic "A is true" and "$\{P\}Q\{R\}$". In type theory there are four different forms of judgements:

A set
$A = B$
$a \in A$
$a = b \in A$

In general, judgements may depend on assumptions. A simple assumption is always on the form $x \in A$ where x is a variable of arity $\boldsymbol{0}$ and A is a set. To make an assumption corresponds to declare a variable in a programming language and to assume that a proposition is true in predicate logic. Being a set may in general depend on variables, so in general one assumption may depend on earlier assumptions. The hypothetical judgements have the form:

$$A(x_1, ..., x_n) \text{ set } [x_1 \in A_1, x_2 \in A_1(x_1), ..., x_n \in A_n(x_1, ..., x_{n-1})]$$
$$A(x_1, ..., x_n) = B(x_1, ..., x_n)\ [x_1 \in A_1, x_2 \in A_1(x_1), ..., x_n \in A_n(x_1, ..., x_{n-1})]$$
$$a(x_1, ..., x_n) \in A(x_1, ..., x_n)\ [x_1 \in A_1, x_2 \in A_1(x_1), ..., x_n \in A_n(x_1, ..., x_{n-1})]$$
$$a(x_1, ..., x_n) = b(x_1, ..., x_n) \in A(x_1, ..., x_n)$$
$$[x_1 \in A_1, x_2 \in A_1(x_1), ..., x_n \in A_n(x_1, ..., x_{n-1})]$$

I will start by giving the semantics of the different judgement forms when they don't depend on assumptions.

A set

We understand that A is a set when we know how to form the canonical elements of the set and know when two canonical elements are equal.

$A = B$

If A and B are sets, then $A = B$ means that the canonical elements of A are canonical elements in B and equal canonical elements in A are equal elements in B and vice versa. This explanation makes sense, since if we know that A are B are sets then we also know their canonical elements and when two canonical elements are equal.

$$a \in A$$

If A is a set, then $a \in A$ means that the value of a is a canonical element in A.

$$a = b \in A$$

If A is a set, then $a = b \in A$ means that the values of a and b are equal canonical elements in A.

The meaning of the hypothetical judgements are given by an induction over the length of the assumption list. To illustrate the general principle, I will give the meaning of the judgement forms when they depend on one assumption.

$$B(x) \text{ set } [x \in A]$$

If A is a set then $B(x)$ is a family of sets indexed by elements in A means that $B(a)$ is a set for an arbitrary element a in A. It also means that B is extensional in the sense that if $a = a' \in A$ then $B(a) = B(a')$. Since we know that A is a set we also know the meaning of $a \in A$ and $a = a' \in A$.

$$B(x) = C(x) \; [x \in A]$$

If A is a set and $B(x)$ and $C(x)$ are families of sets indexed by elements in A then $B(x) = C(x) \; [x \in A]$ means that $B(a) = C(a)$ for an arbitrary element a in A.

$$b(x) \in B(x) \; [x \in A]$$

If A is a set and $B(x)$ is a family of sets indexed by elements in A then $b(x) \in B(x) \; [x \in A]$ means that $b(a) \in B(a)$ whenever $a \in A$. It also means that b is extensional in the sense that $b(a) = b(a') \in B(a)$ whenever $a = a' \in A$.

$$b(x) = b'(x) \in B(x) \; [x \in A]$$

If A is a set and $B(x)$ is a family of sets indexed by elements in A then $b(x) = b'(x) \in B(x) \; [x \in A]$ means that $b(a) = b'(a) \in B(a)$ whenever $a \in A$.

E. Propositions and types

The judgement $a \in A$ can be given different readings:

— a is an element in the set A

— a is an object in the type A

— a is a program for the task (specification) A

— a is a proof object of the proposition A.

This is not just a formal coincidence. The reason is that the notions set, type, task and proposition are explained in the same way. Let's look at Heyting's explanation of the logical constants:

A proof of:	**consists of:**
$A \& B$	a proof of A and a proof of B.
$A \vee B$	a proof of A or a proof of B.
$A \supset B$	a method which when applied to an arbitrary proof of A yields a proof of B.
$\forall x \in A. B(x)$	a method which when applied to an arbitrary object a in A yields a proof of $B(a)$.
$\exists x \in A. B(x)$	an object a in A and a proof of $B(a)$.

Let's compare this with the explanation of some types:

An object of:	**consists of:**
$A \times B$	**pair**(a,b), where $a \in A$ and $b \in B$, i.e. an object of A and an object of B.
$A + B$	**inl**(a), where $a \in A$ or **inr**(b), where $b \in B$, i.e. an object of A or an object of B.
$A \to B$	$\lambda(b)$, where $b(x) \in B$ if $x \in A$, i.e. a method which when applied to an arbitrary object of A yields an object of B.
$\Pi x \in A. B(x)$	$\lambda(b)$, where $b(x) \in B(x)$, if $x \in A$, i.e. a method which when applied to an arbitrary object a of A yields an object of $B(a)$.
$\Sigma x \in A. B(x)$	**pair** (a,b), where $a \in A$ and $b \in B(a)$, i.e. an object a of A and an object of $B(a)$.

So we see that conjunction is explained similarly to the cartesian product, disjunction similarly to the disjoint union etc. We therefore make the following explicit definitions:

$$\begin{aligned} A \& B &\equiv A \times B \\ A \vee B &\equiv A + B \\ A \supset B &\equiv A \to B \\ \forall x \in A. B(x) &\equiv \Pi x \in A. B(x) \\ \exists x \in A. B(x) &\equiv \Sigma x \in A. B(x) \end{aligned}$$

In order to express atomic propositions, we must have sets corresponding to $\perp$ (the absurdity), $\mathbf{T}$ (the truth) and one to express that two elements a and b are equal in a set A. We have no proof of $\perp$, so it corresponds to $\varnothing$, the empty set. We always have a proof of $\mathbf{T}$, so any nonempty set can be used to express it. Finally the set $\mathrm{Eq}(A, a, b)$ is nonempty exactly when $a = b \in A$.

If we have a set A which we are going to read as a proposition, then we are usually not interested in the elements in the set, only in the fact that the set is inhabited, i.e. that the proposition is true. In that case I will write A prop instead of A set and A true instead of $a \in A$.

III. AN OVERVIEW OF SOME SET FORMING OPERATIONS

In the rest of this course I will talk briefly about the most important set forming operations in type theory. Intuitively, we have the following set forming operations:

$$\begin{aligned}
\Sigma x \in A.B(x) &= \{\mathbf{pair}(a,b) \mid a \in A,\ b \in B(a)\}\\
A \times B &= \{\mathbf{pair}(a,b) \mid a \in A,\ b \in B\}\\
\Pi x \in A.B(x) &= \{\lambda(b) \mid b(x) \in B(x)\ [x \in A]\}\\
A \rightarrow B &= \{\lambda(b) \mid b(x) \in b\ [x \in A]\}\\
A + B &= \{\mathbf{inl}(a),\ \mathbf{inr}(b) \mid a \in A,\ b \in B\}\\
\mathrm{N} &= \{0, \mathbf{succ}(a) \mid a \in \mathrm{N}\}\\
\mathbf{List}(A) &= \{\mathbf{nil}, \mathbf{cons}(a,b) \mid a \in A,\ b \in \mathbf{List}(A)\}\\
\mathbf{T} &= \{\mathbf{tt}\}\\
\perp &= \{\ \}\\
\mathbf{Eq}(A,a,b) &= \{\mathbf{eq} \mid a = b \in A\}\\
\{x \in A \mid B(x)\} &= \{a \mid a \in A, b \in B(a)\}
\end{aligned}$$

These equalities should be read in the following way: The canonical elements in the set $\Sigma x \in A.B(x)$ are of the form $\mathbf{pair}(a,b)$ where $a \in A$ and $b \in B(a)$ etc.

In the formal system of type theory there are first some general rules about assumptions, equality and substitution of variables. Then for each set forming operation F there are four kinds of rules:

The *formation rules* for F describes when $F(A_1, \ldots, A_n)$ is a set and when $F(A_1, \ldots, A_n) = F(B_1, \ldots, B_n)$.

The *introduction rules* for F defines F in the sense that it describes how the canonical elements of $F(A_1, \ldots, A_n)$ are formed and when two canonical elements are equal.

The *elimination rules* for F expresses an induction principle for $F(A_1, \ldots, A_n)$. If $C(x)$ is a family of sets indexed by elements in $F(A_1, \ldots, A_n)$ and p is an element in $F(A_1, \ldots, A_n)$ the elimination rule gives conditions for the judgement $e \in C(p)$, where e is (in general) a noncanonical expression involving the selector associated with F. If we read $C(x)$ as a property of elements in $F(A_1, \ldots, A_n)$ we can see the elimination rule as giving conditions for proving that $C(p)$ is true for $p \in F(A_1, \ldots, A_n)$.

The *equality rules* for F is a formalization of the equalities between elements which the computation rule for the selector of F gives rise to.

I will only give the most important rules and justify one of them. For a more complete treatment I refer to Martin-Löf's Hannover paper [8] and his lectures in Padova [9]. In Göteborg, we are currently writing an introduction to type theory for computer scientists.

The rules will be given in a natural deduction style:

$$\frac{P_1 \qquad P_2 \qquad \ldots \qquad P_n}{C}$$

or

$$\begin{array}{c} P_1 \\ P_2 \\ \cdot \\ \cdot \\ \cdot \\ \dfrac{P_n}{C} \end{array}$$

where the premises $P_1, \ldots, P_n$ and the conclusion C are judgements. The judgements are in general hypothetical, but only those assumptions that are discharged in a rule will be presented. If a rule has a conclusion of the form $a \in A$ then the premise A set will not always be given. Similarly, if the conclusion is of the form $a = b \in A$ then the premises A set, $a \in A$ and $b \in B$ will sometimes be left implicit.

A. The disjoint union of a family of sets: the existential quantifier, the cartesian product of two sets and conjunction

In order to form a disjoint union of a family of sets we must have a set A and a family B of sets over A, i.e. $B(x)$ set $[x \in A]$. The primitive constant Σ of arity $\mathbf{0} \twoheadrightarrow (\mathbf{0} \twoheadrightarrow \mathbf{0}) \twoheadrightarrow \mathbf{0}$ will be used to form a disjoint union of a family of sets, so $\Sigma(A, B)$ is the disjoint union of B over A. It will also be written $(\Sigma x \in A)B(x)$ and $\sum_{x \in A} B(x)$. The formation rule for the set is:

Σ-formation:

$$\frac{A \text{ set} \qquad B(x) \text{ set } [x \in A]}{\Sigma(A,B) \text{ set}}$$

$$\frac{A = A' \qquad B(x) = B'(x)\ [x \in A]}{\Sigma(A,B) = \Pi(A',B')}$$

The canonical elements in $\Sigma(A,B)$ are of the form **pair**(a,b), where $a \in A$ and $b \in B(a)$. Two canonical elements are equal if their components are equal:

Σ-introduction:

$$\frac{a \in A \qquad b \in B(a)}{\mathbf{pair}(a,b) \in \Sigma(A,B)}$$

$$\frac{a = a' \in A \qquad b = b' \in B(a)}{\mathbf{pair}(a,b) = \mathbf{pair}(a',b') \in B(a)}$$

I will sometimes use the following abbreviation: $<a,b> \equiv \mathbf{pair}(a,b)$.

The selector associated with Σ is **split** of arity $\mathbf{0}\twoheadrightarrow(\mathbf{0}\twoheadrightarrow\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$. The expression **split**(p,e) is computed by first computing p. If the value of p is **pair**(a,b) then the value of **split**(p,e) is the value of $e(a,b)$.

The remaining rules are:

Σ-elimination:

$$\frac{p \in \Sigma(A,B) \qquad e(x,y) \in C(\mathbf{pair}(x,y))\ [x \in A, y \in B(x)]}{\mathbf{split}(p,e) \in C(p)}$$

$$\frac{p = p' \in \Sigma(A,B) \qquad e(x,y) = e'(x,y) \in C(\mathbf{pair}(x,y))\ [x \in A, y \in B(x)]}{\mathbf{split}(p,e) = \mathbf{split}(p',e') \in C(p)}$$

Σ-equality:

$$\frac{a \in A \qquad b \in B(a) \qquad e(x,y) \in C(\mathbf{pair}(x,y))\ [x \in A, y \in B(x)]}{\mathbf{split}(\mathbf{pair}(a,b),e) = e(a,b) \in C(\mathbf{pair}(a,b))}$$

In the following I will not present the formation-, introduction- and elimination-rules involving equality, they are always in the same style as the ones for the Σ set former. They state roughly that two elements (sets) are equal if their components are equal.

As an illustration how to justify these rules, I will show that the first elimination rule is correct. Assume the premises $p \in \Sigma(A,B)$ and $e(x,y) \in C(\mathbf{pair}(x,y))\ [x \in A, y \in B(x)]$. We must show that the value of **split**(p,e) is a canonical element in $C(p)$. What would we get if we compute **split**(p,e)? We first compute the value of p. But since $p \in \Sigma(A,B)$ we know that the value of p is a canonical element in $\Sigma(A,B)$, i.e. of the form **pair**(a,b), where $a \in A$ and $b \in B(a)$. But in that case, the value of **split**(p,e) is the value of $e(a,b)$. The meaning of the second premise gives now that the value of $e(a,b)$ is a canonical element in $C(\mathbf{pair}(a,b))$ since $a \in A$ and $b \in B(a)$.

So we have shown that the value of **split**(p,e) is a canonical element in $C(\mathbf{pair}(a,b))$. What remains to show is that $C(p) = C(\mathbf{pair}(a,b))$, because in that case, the value of **split**(p,e) is also a canonical element in $C(p)$ (by the meaning of type equality).

We know that $p = \mathbf{pair}(a,b) \in \Sigma(A,B)$ since the values of p and **pair**(a,b) are equal canonical elements in $\Sigma(A,B)$. Finally, from the meaning of the implicit premise $C(z)$ set $[z \in \Sigma(A,B)]$ we get that $C(p) = C(\mathbf{pair}(a,b))$.

1. The existential quantifier

We get the existential quantifier by reading the judgement $B(x)$ set $[x \in A]$ as $B(x)$ is a proposition for $x \in A$, or equivalently, that B is a property of elements in A.

We get the rules for the existential quantifier by using the explicit definition $\exists x \in A.B(x) \equiv \Sigma(A,B)$ and by also reading $\Sigma(A,B)$ as a proposition and the family $C(z)$ in the elimination rule as a proposition (no longer depending on objects in $\Sigma(A,B)$):

∃-formation:

$$\frac{A \text{ set} \qquad B(x) \text{ prop } [x \in A]}{\exists x \in A.B(x) \text{ prop}}$$

∃-introduction:

$$\frac{a \in A \qquad B(a) \text{ true}}{\exists x \in A.B(x) \text{ true}}$$

∃-elimination:

$$\frac{\exists x \in A.B(x) \text{ true} \qquad C \text{ true } [x \in A, B(x) \text{ true}]}{C \text{ true}}$$

2. The cartesian product of two sets

We get the cartesian product of two sets A and B as a special case of $\Sigma(A,B)$ when B is a set instead of being a family of sets indexed by A. We get the rules for $A \times B$ if we make the explicit definition $A \times B \equiv \Sigma(A,(x)B)$ (where x is not free in B):

×-formation:

$$\frac{A \text{ set} \qquad B \text{ set } [x \in A]}{A \times B \text{ set}}$$

×-introduction:

$$\frac{a \in A \qquad b \in B}{\mathbf{pair}(a,b) \in A \times B}$$

×-elimination:

$$\frac{p \in A \times B \qquad e(x,y) \in C(\mathbf{pair}(x,y))\ [x \in A, y \in B]}{\mathbf{split}(p,e) \in C(p)}$$

3. Conjunction

We get conjunction by making the explicit definition $A \& B \equiv A \times B$ and reading both A and B as propositions. The rules we get is a slight generalization of the ordinary rules for conjunction:

&-formation:

$$\frac{A \text{ prop} \qquad B \text{ prop } [A \text{ true}]}{A \& B \text{ prop}}$$

&-introduction:

$$\frac{A \text{ true} \qquad B \text{ true}}{A \& B \text{ true}}$$

&-elimination:

$$\frac{A \& B \text{ true} \qquad C \text{ true } [A \text{ true}, B \text{ true}]}{C \text{ true}}$$

The generalization is in the weakening of the second premise in the formation rule. To show that $A \& B$ is a proposition it is enough to show that A is a proposition and that B is a proposition under the assumption that A is true. This is important for Computer Science applications when dealing with partially defined functions, we want for instance to be able to write propositions like:

$$(x \neq 0) \ \& \ ((z/x) \cdot x \ =_{\mathbf{N}} z)$$

where the second conjunct is not a proposition if $x = 0$.

There is a similar weakening of the second premise in the elimination rule. The assumption that B is true may depend on the assumption that A is true.

B. The disjoint union of two sets

If A and B are sets, then the canonical elements in the set $A+B$ are of the form $\mathbf{inl}(a)$, where $a \in A$ or $\mathbf{inr}(b)$, where $b \in B$. The arity of + is $\boldsymbol{0} \twoheadrightarrow \boldsymbol{0} \twoheadrightarrow \boldsymbol{0}$ and the arity of **inl** and **inr** is $\boldsymbol{0} \twoheadrightarrow \boldsymbol{0}$.

+-formation:

$$\frac{A \text{ set} \qquad B \text{ set}}{A+B \text{ set}}$$

+-introduction:

$$\frac{a \in A}{\mathbf{inl}(a) \in A+B} \qquad \frac{b \in B}{\mathbf{inr}(b) \in A+B}$$

The selector for elements in a disjoint union is the constant **when** of arity $\boldsymbol{0} \twoheadrightarrow (\boldsymbol{0} \twoheadrightarrow \boldsymbol{0}) \twoheadrightarrow (\boldsymbol{0} \twoheadrightarrow \boldsymbol{0}) \twoheadrightarrow \boldsymbol{0}$. The expression $\mathbf{when}(p,d,e)$ is computed in the following way. Compute first p. If the value of p is $\mathbf{inl}(a)$, then the value of $\mathbf{when}(p,d,e)$ is the value of $d(a)$. If the value of p is $\mathbf{inr}(b)$, then the value of $\mathbf{when}(p,d,e)$ is the value of $e(b)$.

The remaining rules are:

+-elimination:

$$\frac{p \in A+B \qquad d \in C(\mathbf{inl}(a))\ [a \in A] \qquad e \in C(\mathbf{inr}(b))\ [b \in B]}{\mathbf{when}(p,d,e) \in C(p)}$$

+-equality:

$$\frac{a \in A \qquad d \in C(\mathbf{inl}(a))\ [a \in A] \qquad e \in C(\mathbf{inr}(b))\ [b \in B]}{\mathbf{when}(\mathbf{inl}(a),d,e) = d(a) \in C(\mathbf{inl}(a))}$$

$$\frac{b \in B \qquad d \in C(\mathbf{inl}(a))\ [a \in A] \qquad e \in C(\mathbf{inr}(b))\ [b \in B]}{\mathbf{when}(\mathbf{inr}(b),d,e) = e(b) \in C(\mathbf{inr}(b))}$$

We will later see that it is possible to define a restricted disjoint union of two sets by looking at it as a disjoint union of a family of sets indexed by a set with two elements. But in order to form the family of sets indexed by a set with two elements we need the set **U** which I will talk about in the last lecture.

C. The cartesian product of a family of sets: the universal quantifier, the set of functions and implication

If A is a set and B a family of sets over A then the canonical elements in the set $\Pi(A,B)$ are functions $\lambda(b)$ where $b(x) \in B(x)$ if $x \in A$. The primitive constants Π and λ have arities $\mathbf{0}\twoheadrightarrow(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$ and $(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$, respectively. The set $\Pi(A,B)$ will also be written $\Pi x \in A.\, B(x)$ and $\prod_{x\in A} B(x)$. Two canonical elements $\lambda(b)$ and $\lambda(b')$ are equal if $b(x) = b'(x)\ [x \in A]$. The formation and introduction rules for the set are:

Π-formation:

$$\frac{A \text{ set} \qquad B(x) \text{ set } [x \in A]}{\Pi(A,B) \text{ set}}$$

Π-introduction:

$$\frac{b(x) \in B(x)\ [x \in A]}{\lambda(b) \in \Pi(A,B)}$$

The selector associated with Π is **funsplit** of arity $\mathbf{0}\twoheadrightarrow((\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$. The expression **funsplit**(p,e) is computed by first computing p. If the value of p is $\lambda(b)$ then the value of **funsplit**(p,e) is the value of $e(b)$. This selector corresponds to a pattern matching operation in functional programming languages: **funsplit**$(p, u.g)$ corresponds to an expression like **cases** p **of** $\lambda(u) : g$ **endcases** where u is a variable of arity $\mathbf{0}\twoheadrightarrow\mathbf{0}$ and g an expression which may contain occurrences of u. We get the ordinary operator for function application by the explicit definition:

apply$(f, a) \equiv$ **funsplit** $(f, u.(u(a)))$

The elimination and equality rules for the type are the following:

Π-elimination 1:

$$\frac{p \in \Pi(A,B) \qquad e(u) \in C(\lambda(u))\ [u(x) \in B(x)\ [x \in A]]}{\textbf{funsplit}\ (p,e) \in C(p)}$$

Π-equality:

$$\frac{b(x) \in B(x)\ [x \in A] \qquad e(u) \in C(\lambda(u)\ [u(x) \in B(x)\ [x \in A]]}{\textbf{funsplit}\ (\lambda(b), e) = e(b) \in C(\lambda(b))}$$

In these rules I have used a more general way of making assumptions than the simple assumptions of the form $x \in A$ which I talked about in the previous lecture. It is not difficult to show that the following rule can seen as a derived rule:

Π-elimination 2:

$$\frac{p \in \Pi(A,B) \qquad a \in A}{\textbf{apply}(p,a) \in B(a)}$$

1. The universal quantifier

The universal quantifier is introduced by the explicit definition

$$\forall x \in A.\, B(x) \equiv \Pi(A,B)$$

and we get the rules for $\forall$ from the rules of Π by reading B as a property of elements in A and reading the family C in the elimination rule as a proposition (no longer depending on objects in $\Pi(A,B)$):

$\forall$-formation:

$$\frac{A \text{ set} \qquad B(x) \text{ prop } [x \in A]}{\forall x \in A.\, B(x) \text{ prop}}$$

$\forall$-introduction:

$$\frac{B(x) \text{ true } [x \in A]}{\forall x \in A.\, B(x) \text{ true}}$$

$\forall$-elimination 1:

$$\frac{\forall x \in A.\, B(x) \text{ true} \qquad C \text{ true } [B(x) \text{ true } [x \in A]]}{C \text{ true}}$$

From the derived Π-elimination 2 -rule, we get the ordinary elimination rule for $\forall$:

$\forall$-elimination 2:

$$\frac{\forall x \in A.\, B(x) \text{ true} \qquad a \in A}{B(a) \text{ true}}$$

2. The set of functions from a set to a set

If x is a variable which is not free in B then we can make the following explicit definition

$$A \rightarrow B \equiv \Pi(A, x.B)$$

and we get the following rules for the set $A \rightarrow B$ of functions from A to B:

$\rightarrow$ formation:

$$\frac{A \text{ set} \qquad B \text{ set } [x \in A]}{A \rightarrow B \text{ set}}$$

$\rightarrow$ introduction:

$$\frac{b(x) \in B \ [x \in A]}{\lambda(b) \in A \rightarrow B}$$

$\rightarrow$ elimination 1:

$$\frac{p \in A \rightarrow B \qquad e(u) \in C(\lambda(u)) \ [u(x) \in B \ [x \in A]]}{\mathbf{funsplit}(p, e) \in C(p)}$$

$\rightarrow$ elimination 2:

$$\frac{p \in A \rightarrow B \qquad a \in A}{\mathbf{apply}(p, a) \in B}$$

Notice the weakened second premise of the formation rule. In order to show that $A \rightarrow B$ is a set it is sufficient to show that A is a set and that B is a set if A is non-empty.

3. Implication

We get the implication $A \supset B$ by making the explicit definition $A \supset B \equiv A \rightarrow B$ and reading both A and B as propositions:

$\supset$-formation:

$$\frac{A \text{ prop} \qquad B \text{ prop } [A \text{ true}]}{A \supset B \text{ prop}}$$

$\supset$-introduction:

$$\frac{B \text{ true } [A \text{ true}]}{A \supset B \text{ true}}$$

$\supset$-elimination 1:

$$\frac{A \supset B \text{ true} \qquad C \text{ true } [B \text{ true } [A \text{ true}]]}{C \text{ true}}$$

$\supset$-elimination 2:

$$\frac{A \supset B \text{ true} \qquad A \text{ true}}{B \text{ true}}$$

D. The natural numbers

The canonical elements in **N** are 0 and **succ**(a), where $a \in$ **N**. If $a = a' \in$ **N** then **succ**(a) = **succ**(a') $\in$ **N**. The selector associated with **N** is **rec**, the primitive recursion operator. The arity of **N** and 0 is ***0***, the arity of **succ** is $\boldsymbol{0} \twoheadrightarrow \boldsymbol{0}$ and the arity of **rec** is $\boldsymbol{0} \twoheadrightarrow \boldsymbol{0} \twoheadrightarrow (\boldsymbol{0} \twoheadrightarrow \boldsymbol{0} \twoheadrightarrow \boldsymbol{0}) \twoheadrightarrow \boldsymbol{0}$. The noncanonical expression

$$\mathbf{rec}(p, d, e)$$

is computed in the following way:

First compute p. If the value of p is 0, then the value of **rec**(p, d, e) is the value of d. If the value of p is **succ**(a), then the value of **rec**(p, d, e) is the value of $e(a,$ **rec** $(a, d, e))$.

If we have made the following explicit definition:

$$f(x) \equiv \mathbf{rec}(x, d, e)$$
$$\text{where } x \in N, d \in A, \ e(y, z) \in A \ [y \in \mathbf{N}, z \in A]$$

then we know that

$$\begin{cases} f(0) = d \in A \\ f(\mathbf{succ}(x)) = e(x, f(x)) \in A \end{cases}$$

There is no operator in type theory which corresponds to the general recursion operator. This may seem restrictive, but we know that the primitive recursion operator together with the possibilities of using higher order functions give us the possibility to express all provably (within Peano arithmetic) total functions. For instance, Ackermann's function can be expressed in type theory [10].

We have the following rules for the set of natural numbers:

N-formation:

$$\mathbf{N} \text{ set}$$

N-introduction:

$$0 \in \mathbf{N} \qquad\qquad \frac{a \in \mathbf{N}}{\mathbf{succ}(a) \in \mathbf{N}}$$

N-elimination:

$$\frac{p \in \mathbf{N} \qquad d \in C(0) \qquad e(x, y) \in C(\mathbf{succ}(x)) \ [x \in \mathbf{N}, y \in C(x)]}{\mathbf{rec}(p, d, e) \in C(p)}$$

N-equality 1:

$$\frac{d \in C(0)}{\mathbf{rec}(0,d,e) = d \in C(0)}$$

N-equality 2:

$$\frac{a \in \mathbf{N} \qquad e(x,y) \in C(\mathbf{succ}(x))\ [x \in \mathbf{N}, y \in C(x)]}{\mathbf{rec}(\mathbf{succ}(a),d,e) = e(a,\mathbf{rec}(a,d,e)) \in C(\mathbf{succ}(a))}$$

If we in the elimination rule read $C(x)$ set $[x \in \mathbf{N}]$ as that C is a property of natural numbers we get Peano's fifth axiom:

$$\frac{p \in \mathbf{N} \qquad C(0) \text{ true} \qquad C(\mathbf{succ}(x)) \text{ true } [x \in \mathbf{N}, C(x) \text{ true}]}{C(p) \text{ true}}$$

The rules in type theory corresponding to Peano's third and fourth axiom can be derived within the formal system.

Examples:

$x+y \equiv \mathbf{rec}(x,y,u.w.\mathbf{succ}(w))$

$x \cdot y \equiv \mathbf{rec}(x,0,u.w.(y+w))$

E. Lists

Lists are defined similarly to the natural numbers. If A is a set then $\mathbf{List}(A)$ is a set. The canonical elements in $\mathbf{List}(A)$ are **nil** and $\mathbf{cons}(a,b)$, where $a \in A$ and $b \in \mathbf{List}(A)$.

List-formation:

$$\frac{A \text{ set}}{\mathbf{List}(A) \text{ set}}$$

List-introduction:

$$\mathbf{nil} \in \mathbf{List}(A) \qquad\qquad \frac{a \in A \qquad b \in \mathbf{List}(A)}{\mathbf{cons}(a,b) \in \mathbf{List}(A)}$$

The selector for lists is the primitive constant **listrec** of arity $\boldsymbol{0 \twoheadrightarrow 0 \twoheadrightarrow (0 \twoheadrightarrow 0 \twoheadrightarrow 0 \twoheadrightarrow 0) \twoheadrightarrow 0}$.

The expression $\mathbf{listrec}(p,d,e)$ is computed by first computing p. If the value of p is **nil** then the value of the **listrec**-expression is the value of d. If the value of p is $\mathbf{cons}(a,b)$ then the value of the **listrec**-expression is the value of $e(a,b,\mathbf{listrec}(b,d,e))$.

The remaining rules are:

List-elimination:

$$\frac{\begin{array}{l} p \in \mathbf{List}(A) \\ d \in C(\mathbf{nil}) \\ e(x,y,z) \in C(\mathbf{cons}(x,y)\ [x \in A,\ y \in \mathbf{List}(A),\ z \in C(y)] \end{array}}{\mathbf{listrec}(p,d,e) \in C(p)}$$

List-equality:

$$\frac{d \in C(\mathbf{nil})}{\mathbf{listrec}(\mathbf{nil},d,e) = d \in C(\mathbf{nil})}$$

$$\frac{\begin{array}{l} a \in A \\ b \in \mathbf{List}(A) \\ e(x,y,z) \in C(\mathbf{cons}(x,y))\ [x \in A,\ y \in \mathbf{List}(A),\ z \in C(y)] \end{array}}{\mathbf{listrec}(\mathbf{cons}(a,b),d,e) = e(a,b,\mathbf{listrec}(b,d,e)) \in C(\mathbf{cons}(a,b)}$$

F. Enumeration sets: the empty set, the absurdity, the one-element set, the truth, the set of Boolean values

If we have n canonical constants $i_1, i_2, \ldots, i_n$, for $n \geq 0$, then the enumeration set $\{i_1, \ldots, i_n\}$ has the canonical elements $i_1, \ldots, i_n$. The selector associated with the set is **case** and $\mathbf{case}(p, e_1, \ldots, e_n)$ is computed by first computing p. If the value of p is i_k, $1 \leq k \leq n$ then the value of the **case**-expression is the value of e_k.

We have the following rules:

$\{i_1, \ldots, i_n\}$-formation:

$$\{i_1, \ldots, i_n\} \text{ set}$$

$\{i_1, \ldots, i_n\}$-introduction:

$$i_1 \in \{i_1, \ldots, i_n\} \qquad \ldots \qquad i_n \in \{i_1, \ldots, i_n\}$$

$\{i_1, \ldots, i_n\}$-elimination:

$$\frac{p \in \{i_1, \ldots, i_n\} \qquad e_1 \in C(i_1) \qquad \ldots \qquad e_n \in C(i_n)}{\mathbf{case}(p, e_1, \ldots, e_n) \in C(p)}$$

$\{i_1, \ldots, i_n\}$-equality:

$$\frac{e_1 \in C(i_1)}{\mathbf{case}(i_1, e_1, \ldots, e_n) = e_1 \in C(i_1)}$$

$$\vdots$$

$$\frac{e_n \in C(i_n)}{\mathbf{case}(i_n, e_1, \ldots, e_n) = e_n \in C(i_n)}$$

1. The empty set

For n=0, we get

{ }-formation:

$$\frac{}{\{\}\ \text{set}}$$

{ }-introduction:

{ }-elimination:

$$\frac{p \in \{\}}{\mathbf{case}(p) \in C(p)}$$

2. The absurdity

Reading { } as a proposition, we get the absurdity, the proposition which has no proof. So we make the explicit definition $\bot \equiv \{\}$ and get the natural deduction rule for absurdity from the rule for { }-elimination.

$\bot$-elimination:

$$\frac{\bot\ \text{true}}{C\ \text{true}}$$

where C is an arbitrary proposition.

3. The one-element set, the truth

There are many sets which are non-empty and hence can be used to represent the proposition which is always true. We make the following definition:

$$\mathbf{T} \equiv \{\mathbf{tt}\}$$

and get the following rules as special cases of the $\{\mathbf{tt}\}$-rules:

T-introduction:

$$\mathbf{T}\ \text{true}$$

T-elimination:

$$\frac{\mathbf{T}\ \text{true} \qquad C\ \text{true}}{C\ \text{true}}$$

4. The two-element set, Bool

We make the definitions:

$$\mathbf{Bool} \equiv \{\mathbf{true}, \mathbf{false}\}$$

$$\mathbf{if}(p, d, e) \equiv \mathbf{case}(p, d, e)$$

and get the following rules for $\{\mathbf{true}, \mathbf{false}\}$.

Bool-formation:

$$\mathbf{Bool}\ \text{set}$$

Bool-introduction:

$$\mathbf{true} \in \mathbf{Bool} \qquad \mathbf{false} \in \mathbf{Bool}$$

Bool-elimination:

$$\frac{p \in \mathbf{Bool} \qquad e_1 \in C(\mathbf{true}) \qquad e_2 \in C(\mathbf{false})}{\mathbf{if}(p, e_1, e_2) \in C(p)}$$

Bool-equality:

$$\frac{e_1 \in C(\mathbf{true})}{\mathbf{if}(\mathbf{true}, e_1, e_2) = e_1 \in C(\mathbf{true})} \qquad \frac{e_2 \in C(\mathbf{false})}{\mathbf{if}(\mathbf{false}, e_1, e_2) = e_2 \in C(\mathbf{false})}$$

G. Propositional equality

In order to build up propositions from equality between programs, it is necessary to have a set (proposition) which expresses that two elements (programs) are equal. This set is **Eq**(A, a, b) and it has **eq** as canonical element if $a = b \in A$.

Eq-formation:

$$\frac{A \text{ set} \qquad a \in A \qquad b \in A}{\mathbf{Eq}(A, a, b) \text{ set}}$$

Eq-introduction:

$$\frac{a = b \in A}{\mathbf{eq} \in \mathbf{Eq}(A, a, b)}$$

Eq-elimination:

$$\frac{c \in \mathbf{Eq}(A, a, b)}{a = b \in A}$$

Eq-equality:

$$\frac{c \in \mathbf{Eq}(A, a, b)}{c = \mathbf{eq} \in \mathbf{Eq}(A, a, b)}$$

H. Subsets

Many programming problem are of the following form:

> Find a function f which takes input a from a set A and outputs an element b from a set B such that $Q(a, b)$ holds.

This is almost expressed in type theory as:

$$\forall x \in a. \exists y \in B. Q(x, y)$$

or, equivalently

$$\Pi x \in A. \Sigma y \in B. Q(x, y)$$

but this set doesn't exactly express the problem! A program in this set is a function f which when applied to $a \in A$ yields a pair $<b, c>$, where $b \in B$ and c is a proof object in $Q(a, b)$. The problem is that the canonical elements in a Σ-set are *pairs* and in this case we are not interested in the second component of these pairs. The situation becomes more unsatisfactory in programming problems of the form:

> Find a function f which takes as input an element a in A for which $P(a)$ holds and outputs an element $b \in B$ such that $Q(a, b)$ holds.

This is almost expressed as:

$$\Pi x \in (\Sigma z \in A. P(z)). \Sigma y \in B. Q(x, y)$$

which is a set of functions f which when applied to *pairs* of the form $<a, d>$, where $a \in A$ and $d \in P(a)$, yield a *pair* $<b, c>$, where $b \in B$ and $c \in Q(<a, d>, b)$.

In order to cope with situations like these we have a set $\{x \in A \mid B(x)\}$. The canonical elements in this set are the canonical elements in A for which $B(a)$ is true.

The following rules are easily justified:

subset-formation:

$$\frac{A \text{ set} \qquad B(x) \text{ set } [x \in A]}{\{x \in A \mid B(x)\} \text{ set}}$$

subset-introduction:

$$\frac{a \in A \qquad b \in B(a)}{a \in \{x \in a \mid B(x)\}}$$

subset-elimination:

$$\frac{p \in \{x \in a \mid B(x)\} \qquad e(x) \in C(x)\ [x \in A, y \in B(x)]}{e(p) \in C(p)}$$

So the subset $\{x \in A \mid B(x)\}$ works almost like the set $\Sigma x \in A. B(x)$, we just forget the second component of the pairs which are the canonical elements in the Σ-set.[1] We can now formulate the two previous problems as:

$$\Pi x \in A. \{y \in B \mid Q(x)y)\}$$
$$\Pi x \in \{z \in A \mid P(z)\}. \{y \in B \mid Q(x, y)\}$$

which express exactly what we wanted.

In situations like these, we are only interested to read a set as a proposition and the elements in the set are not computationally interesting. It would be much more convenient to have a formalized programming logic where we also have the judgement forms A prop and A true. We would then have a language which is closer to Martin-Löf's logical language presented here in Siena a couple of years ago.

1. Notice that if we instead forget the first component of the pairs we get a set forming operation similar to a join of a family of sets.

I. The first universe $\mathbf{U}_0$

$\mathbf{U}_0$ is a set which contains (codings of) sets as elements. In Computer Science we need it for many specifications when the most natural way of expressing a proposition is to use recursion or conditionals. It is also necessary when defining abstract data types in type theory.

Let's look at the following example which is a specification of a sorting algorithm. The problem is to find a function which outputs a sorted permutation of its input.

$$Sort \equiv \Pi x \in \mathbf{List}(\mathbf{N}).\{y \in \mathbf{List}(\mathbf{N}) \mid Perm(x,y) \& Sorted(y)\}$$

where

$$Perm(x,y) \equiv \forall z \in \mathbf{N}.(\#z\, in\, x) = (\#z\, in\, y)$$

$$\begin{cases} \#z\, in\, \mathbf{nil} = 0 \\ \#z\, in\, \mathbf{cons}(a,s) = \#z\, in\, s + (\mathbf{if}(nateq(a,z), 1, 0) \end{cases}$$

$$\begin{cases} Sorted(\mathbf{nil}) = \mathbf{T} \\ Sorted(cons(a,\mathbf{nil})) = \mathbf{T} \\ Sorted(\mathbf{cons}(a,\mathbf{cons}(b,s))) = (a \le b) \& Sorted(\mathbf{cons}(b,s)) \end{cases}$$

where *nateq* is a boolean function whose value is **true** if its two arguments are equal natural numbers, otherwise the value is **false**. In this example, the propositional function (family of sets) *Sorted* is defined using primitive recursion over lists so we need a way to treat sets as elements in a set.

The universal set $\mathbf{U}_0$ which will be defined here contains codings of the sets formed by the following set forming operations:

$$\bot, \mathbf{T}, \mathbf{N}, +, \mathbf{Eq}, \Pi, \Sigma.$$

Notice that $\mathbf{U}_0$ is not the set of all sets, $\mathbf{U}_0$ has a *fixed* inductive structure while being a set is defined in a more general way (as I did in the first lecture). There is no coding of $\mathbf{U}_0$ in $\mathbf{U}_0$, this would lead to a circularity in the semantical explanation.

When defining the set $\mathbf{U}_0$, I will simultaneously define a family **Set** of sets, $\mathbf{Set}(x)$ set $[x \in \mathbf{U}_0]$ which decodes the elements in $\mathbf{U}_0$ to the set they represent.

$\mathbf{U}_0$-formation:

$$\mathbf{U}_0 \text{ set} \qquad\qquad \frac{x \in \mathbf{U}_0}{\mathbf{Set}(x) \text{ set}}$$

$\mathbf{U}_0$-introduction:

$$\bot' \in \mathbf{U}_0 \qquad\qquad \mathbf{Set}(\bot') = \bot$$

$$\mathbf{T}' \in \mathbf{U}_0 \qquad\qquad \mathbf{Set}(\mathbf{T}') = \mathbf{T}$$

$$\mathbf{N}' \in \mathbf{U}_0 \qquad\qquad \mathbf{Set}(\mathbf{N}') = \mathbf{N}$$

$$\frac{a \in \mathbf{U}_0 \quad b \in \mathbf{U}_0}{+'(a,b) \in \mathbf{U}_0} \qquad\qquad \frac{a \in \mathbf{U}_0 \quad b \in \mathbf{U}_0}{\mathbf{Set}(+'(a,b)) = \mathbf{Set}(a) + \mathbf{Set}(b)}$$

$$\frac{a \in \mathbf{U}_0 \quad b \in \mathbf{Set}(a) \quad c \in \mathbf{Set}(a)}{\mathbf{Eq}'(a,b,c) \in \mathbf{U}_0} \qquad\qquad \frac{a \in \mathbf{U}_0 \quad b \in \mathbf{Set}(a) \quad c \in \mathbf{Set}(a)}{\mathbf{Set}(\mathbf{Eq}'(a,b,c)) = \mathbf{Eq}(\mathbf{Set}(a), b, c)}$$

$$\frac{a \in \mathbf{U}_0 \quad b(x) \in \mathbf{U}_0\,[x \in \mathbf{Set}(a)]}{\Pi'(a,b) \in \mathbf{U}_0} \qquad\qquad \frac{a \in \mathbf{U}_0 \quad b(x) \in \mathbf{U}_0\,[x \in \mathbf{Set}(a)]}{\mathbf{Set}(\Pi'(a,b)) = \Pi(\mathbf{Set}(a), (x)\mathbf{Set}(b(x)))}$$

$$\frac{a \in \mathbf{U}_0 \quad b(x) \in \mathbf{U}_0\,[x \in \mathbf{Set}(a)]}{\Sigma'(a,b) \in \mathbf{U}_0} \qquad\qquad \frac{a \in \mathbf{U}_0 \quad b(x) \in \mathbf{U}_0\,[x \in \mathbf{Set}(a)]}{\mathbf{Set}(\Sigma'(a,b)) = \Sigma(\mathbf{Set}(a), (x)\mathbf{Set}(b(x)))}$$

So there are 7 constructors in $\mathbf{U}_0$: the arity of $\bot'$, $\mathbf{T}'$ and $\mathbf{N}'$ are $\mathbf{0}$, the arity of $+'$ is $\mathbf{0}\twoheadrightarrow\mathbf{0}\twoheadrightarrow\mathbf{0}$, the arity of $\mathbf{Eq}'$ is $\mathbf{0}\twoheadrightarrow\mathbf{0}\twoheadrightarrow\mathbf{0}\twoheadrightarrow\mathbf{0}$ and finally the arity of Π' and Σ' is $\mathbf{0}\twoheadrightarrow(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$. The selector associated with $\mathbf{U}_0$ is **u-rec** of arity:

$$\begin{array}{l} \mathbf{0}\twoheadrightarrow \\ \mathbf{0}\twoheadrightarrow \\ \mathbf{0}\twoheadrightarrow \\ \mathbf{0}\twoheadrightarrow \\ (\mathbf{0}\twoheadrightarrow\mathbf{0}\twoheadrightarrow\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow \\ (\mathbf{0}\twoheadrightarrow\mathbf{0}\twoheadrightarrow\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow \\ (\mathbf{0}\twoheadrightarrow(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}\twoheadrightarrow(\mathbf{0}\twoheadrightarrow\mathbf{0}))\twoheadrightarrow \\ (\mathbf{0}\twoheadrightarrow(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}\twoheadrightarrow(\mathbf{0}\twoheadrightarrow\mathbf{0}))\twoheadrightarrow\mathbf{0} \end{array}$$

The expression $\mathbf{u\text{-}rec}(p, e_1, e_2, e_3, e_4, e_5, e_6, e_7)$ is computed in the following way. I will use the abbreviation $q(x) \equiv \mathbf{u\text{-}rec}(x, e_1, \ldots, e_7)$. First compute p.

1. If the value of p is $\bot'$ then the value of the **u-rec** expression is the value of e_1.
2. if the value of p is $\mathbf{T}'$ then the value is the value of e_2.
3. If the value of p is $\mathbf{N}'$ then the value is the value of e_3.
4. If the value of p is $+'(a,b)$ then the value is the value of $e_4(a, b, q(a), q(b))$.
5. If the value of p is $\mathbf{Eq}'(a,b,c)$ then the value is the value of $e_5(a, b, c, q(a))$.

6. If the value of p is $\Pi'(a,b)$ then the value is the value of $e_6(a,b,q(a), v.q(b(v)))$.
7. If the value of p is $\Sigma'(a,b)$ then the value is the value of $e_7(a,b,q(a), v.q(b(v)))$.

The elimination rule is the following:

$$\frac{\begin{array}{r@{}l} p \in{} & \mathbf{U}_0 \\ e_1 \in{} & C(\bot') \\ e_2 \in{} & C(\mathbf{T}') \\ e_3 \in{} & C(\mathbf{N}') \\ e_4(x,y,z,u) \in{} & C(+'(x,y))\,[x \in \mathbf{U}_0, y \in \mathbf{U}_0, z \in C(x), u \in C(y)] \\ e_5(x,y,z,u) \in{} & C(\mathbf{Eq}'(x,y,z))\,[x \in \mathbf{U}_0, y \in \mathbf{Set}(x), z \in \mathbf{Set}(x), u \in C(x)] \\ e_6(x,y,z,u) \in{} & C(\Pi'(x,y))\,[x \in \mathbf{U}_0, y(v) \in \mathbf{U}_0\,[v \in \mathbf{Set}(x)], z \in C(x), u(v) \in C(y(v))\,[v \in \mathbf{Set}(x)]] \\ e_7(x,y,z,u) \in{} & C(\Sigma'(x,y))\,[x \in \mathbf{U}_0, y(v) \in \mathbf{U}_0\,[v \in \mathbf{Set}(x)], z \in C(x), u(v) \in C(y(v))\,[v \in \mathbf{Set}(x)]] \end{array}}{\mathbf{u\text{-}rec}(p, e_1, e_2, e_3, e_4, e_5, e_6, e_7) \in C(p)}$$

There are also 7 equality rules, one for each constructor in $\mathbf{U}_0$.

It is now possible to define new types using conditionals and recursion. For instance, if $a, b \in \mathbf{U}_0$ then the disjoint union of $\mathbf{Set}(a)$ and $\mathbf{Set}(b)$ can be introduced by the following abbreviations:

$$\begin{aligned} a+b &\equiv \Sigma x \in Bool.\ Setif(x,a,b) \\ \mathbf{inl}(c) &\equiv <\mathbf{true}, c> \\ \mathbf{inr}(d) &\equiv <\mathbf{false}, d> \\ \mathbf{when}(p,d,e) &\equiv \mathbf{if}(\mathbf{fst}(p), d(\mathbf{snd}(p)), e(\mathbf{snd}(p))) \end{aligned}$$

where

$$\begin{aligned} Setif(x,y,z) &\equiv \mathbf{Set}(\mathbf{if}(x,y,z)) \\ \mathbf{fst}(p) &\equiv \mathbf{split}(p, x.y.x) \\ \mathbf{snd}(p) &\equiv \mathbf{split}(p, x.y.y) \end{aligned}$$

IV. REFERENCES

[1] E. Bishop, "Mathematics as a Numerical Language", pp. 53-71 in *Intuitionism and Proof Theory*, ed. Myhill, Kino, Vesley, North Holland, Amsterdam (1970).

[2] R. L. Constable, "Constructive Mathematics and Automatic Program Writers", pp. 229-233 in *Proceedings of IFIP Congress*, North-Holland, Ljubljana (1971).

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

[4] C. A. Goad, *Computational Uses of the Manipulation of Formal Proofs*, Computer Science Department, Stanford University, Stanford (August 1980), Ph. D. thesis.

[5] S. Goto, "Program Synthesis from Natural Deduction Proofs", *IJCAI*, Tokyo, 1979.

[6] A. Heyting, *Intuitionism, an Introduction*, North-Holland (1956).

[7] Kolmogorov, "Zur Deutung der intuitionistischen Logik", *Mathematische Zeitschrift*, Vol. 35, pp. 58-65 (1932).

[8] P. Martin-Löf, "Constructive Mathematics and Computer Programming", pp. 153-175 in *Logic, Methodology and Philosophy of Science, VI*, North-Holland Publishing Company, Amsterdam (1982), Proceedings of the 6th International Congress, Hannover, 1979.

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

[10] B. Nordström, "Programming in Constructive Set Theory — Some Examples" in *Proceedings 1981 Conference on Functional Languages and Computer Architecture*, Wentworth-by-the-Sea, Portsmouth, New Hampshire (October 1981).

[11] B. Nordström and K. Petersson, "Types and Specifications", pp. 915-920 in *Proceedings IFIP '83, Paris*, ed. R. E. A. Mason, Elsevier Science Publishers (North-Holland), Amsterdam (1983).

[12] B. Nordström and J. M. Smith , "Propositions, Types and Specifications of Programs in Martin-Löf's Type Theory", *BIT*, Vol. 24 no. 3, pp. 288-301 (October 1984).

[13] M. Sato, "Toward of a Mathematical Theory of Program Synthesis", *IJCAI*, Tokyo, 1979.

[14] J. M. Smith, "The Identification of Propositions and Types in Martin-Löf's Type Theory — A Programming Example" in *International Conference on Foundations of Computation Theory*, Vol. 158, Springer-Verlag, Borgholm, Sweden (August 21-27, 1983).

[15] S. Takasu, "Proofs and Programs", *Proceedings of the 3:rd IBM Symposium on Mathematical Foundations of Computer Science*, IBM, Japan, 1978.

[16] R. L. Constable et al, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, Englewood Cliffs, New Jersey (1986).

Appendix A. Some Set-Forming Operations in Type Theory

Set	Canonical elements	Computation rule for the selector
$\perp$	$\perp = \{\}$	There is no computation rule for $\mathbf{case}_0$
T	$\mathbf{T} = \{tt\}$	$\mathbf{case}_1(p,b) \Rightarrow q$, if $p \Rightarrow \mathbf{tt}$ and $b \Rightarrow q$
Bool	$\mathbf{Bool} = \{\mathbf{true}, \mathbf{false}\}$	$\mathbf{if}(p,d,e) \Rightarrow q$, if $p \Rightarrow \mathbf{true}$ and $d \Rightarrow q$ $\mathbf{if}(p,d,e) \Rightarrow q$, if $p \Rightarrow \mathbf{false}$ and $e \Rightarrow q$
N	$\mathbf{N} = \{0, \mathrm{succ}(a) \mid a \in \mathbf{N}\}$	$\mathbf{rec}(p,d,e) \Rightarrow q$, if $p \Rightarrow 0$ and $d \Rightarrow q$ $\mathbf{rec}(p,d,e) \Rightarrow q$, if $p \Rightarrow \mathrm{succ}(a)$ and $e(a, \mathrm{rec}(a,d,e)) \Rightarrow q$
+	$A+B = \{\mathbf{inl}(a), \mathbf{inr}(b) \mid a \in A, b \in B\}$	$\mathbf{when}(p,d,e) \Rightarrow q$, if $p \Rightarrow \mathbf{inl}(a)$ and $d(a) \Rightarrow q$ $\mathbf{when}(p,d,e) \Rightarrow q$, if $p \Rightarrow \mathbf{inr}(b)$ and $e(b) \Rightarrow q$
Π	$\Pi(A,B) = \{\lambda(b) \mid b(x) \in B(x)[x \in A]\}$	$\mathbf{funsplit}(p,d) \Rightarrow q$, if $p \Rightarrow \lambda(b) \Rightarrow q$
Σ	$\Sigma(A,B) = \{\mathbf{pair}(a,b) \mid a \in A, b \in B(a)\}$	$\mathbf{split}(p,d) \Rightarrow q$, if $p \Rightarrow \mathbf{pair}(a,b)$ and $d(a,b) \Rightarrow q$
Eq	$\mathbf{Eq}(A,a,b) = \{\mathbf{eq} \mid a = b \in A\}$	

I have used the notation $p \Rightarrow q$ for "the value of p is q".

Defined sets

$$
\begin{aligned}
\Pi x \in A.B(x) &\equiv \Pi(A,B) \\
A \to B &\equiv \Pi(A,(x)B)\text{, if } x \text{ is not free in } B \\
\Sigma x \in A.B(x) &\equiv \Sigma(A,B) \\
A \times B &\equiv \Sigma(A,(x)B)\text{, if } x \text{ is not free in } B
\end{aligned}
$$

Defined propositions

$$
\begin{aligned}
\forall x \in A.B(x) &\equiv \Pi x \in A.B(x) \\
A \supset B &\equiv A \to B \\
\neg A &\equiv A \supset \perp
\end{aligned}
$$

$$
\begin{aligned}
\exists x \in A.B(x) &\equiv \Sigma x \in A.B(x) \\
A \& B &\equiv A \times B \\
A \vee B &\equiv A + B
\end{aligned}
$$