A study of Landau's Grundlagen der Analysis
and
AUTOMATH

Wrocław, November 23d, 2007

Hans de Nivelle

'Bitte vergiss alles, was Du auf der Schule gelernt hast; denn Du hast es nicht gelernt'

(I give you some time to forget)

Why this Talk?

- In the 'Grundlagen der Analysis' (Foundations of Analysis), of 1929, Edmund Landau introduces the natural numbers, the integers, the rational numbers and the reals, from Peano arithmetic. (and some implicit logical assumptions)

- In Theorem 4, addition is introduced, and it is shown that this operation is well-defined. (using the axioms $x + 0 = x$ and $x + \mathrm{succ}(y) = \mathrm{succ}(x + y)$.)

- The definition of $+$ is a recursive function definition using the inductive type defined by $\mathrm{Nat} = 0 : \mathrm{Nat} \mid \mathrm{succ} : \mathrm{Nat} \rightarrow \mathrm{Nat}$.

  The possibility of recursive definitions is based on the fact that the datastructure is freely generated.

  However, Landau does make not use of this fact when he introduces $+$.

Why this Talk (2)?

- Automath is an artificial language that was designed by N.G. de Bruijn around 1967. During the following ten years, several dialects of it were defined and implemented. In 1977, the complete Grundlagen were formalized in AUT-QE.
- Since the verifier was implemented in a low level language, the verifier disappeared with the computer.
- But fortunately, the text was not lost. In 1992, Freek Wiedijk, managed to recover the text of the Grundlagen. In (probably) 1997, he reimplemented a checker in C, which can be downloaded.

Reconstruction of Satz 1-4

- I try to follow the reasoning of Landau, but try to make reasoning precise.
- Replace Nat $= \{1, 2, 3, 4, \ldots\}$ by Nat $= \{0, 1, 2, 3, 4, \ldots\}$.
- Use λ-calculus and higher-order logic, instead of set theory.

Assumptions

- Typed higer-order logic with built-equality, with standard equality rules, and the usual reductions.
- Function introduction:

$$(\ \forall x : X \ \exists! y : Y \ \ P(x, y) \ ) \rightarrow \exists f : X \rightarrow Y \ \ \forall x : X \ \ \ P(x, f(x)).$$

- Function extensionality:

$$\forall f_1, f_2 : X \rightarrow Y \ \ (\ \forall x : X \quad f_1(x) = f_2(x) \ ) \rightarrow f_1 = f_2.$$

- The type Nat, the symbol $0 : \mathrm{Nat}$, and the symbol $\mathrm{succ} : \mathrm{Nat} \rightarrow \mathrm{Nat}$.

Assumptions (2)

- The Peano axioms:

$$\forall x : \mathrm{Nat}\ \mathrm{succ}(x) \neq 0.$$

$$\forall x, y : \mathrm{Nat}\ \mathrm{succ}(x) = \mathrm{succ}(y) \rightarrow x = y.$$

$$\forall P : \mathrm{Nat} \rightarrow \mathrm{Prop}\ P(0) \wedge (\ \forall n : \mathrm{Nat}\ P(n) \rightarrow P(\mathrm{succ}(n))\ )$$
$$\rightarrow \forall x : \mathrm{Nat}\ P(x).$$

Satz 1

$$\forall x, y : \text{Nat } x \neq y \rightarrow \text{succ}(x) \neq \text{succ}(y).$$

Proof: Contraposition of axiom.

Satz 2

$$\forall x : \text{Nat succ}(x) \neq x.$$

Proof:

Induction Base:

$\text{succ}(0) \neq 0$ follows from axiom. Induction Step: Assume $\text{succ}(x) \neq x$. From Satz 1 follows $\text{succ}(\text{succ}(x)) \neq \text{succ}(x)$.

Sat 3A

$$\forall x : \mathrm{Nat}\ x \not\approx 0 \rightarrow \exists y : \mathrm{Nat}\ \mathrm{succ}(y) = x.$$

Proof: Induction.

Satz 3B

$$\forall x : \mathrm{Nat}\ x \not\approx 0 \rightarrow$$

$$\forall y_1, y_2 : \mathrm{Nat}\ \mathrm{succ}(y_1) = x \rightarrow \mathrm{succ}(y_2) = x \rightarrow y_1 = y_2.$$

Proof: Follows from axiom.

Satz 4 (zugleich Definition 1)

Auf genau eine Art lässt sich jedem Zahlenpaar x, y eine natürliche Zahl, $x + y$ genannt ($+$ sprich: plus), so zuordnen dass,

1. $x + 1 = x'$ für jedes x.

2. $x + y' = (x + y)'$, für jedes x und jedes y.

$x + y$ heisst die Summe von x und y oder die durch Addition von y zu x entstehende Zahl.

Satz 4 (at the same time Definition 1)

(In exactly one way it is possible to assign to every pair of numbers x, y, a natural number called $x + y$, (pronounce 'plus') such that

1. $x + 1 = x'$ for every x.
2. $x + y' = (x + y)'$ for every x and every y.

$x + y$ is called the sum of x and y, or the number that is obtained through addition of y to x.)

For $f : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}$, let $\Pi(f)$ denote

$$\forall x : \mathrm{Nat}\ f(x, 0) = x \wedge \forall x, y : \mathrm{Nat}\ f(x, \mathrm{succ}(y)) = \mathrm{succ}(f(x, y)).$$

Then Satz 4A $= \exists f : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}\ \Pi(f)$,

Satz 4B $= \forall f_1, f_2 : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}\ \ \Pi(f_1) \to \Pi(f_2) \to f_1 = f_2$.

Zunächst zeigen wir dass es bei jedem festen x höchstens eine Möglichkeit gibt, $x+y$ für alle y so zu definieren dass

$$x+1=x',$$

und

$$x+y'=(x+y)' \text{ für jedes } y.$$

First we show that for every fixed x, there is at most one possibility to define $x+y$ for all y, such that

$$x+1=x',$$

and

$$x+y'=(x+y)' \text{ for every } y.$$

For $x : \mathrm{Nat}$ and $f : \mathrm{Nat} \to \mathrm{Nat}$, let

$$\Sigma(x, f) := \quad f(0) = x \wedge \forall y : \mathrm{Nat} \quad f(\mathrm{succ}(y)) = \mathrm{succ}(f(y)).$$

First we show:

$$\forall x : \mathrm{Nat} \quad \forall A, B : \mathrm{Nat} \to \mathrm{Nat} \quad \Sigma(x, A) \wedge \Sigma(x, B) \to A = B.$$

We assumed extensionality, so we show:

$$\forall x : \mathrm{Nat} \quad \forall A, B : \mathrm{Nat} \to \mathrm{Nat},$$

$$\Sigma(x, A) \wedge \Sigma(x, B) \to \forall y : \mathrm{Nat}\ A(y) = B(y).$$

We fix x, A, B, assume $\Sigma(x, A), \Sigma(x, B)$ and use induction on $y :$

- Obviously $A(x) = x$ and $B(x) = x$.
- Assume $A(y) = B(y)$. Then
  $A(\mathrm{succ}(y)) = \mathrm{succ}(A(y)) = \mathrm{succ}(B(y)) = B(\mathrm{succ}(y))$.

Now comes the mysterious part:

Wir zeigen jetzt, dass es zu jedem x eine Möglichkeit gibt, $x + y$ für alle y so zu definieren, dass

$$x + 1 = x',$$

und

$$x + y' = (x + y)' \text{ für jedes } y.$$

We now show that there exists for every x, a possibility to define $x + y$ for every y, such that

$$x + 1 = x',$$

and

$$x + y' = (x + y)' \text{ for every } y.$$

Remember that, with parameters $x : \mathrm{Nat}$ and $f : \mathrm{Nat} \to \mathrm{Nat}$,

$$\Sigma(x, f) := \quad f(0) = x \wedge \forall y : \mathrm{Nat} \quad f(\mathrm{succ}(y)) = \mathrm{succ}(f(y)).$$

We show that $\forall x : \mathrm{Nat} \quad \exists f : \mathrm{Nat} \to \mathrm{Nat}, \quad \Sigma(x, f)$.

The proof uses induction on x :

- For 0, take $f = \lambda x : \mathrm{Nat}\ x$.
- Assume for x, there is an f, s.t.

  $$f(0) = x, \text{ and } \forall y : \mathrm{Nat} \quad f(\mathrm{succ}(y)) = \mathrm{succ}(f(y)).$$

  Then for $\mathrm{succ}(x)$ take $g = \lambda y : \mathrm{Nat}\ \mathrm{succ}(f(y))$. We show that $\Sigma(\mathrm{succ}(x), g)$:

  $g(0) = \mathrm{succ}(f(0)) = \mathrm{succ}(x)$.

  $g(\mathrm{succ}(y)) = \mathrm{succ}(f(\mathrm{succ}(y))) = \mathrm{succ}(\mathrm{succ}(f(y)) = \mathrm{succ}(g(y))$.

Same part of the existence proof for multiplication

With parameters $x : \mathrm{Nat}$ and $f : \mathrm{Nat} \to \mathrm{Nat}$,

$$\Sigma'(x, f) := \quad f(0) = 0 \wedge \forall y : \mathrm{Nat} \quad f(\mathrm{succ}(y)) = f(y) + x.$$

We show that $\forall x : \mathrm{Nat} \quad \exists f : \mathrm{Nat} \to \mathrm{Nat}, \quad \Sigma'(x, f)$.

This proof also uses induction on x :

- For 0, take $f = \lambda x : \mathrm{Nat}\ 0$.
- Assume for x, there is an f, s.t.

  $$f(0) = 0, \text{ and } \forall y : \mathrm{Nat} \quad f(\mathrm{succ}(y)) = f(y) + x.$$

  Then for $\mathrm{succ}(x)$ take $g = \lambda y : \mathrm{Nat}\ f(y) + y$. We show that $\Sigma'(\mathrm{succ}(x), g)$. First: $g(0) = f(0) + 0 = 0$.

  Second:
  $g(\mathrm{succ}(y)) = f(\mathrm{succ}(y)) + \mathrm{succ}(y) = f(y) + x + \mathrm{succ}(y)$, and
  $g(y) + \mathrm{succ}(x) = f(y) + y + \mathrm{succ}(x) = f(y) + \mathrm{succ}(y) + x$.

Both proofs don’t use the axioms about Nat being a free datatype.

It took me some time to accept that both proofs are actually correct.

Especially, the multiplication proof, I find incomprehensible.

The proofs proceed ’in the wrong direction’: In the case of +, for each succ(x), the function $\lambda y : Nat$ succ(x) $+ y$ is constructed from the function $\lambda y : Nat\ x + y$ used for x. For 0, the function is $\lambda y : Nat\ y$. (One would expect, for each x, a definition by recursion on y)

We are going to look into the AUTOMATH text to see if Van Benthem Jutting did not accidentily use additional principles (e.g. a recursion axiom)

A Short description of AUTOMATH

Automath is a proof checker based on the Curry-Howard isomorphism. Is has no built-in datatypes, or inductive types.

Nearly all lines in an Automath text have the following form:

Proper Definition: $[a_1 : A_1, \ldots, a_p : A_p]\ g := t : T.$

g must be an identifier, which has not been defined before.

The meaning is: In context $a_1 : A_1, \ldots, a_p : A_p$, define g as t, which has type T. (The automath checker checks that t does have the pretended type)

Use of Parameters

The a_i act like function arguments. Let $t_1, \ldots, t_p$ be a sequence of terms. Let Σ be the (parallel) substitution $a_1 := t_1, \ldots, a_p := t_p$

If

$$\vdash t_1 : A_1\Theta,$$

$$[\ t_1 : A_1\Theta\ ] \vdash t_2 : A_2\Theta,$$

$$[\ t_1 : A_1\Theta,\ t_2 : A_2\Theta\ ] \vdash t_3 : A_3\Theta,$$

$$\cdots$$

$$[\ t_1 : A_1\Theta,\ t_2 : A_2\Theta, \ldots, t_{p-1} : T_{p-1}\Theta\ ] \vdash t_p : T_p\Theta,$$

then $g(t_1, \ldots, t_n) : T\Theta$.

Automath distinguishes application of parameters from usual function application. To me, this distinction seems useless, and it causes some problems as well.

(For example, g could have been defined as

$$g := \lambda a_1 : A_1 \cdots a_n : A_p\ t$$

of type

$$\Pi a_1 : A_1 \cdots a_n : A_p\ T.$$

In Automath, this Π-type can be coerced into every type of form

$$\Pi a_i : A_i \cdots a_n : A_p\ T,$$

which sometimes has strange consequences.

Other Automath Constructs

Assumptions are like definitions, but without witness. They have form $[\ a_1 : A_1, \ldots, a_p : A_p\ ]\ g := \mathbf{prim} : T$.

In addition, there is a paragraph mechanism, which is similar to C^{++} namespaces.

$+\mathsf{name}$: Start paragraph with name.

$-\mathsf{name}$: End paragraph with name.

$+*\,\mathsf{name}$: Continue in paragraph with name.

The extended identifier $\mathsf{id}".p1.p2"$ denotes id of paragraph $p2$ which occurs in $p1$. (Modern notation would be $p1 :: p2 :: \mathsf{id}$)

Other Automath Constructs

$< t > p :$ Application of p on t.

$[x : X]t : \lambda x : X \ t$.

An ugly thing: Implicit Contexts

Contexts in Automath are implicit: When a context is declared, it becomes the default context. This means that it will be the context of future definitions until a new context is declared.

Contexts are controlled by the '@'-symbol.

- $'@'[a_1 : A_1, \ldots, a_p : A_p]$. Make $[a_1 : A_1, \ldots, a_p : A_p]$ the default context.
- $a'@'[a_1 : A_1, \ldots, a_p : A_p]$. It must be the case that a has a declaration in the current default context. Let $\Gamma + [a : A]$ be obtained from the default context by deleting all declarations after a.

  Then $\Gamma + [a : A, a_1 : A_1, \ldots, a_p : A_p]$ will be the new default context. [a]

[a] This rule is the main difficulty when reading an Automath text

- $[a_1 : A_1, \ldots, a_p : A_p]$. Let Γ be the present default context. From now on, the default context will be $\Gamma + [a_1 : A_1, \ldots, a_p : A_p]$.

Exercise in Automath Contexts

$@[a_1 : A_1,\ a_2 : A_2]$

$\quad g_1 := t_1 : T_1$

$\quad g_2 := t_2 : T_2$

$@[b : B]$

$\quad g_3 := t_3 : T_3$

$a_2@[c : C]$

$\quad g_4 := t_4 : T_4$

$a_2@$

$\quad g_5 := t_5 : T_5$

$\quad g_6 := t_6 : T_6$

$[b : B]$

$\quad g_7 := t_7 : T_7$

(g_3 has context $[b : B]$) (g_1, g_2, g_5, g_6 have $[a_1 : A_1, a_2 : A_2]$) (g_4 has $[a_1 : A_1, a_2 : A_2, c : C]$) ($g_7$ has $[a_1 : A_1, a_2 : A_2, b : B]$)

Selected Definitions from the Grundlagen

(I have sorted out the contexts, replaced the paragraph notation by namespace notation, removed quotes around prim/set/prop, and filled in implicit arguments)

The beginning of the file

```
[ a:prop, b:prop] imp := [x:a] b prop
   % Defines a -> b as function space.
[ a:prop, b:prop, a1: a, i: imp(a,b) ] mp := <a1> i : b
   % Modus ponens through function application.
[ a:prop ] refimp := [x:a] x : imp(a,a)
   % reflexivity of implication.
[ a:prop, b:prop, c:prop, i : imp(a,b), j:imp(b,c) ]
   trimp := [ x: a ] << x > i > j > : imp(a,c)
   % transitivity of implication.
```

Some Propositional Operators

```
[ ] con := prim : prop
   % contradiction as primitive notion.
[ a : prop ] not := imp( a, con ) : prop
   % In the file, a was an implicit argument.

[ a: prop ] wel := not(not(a)) : prop
[ a: prop, a1 : a ] weli := [ x : not(a) ] <a1> x : wel(a)

[ a: prop, w : wel(a) ] et := prim : a
   % Introduces removal of double negation.

[ ] obvious := imp( con, con ) : prop
[ a : prop, b : prop ] ec := imp( a, not(b)) : prop
[ a : prop, b : prop ] and := not( ec( a, b )) : prop
[ a : prop, b : prop ] or := imp( not(a), b ) : prop
```

Quantifiers in Automath

```
[ sigma : type, p : [ x : sigma ] prop ] all := p : prop
    % This definition is possible because Automath
    % allows coercion from [x:sigma] prop to prop.
    % As a consequence, predicates can have inhabitants.
    % t : P implies <x> t : <x> P.

[ sigma : type, p : [ x : sigma ] prop ]
   non := [ x : sigma ] not( <x> p ) : [ x : sigma ] prop
      % non(sigma,p) means lambda x. not(p(x)), which in
      % Automath not only has type [x:sigma] prop, but
      % also prop.

[ sigma : type, p : [ x : sigma ] prop ]
   some := not( non( p )) : prop.
```

Equality in Automath

```
[ sigma : type, s : sigma, t : sigma ] is := prim : prop
[ sigma : type, s : sigma ] refis := prim : is(s,s)

[ sigma : type, p : [x:sigma] prop, s:sigma, t:sigma,
     sp: <s> p, i : is(s,t) ] isp := prim : <t> p.
   % Equality replacement rule.

[ sigma: type, s:sigma, t:sigma ] [ i : is(s,t) ]
   symis := isp( [x:sigma] is(x,s), s,t, refis(s), i )
                                          : is(t,s)
[ sigma: type, s:sigma, t:sigma, u:sigma,
              i:is(s,t), j:is(t,u) ]
   tris := isp( [x:sigma] is(x,u), t, s, j, symis(i))
                                          : is(s,u)
```

Introduction of New Objects

```
[ sigma: type, p : [x:sigma] prop ]
   amone := [ x :sigma ][ y: sigma ]
            [ u : <x> p ] [ v: <y> p ] is(x,y) : prop
[ sigma: type, p : [x:sigma] prop ]
   one := and( amone(sigma,p), some(sigma,p)) : prop
   % at most one and exactly one.

[ sigma:type, p : [x:sigma] prop ] [ o1: one(sigma,p) ]
   ind := prim : sigma
[ sigma:type, p : [x:sigma] prop ] [ o1: one(sigma,p) ]
   oneax := prim : <ind> p
   % If there is exactly one x, for which p(x) holds, and
   % o1 is a proof of this fact, then ind(sigma, p, o1)
   % is this x. Note that ind has implicit parameters.
```

Functional Extensionality Axiom

```
[ sigma: type, tau: type,
   f: [ x : sigma ] tau, g : [ x : sigma ] tau,
      i : [ x : sigma ] is( tau, <x> f, <x> g ) ]
         fisi := prim : is( [ x : sigma ] tau, f, g )
```

Sets in Automath

```
[ sigma: type ] set := prim : type
[ sigma: type, s: sigma, s0 : set ] esti := prim : prop
   % esti( sigma, s, s0 ) means:   s in s0.

[ sigma: type, p: [ x: sigma ] prop ] setof := prim : set
[ sigma: type, p: [ x: sigma ] prop, s: sigma, sp : <s>p ]
   estii := prim : esti( sigma, s, setof(p))
[ sigma: type, p: [ x: sigma ] prop, s: sigma,
  e: esti( sigma, s, setof(p)) ] estie := prim : <s> p
```

Some of the preliminary definitions

```
[ ] nat := prim : type

% First, some polymorphic operators are instantiated:

[ x : nat, y : nat ] is := e::is( nat, x, y ) : prop
[ x : nat, y : nat ] nis := not( is( x,y )) : prop
[ x : nat, s : set(nat) ] in : estie( nat, x, s ) : prop
[ p : [x:nat] prop ] some := l::some(nat,p) : prop
[ p : [x:nat] prop ] all := l:all(nat,p) : prop
[ p : [x:nat] prop ] one := l:one(nat,p) : prop

% Two prominent inhabitants of nat appear on the scene:

[ ] 1 := prim: nat
[ suc ] := prim [ x : nat ] : nat
```

```
[x : nat] [y:nat] [ i : is(x,y)]
   ax2 := isf( nat, nat, suc, x, y, i ) :
      is( <x> suc, <y> suc )
% (ax2 is vacuous if one uses functional notation)

[ ] ax3 := prim: [x:nat] nis( <x> suc, 1 )
[ ] ax4 := prim: [x:nat] [y:nat]
             [ u : is( <x> suc, <y> suc) ] is(x,y)

[ s : set(nat) ] cond1 := in( 1, s ) : prop
[ s : set(nat) ] cond2 := all( [ x:nat ]
            imp( in(x,s), in( <x> suc, s ))) : prop

[ ] ax5 := prim :
      [ s : set(nat) ] [ u : cond1(s) ] [ v : cond2(s) ]
                        [ x : nat ] in(x, s)
```

Proof of Satz 4

```
[x: nat, f: [ y : nat ] nat ]
   prop1 := all( [ y : nat ] is( <<y> suc > f,
                                 << y > f > suc )) : prop

[x: nat, f: [ y : nat ] nat ]
   prop2 := and( is( <1> f, <x> suc ), prop1 ) : prop

[x: nat, a : [ y : nat ] nat, b : [ y : nat ] nat,
   pa: prop2(a), pb : prop2(b), y : nat ]
      prop3 := is( <y> a, <y> b )

[ x : nat ] prop4 := 1::some( [ y : nat ] nat,
           [ z : [ y : nat ] nat ] prop2(z) ) : prop
```

```
[x: nat] aa := (proof omitted) :
   amone( [ y : nat ] nat,
          [ z : [ y : nat ] nat ] prop2(z) )

[x : nat] bb := (proof omitted) : prop4(x)

[x : nat ] satz4 :=
   onei( [ y : nat ] nat,
         [ z : [ y : nat ] nat ] 24:prop2(z),
            24::aa, 24::bb ) :
      e::one( [ y : nat ] nat,
            [ z : [ y : nat ] nat ]
          and( is( < 1 > z, <x> suc ),
               all( [ y : nat ]
                   is( <<y>succ> z, <<y>z>suc))))
```

```
[x : nat ] plus :=
             ind( [ y : nat ] nat,
                  [ z : [ y : nat ] nat ] 24::prop2(z),
                  satz4 ) : [ y : nat ] nat

% For every x, plus(x) is the unique function f
% that satisfies Sigma(x,f)

[ x : nat, y : nat ] pl := <y> plus(x) : nat

% The proof and the introduction of + are completely
% accurate!
```

Conclusions

- There is nothing wrong in the Grundlagen, but addition becomes only what it is supposed to be in Satz 7 and 8.
- I am quite impressed by the maturity of the AUTOMATH system, especially if you consider how difficult it was to experiment with implementations in those days.
- The reason that addition and multiplication can be defined without referring to free generation, must be the fact that they are definable in non-standard models as well. (A non-standard model is defined by the smallest two numbers n, m for which $s^n(0) = s^m(0)$, and $n \neq m$)

  Are there other meaningful functions that are definable in non-standard models?

  Can they be formally defined in a similar way?

Recommendations for AUTOMATH

Actually, van Benthem Jutting's verification is quite readable, if one ignores the syntactic difficulties:

- The main problem when reading AUTOMATH is finding out in which context a definition is taking place.

  Instead of the linking mechanism with '@', one could have the following command, which has explicit scoping:

  Assume [x1 : t1 , ..., xn : tn] in
  {
  }

- Implicit arguments should be forbidden. (Automath has a rule that allows omission of arguments when they are equal to the parameters at the moment of definition.

- The paragraph mechanism should be replaced by more modern notation using C^{++} style namespaces. (`th4"e.inj"` can be replaced by `e::inj::th4` or `e.inj.th4`)
- Instead of using end-of-line as terminator, '.' or ';' should be used.
- I think that the inclusion $A_1 \rightarrow \cdots \rightarrow A_p \rightarrow \text{Prop} \subseteq A_i \rightarrow \cdots \rightarrow A_p \rightarrow \text{Prop}$ was not a good idea (but for readability, it is not so important, and it is much harder to change this)

Maybe it would be nice to reformat Van-Benthem Jutting's translation. One could obtain a readable result.