Number Theory beyond Frege
On the necessity of open arity

Hannes Hutzelmeyer

Summary

A closer look at mathematical proofs led Gottlob Frege to realize that Aristotle's syllogism logic was not sufficient for many theorems. He developed what today is called first-order predicate logic. It is usually thought that predicate logic is sufficient for the theory of natural numbers. However, this first step of modern logic development again is not sufficient. One needs another step, especially to allow for so-called open arity of arrays. This second step cannot be done in general in object-language based on predicate logic but only by metalanguage. Therefore one needs something like the FUME-method (put forward by the author) which allows for a precise treatment of both language levels. Dot-dot-dot ... is not admissible in predicate logics as it needs some kind of recursion. In metalanguage, however, one has to introduce some basic recursion right from the setup (but it is much weaker than primitive recursion).

For natural numbers two examples are given, one for a concrete version of Robinson arithmetic and one for recursive arithmetic. Without the second step to metalanguage one cannot express some of the most important so-called theorems of number theory in a direct fashion, leave alone prove them. Actually some are not theorems but metatheorems. The examples comprise Chinese remainders, Gödel's beta-function, little Gauss's summing up of numbers, Euclid's unlimited primes and the canonical representation of a natural number (fundamental theorem of natural arithmetic).

After one has included the second step which allows one to talk about open arities in metalanguage one can tackle the problem of talking about number-arrays in object language. One can do this to a certain extent by coding number-arrays by (usually) two numbers. This can be done even in Robinson arithmetic using 'Gödel's beta-function'. But one has to make use of the second step before one can return to object-language. Of course, the introduction of two tiers, i.e. object-language and metalanguage, is necessary for many other areas of mathematics, if not to say, most of them.

Contact: Hutzelmeyer@pai.de

https://pai.de

Copyright
All rights reserved. No reproduction of this publication may be made without written permission.
Any person who does any unauthorized act in relation to this publication may be liable to criminal prosecution and civil claims for damages.
1 Beyond the conventional paradigm of logic of mathematics

It all started in the year of 1879 when Gottlieb Frege put forward his revolutionary 'Begriffsschrift'. Until then the syllogism logic of Aristotle had been considered to be sufficient as the basis of logical reasoning and therefore also of mathematics. Besides the usual logical characters \( \neq \rightarrow \land \lor \leftrightarrow \exists \forall \) and variables like e.g. \( A_1 \) or \( A_{13} \) were introduced together with the rules for omnition \( \forall A_1[ \ldots ] \) and entition \( \exists A_2[ \ldots ] \) as well as relation-constant and function-constant strings that allowed for expressing mathematical sentences in a proper fashion. Freges notation differs from this modern form, but that is irrelevant.

The author was confronted with this status when started studying physics, mathematics and philosophy of science in the year of 1960. For a long time he did not enter the field of number theory, however, he always had a bad feeling about theorems of number theory, that he could not relate to the axiomatic approach to, say Robinson arithmetic. The problem to start with is not the proving of theorems of number theory. The first problem is just to write down sentences that are called theorems of number theory. Mathematicians and logicians have constructed complicated systems of so-called classical and intuitionistic logic, theory of types, axiomatic set theory and so on. But are these methods really sufficient for expressing basic sentences of number theory, leave alone proving them in a purely deductive fashion from basically true sentences or axioms? The author contradicts this question and shows a way out by the FUME-method. He claims that you need both object language and metalanguage being formulated with the rigor of formal logic and some basic recursion thrown in. The examples of section 3 to 7 will hopefully - clarify his reasoning. The problem is called open arity. It is not the only reason for the FUME-method, but it is a particularly striking one. 'Dot-dot-dot' is just not a legitimate language element in a precise language. For a short introduction to the FUME-method download file Snark1.1.pdf from https://pai.de.

Heuristically speaking, sequences consist of some kind of infinitely many constituents with a definite start and no end, with a line-arrangement, one constituent put behind another. An array is a finite ordered collection of constituents with a start and an end (where the constituents are separated by a special character). It has an arity given by a natural number that is the count of its constituents. An example for an array is the alphabet of letters separated by commas 'a, b, ..., z' with arity 26, but also the simple array of zeros separated by semicolons '0;0;0;0;0' is an example with arity 5.

Of course these examples are not satisfactory, one needs a precise description for arrays. The FUME-method will be applied as one obviously needs a language that allows for some recursion. If the constituents are taken from a calcule of the object language Funcia, one has to define arrays in metalinguage Mencish. The systems of Funcia are called calcules by the author, they are not to be confused with various calculus-systems or the calculus of real numbers. There are concrete and abstract calcules.

The font-method is used to distinguish between the various levels of languages: Times New Roman of all styles for normal text in English e.g., Symbol and Arial boldface italics for metalanguage Mencish e.g. number-array\((A_1)\) and normal Symbol and Arial for object language Funcia e.g. \( \forall A_1[(A_1+0)=A_1] \).

The other frontier where usual predicate logic is not sufficient for mathematics is connected with higher than first-order logic. Axiomatic set theory claims that all of infinity mathematics is covered by it. The author, however, has some doubts. Anyhow, the conventional approach to real numbers necessitates second-order logics (for some transcendency axiom, be it Dedekind cuts, interval nesting, Cauchy series or whatever). In group theory second-order is just around the corner, as factors, subgroups, normal subgroups, kernels etc. are not first-order entities.

Mathematicians usually do not even mention that there might be a problem at the foundations. And physicists happily use transcendental mathematics although no one has ever measured anything but a rational number. How about dimensionless constants in physics? Sommerfeld's fine-structure constant, is it a real number and is there a deeper reason for its size. You see, once one is thinking about transcendental numbers, one is entering the field of theology, which shows that the name of this numbers has been chosen perfectly!
2 Metalinguial introduction of number-arrays and more

In metalanguage Mencish there are straightforward metaproperties of strings like \textit{number}, \textit{number-array}, \textit{variable}, \textit{sentence} or \textit{formula} and metafunctions for string-replacement \((A;A[A])\) and character-deletion \((A;A)\), the relevant examples are given in appendix A. One can define \textit{number-array} strings by the simple recursion in metalanguage Mencish

\[
\text{number-array} :: \quad \text{number} \mid \text{number-array} \; \text{; number}
\]

However, one has to find a way to talk about \textit{number-array} strings in FUNCish. This will be possible by coding \textit{number-array} strings by \textit{number} strings. That is what it is all about. The following metadefinitions are a little abbreviated, but straightforward, the necessary recursions are admissible in Mencish. For definiteness it is done for the concrete calculus \textit{ALPHA} of Robinson decimal\(^1\) natural arithmetic (as described in the next section). However, the only feature that is used are the decimal numbers themselves, so that the metadefinitions can be transferred to other concrete arithmetic calulates like e.g. \textit{LAMBDA} of decimal pinion arithmetic (which allows for primitive recursive functions):

\[
A''(A) \text{ (succession)}
\]

\[
\begin{align*}
A''(0) &= 1 \\
A''(1) &= 2 \\
A''(2) &= 3 \\
&\vdots
\end{align*}
\]

\[
A''(A1) = A1 1 \\
A''(A1) = A1 2
\]

\[
A''(A1) = A''(A1) \text{ }0
\]

\[
A''(A1) = A''(A1) \text{ with concatenation}
\]

\[
A(A) \text{ (length)}
\]

\[
A(A1) = 1 \text{ if char}(A2) \quad A(A1) = A''(A(A1)) \text{ with char}(A2)
\]

\[
A(A) \text{ (arity)}
\]

\[
A'(A) = 0 \text{ if not number-array}
\]

\[
A'(A) = 1 \text{ else decimal arity, defined as follows (count of semicolon strings)}:
\]

\[
A'(A1) = ((((((A1 \_0) \_1) \_2) \_3) \_4) \_5) \_6) \_7) \_8) \_9)
\]

\[
A1 \less A2 \text{ (projects array-constituent)}
\]

\[
\begin{align*}
A1 \less A2 &= 0 \text{ if A1 not number-array or if A2 not number} \\
A1 \less A2 &= \text{but not less than } A00(A1)
\end{align*}
\]

\[
A \less A \text{ (number)}
\]

\[
A \less A \text{ else constituent at position 3) A2, recursively defined as follows:}
\]

\[
A1 \less A2 \quad \text{if A1 or A2 are not number number strings}
\]

\[
A1 \less A2 \quad \text{if number strings, recursively defined as follows:}
\]

\[
\begin{align*}
A1 \less A2 &= (A1 \_A1 \_A2) \to [A1 \less A2] (A2) \\
A1 \less A2 &= \neg (A1 \less A1) [A1 \less A2] \to \neg [A1 \less A2]
\end{align*}
\]

\[
\forall A1 \forall A2[ [ [ \text{number-array}(A1)] \wedge [\text{number}(A2)] ] \wedge [A2 \less A00(A1)] ] \to
\]

\[
[ [A2 = 0] \wedge [A \less A1; A2 = A1] ] \lor [ [0 \less A2] \wedge [\text{number}(A \less A1; A2)] ] \wedge
\]

\[
\exists A3 \exists A4[ [ [ \text{number-array}(A3) ] \wedge [A00(A3) = A2 ] ] \wedge
\]

\[
[ [A1 = A3; A \less A1; A2 = A1] ] \lor [ [A1 = A3; A \less A1; A2 = A2] ] ]
\]

And one defines \textit{distinct-variable-array} and \textit{omni} strings with a little more complicated recursion using binary metarelation \( \rightarrow A \to A \), that states that string \( A1 \) is suitably containing string \( A2 \):

\[
\]

\[
\textit{omni} :: \quad \forall \text{ variable } [ \text{ omni } \forall \text{ variable } [ \neg [ \text{ variable } ] ]]
\]

\[
\forall A1[ [ \text{omni}(A1) ] \leftrightarrow [ [ \text{omni}(A1) ] \wedge [ \text{distinct-variable-array}(( (A1; [\forall f; ] \partial [ \partial [ \partial ] ) ] ]]
\]

\(^1\) using decimal numbers is just for convenience

\(^2\) \(A''(A), A (A), A(A), A(A) \), \(A A A \), \text{ with double symbols defined with decimal numbers correspond to general } \(A'(A)\), \(A(A)\), \(A(A)\), \(A(A) \), \text{ with double symbols defined with petit numbers}

\(^3\) an array has \textit{constituents, place} numbers constituent from left 1 to arity \( a \), \textit{position} numbers from 0 to \( a-1 \)
3 Robinson numbers arithmetic and Gödel's beta-function

In the following the concrete calcules **ALPHA** of Robinson arithmetic and **LAMBDA** of pinition arithmetic this will be investigated with respect to arrays. One cannot directly talk about **number-array** strings of unspecified arity within Funcish as one cannot express it e.g. in **ALPHA** with dot-dot-dot and one cannot name a variable **A** so that the arity is properly represented: \( \forall A_1[\forall A_2[...[\forall A_2[...]]]] \)

Concrete calcule **ALPHA** of Robinson decimal natural arithmetic uses the following alphabet which is not the shortest one, but it is tried keep as close to conventional logic language as possible:

<table>
<thead>
<tr>
<th>Aarial 8, peti-number for variables</th>
<th>Aarial 12, normal size numbers for decimal individuals</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

Symbol 12, general logic symbols, special calcule symbols

The ontological basis of concrete calcules **ALPHA** of decimal Robinson arithmetic consists of **decimal-number** strings (0 1 2 ...), unary succession **function-constant** \( A' \), binary addition **function-constant** \( A+A \), binary multiplication **function-constant** \( A\times A \) and binary minority **relation-constant** \( A<A \). The start of derivations of **THEOREM** strings is given by so-called **Basiom** strings (corresponding to **Axiom** strings of abstract calcules). In the usual fashion there are:

- Start-existence, injectivity, unary and multary induction of succession \( A' \).
- Right zero and right iteration of addition \( (A+A) \)
- Right zero and right iteration of multiplication \( (A\times A) \)
- Diagonal succession, iteration succession, non-reflexivity and antisymmetry of minority \( A<A \).

The so-called 'chinese remainder theorem' is actually a **metatheorem**; it is necessary for Gödel's beta-function; both necessitate open arities.

Chinese remainder **metatheorem** : if the constituents of a **number-array** \( A_2 \) of arity \( A_1 \) are pairwise coprime and if they are larger than the corresponding constituents of a **number-array** \( A_3 \) of same arity, then there is exactly one **number** \( A_{10} \) (less than the product of the constituents of \( A_2 \)) such that every constituent of \( A_3 \) is obtained as remainder of the division of \( A_{10} \) by the corresponding one of \( A_2 \). This flowery wording has to be translated into precise metalanguage\(^1\). Some string manipulations of section 2 are needed: **relation-constant** \( A << A \) and **function-constant** \( A00(A) \) \( AVV(A;A) \) \( (A;A/A) \) and \( (A\overline{A}A) \).

\[
\forall A_1[\forall A_2[\forall A_3[[[[[[[number(A_1)]]}\land[1<<A_1]]\land[number-array(A_2)]]]]\land
[number-array(A_3)]\land[A00(A_2)=A_1]\land[A00(A_3)=A_1]]\land

\forall A_4[\forall A_5[[[[[[number(A_4)]]\land[number(A_5)]]\land[number(A_6)]]\land[A_4<<A_1]]\land
[A_5=AVV(A_2;A_4)]]\land[A_6=AVV(A_3;A_4)]]\land[[[[1<<A_5]]\land[A_6<<A_5]]]\land

\forall A_7[\forall A_8[[[[[number(A_7)]]\land[number(A_8)]]\land[A_7<<A_4]]\land[A_8=AVV(A_2;A_7)]]\rightarrow

\forall A_9[[[number(A_9)]]\land[TRUTH(парwise coprime

\forall A_9([A_9=((((\forall A_0(0)\not=0)\not=1)\not=2)\not=3)\not=4)\not=5)\not=6)\not=7)\not=8)\not=9);[F(\not=1\times (A_2;A_5)];xF))]\rightarrow

\forall A_1[\forall A_2[\forall A_3[[[A_3=AVV(A_2;A_3)]]\land[A_3=A_1]]\rightarrow[A_1=1]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]}]

Obviously there is no chance to write this down in object-language! The 'Chinese remainder' is not a **THEOREM** of calcule **ALPHA** but a **metatheorem** of its metacalcule **ALPHA**.

---

\(^1\) Both, object-language Funcish and metalanguage Mencish obey the so-called 'Calculation Criterion of Truth': a computer can decide if a certain step of reasoning is in accordance with the rules.

version 1.0 Open Arity 4
In conventional notation: Gödel's beta-function $\text{g}(x,y,z) = \text{divrem}(x,y(z+1)+1)$ with the division remainder function allows for coding an array of numbers with arity $a$ by two codes $x$ and $y$ with positions $z$ from 0 to $a-1$ or places from 1 to $a$. Just like above: the so-called 'Gödel's beta-function theorem' is actually a metatheorem.

Gödel's beta-function metatheorem: a number-array $A_5$ of arity $A_4$ can be coded by two number strings $A_1$ and $A_2$ such that every constituent of the number-array can be obtained using a suitable ternary UNEX-formulo $\text{AXFOgbeta}$ that represents Gödel's beta-function in calulate $\text{ALPH}$ and that has free variable strings $A_0$ for result, $A_1$, $A_2$ as codes and $A_3$ as position inside the array, $A_3 < A_1$.

$$\forall A_4[ \forall A_5[ [[ [\text{number}(A_4)] \land [\text{number-array}(A_5)] \land [A_4 = A_0(A_5)] ] \land [ \exists A_1[ \exists A_2[ [\text{number}(A_1)] \land [\text{number}(A_2)] ] \land [ \forall A_3[ [\text{number}(A_3)] \land [A_3 < A_4] ] \land [\text{number}(A_6)] ] \land [A_6 = A_1 \lor A_2 \lor A_3 \lor A_0 / A_5 ]] ] ] ] ] ]$$

It is proven by taking

$$\text{AXFOgbeta} = \exists A_20[[((A_2\times A_3')\times A_20)+A_0]=A_1]\land[A_0<(A_2\times A_3')]$$

and applying the Chinese remainder metatheorem. The auxiliary bound variable $A_20$ is chosen such that it does not easily collide with free variable strings when the $\text{AXFOgbeta}$ is inserted in a phrase string; obviously $A_20$ is limited by $A_1$.

Based on Gödel's beta-function metatheorem one can talk about number-array strings of any arity in the following way within concrete calcule $\text{ALPH}$ of decimal Robinson arithmetic. Interpret variable $A_4$ as arity, $A_1$ and $A_2$ as codes, $A_3$ as position within number-array and $A_0$ as unique result:

$$\forall A_1[ \forall A_2[ \forall A_3[ \forall A_4[[[0<A_4] \land [A_3<A_4]] \rightarrow [\forall A_0[[\text{AXFOgbeta}] \rightarrow [\ldots ]]])]]]]$$

If one does not like the idea of two code number strings one can combine them into one number by so-called anti-diagonal pair coding that also can be represented in calcule $\text{ALPH}$ , conventionally written as pair of row and column $p=\text{adr}(p)\times\text{adr}(p)+\text{adr}(p+1))/2$ and its inverse functions for row $j=\text{adr}(p)=p-(\text{adr}(p)\times\text{adr}(p)+1))/2$ and for column $k=\text{adr}(p)=(\text{adr}(p)+1)\times\text{adr}(p+1))/2-(p+1)$ with corresponding UNEX-formulo strings, including auxiliary function $\text{adr}(p)=(\text{brt}(8\times t+1)-1)/2$ with entire square-root function $\text{brt}(n)$. Five more extra-individual-constant strings with bound variable strings that do not collide in the following applications (see binary metarelation $A \sim A$ of appendix A).

binary UNEX-formulo : for antidiagional pair

$$\text{AXFOadp} = (A_0+A_0)=(A_1+(A_1+A_2)\times(A_1+A_2'))$$ simple, necessary for bisection

unary UNEX-formulo: for entire square root, anti-diagonal auxiliary, row and column

$$\text{AXFObrt} = \left( (A_0\times A_0)=A_1 \right) \lor \left( \exists A_3[ (A_0\times A_0)\times A_0 (A_1+A_2)] \land [A_3[(A_1+A_2)\times (A_0+A_0)]] \right)$$

$$\text{AXFOada} = \left( (A_1+A_0)\times (A_0+A_0') = (8\times A_1) \right) \lor [\left( \exists A_3[ (A_0+A_0)\times (A_0+A_0')\times (A_1+A_2)] \land [A_3[(A_1+A_2)\times (A_0+A_0')]] \right)$$

$$\text{AXFOadr} = \exists A_3[ (\text{AXFOada}; A_0 \times A_3) \land [(A_0+A_0)+(A_1+A_2)] = (A_3(A_1+A_3))]$$

$$\text{AXFOadc} = \exists A_3[ (\text{AXFOada}; A_0 \times A_3) \land [(A_0+A_0)+(A_1+A_1') = (A_3(A_1+A_3'))]]$$

Inserting this properly in $\text{AXFOgbeta}$ gives the desired (but somewhat lengthy) result.

1) $\text{AXFOgbeta}$ is an extra-individual-constant that is used like a makro in programming languages, just a name for a string that is to be expanded wherever it appears (one has to take care that no collision of bound variable strings appear).
4 Recursive natural numbers arithmetic

The choice for a concrete calcule of recursive natural arithmetic is the concrete calcule LAMBDA of decimal primitive arithmetic. It uses the following alphabet which is not the shortest possible one, but it is tried keep as close to conventional logic language as possible:

<table>
<thead>
<tr>
<th>Arial 8, petit-number for variables</th>
<th>Arial 12, normal size numbers for decimal individuals</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Symbol 12, general logic symbols,</td>
<td>special calcule symbols</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
</tbody>
</table>

List of 38 (plus 1 extra) characters for ontological basis of calcule LAMBDA

|sort :: | λ |
sort-array :: | sort ; sort-array ; sort |
decimal :: number :: | 0 ; 1 ; 2 ; ... |
correct definition see section 5 |
basis-ingredient :: | sort ; decimal ; basis-function-constant ; basis-relation-constant |
basis-function-constant :: | λ( ) ; λ(sort-array) ; (λ*λ) |
pinon strings are natural numbers that code primitive recursive functions, when they replace λ in basis-function-constant string λ( ) or λ(sort-array) resp. ; 0 codes the zero function, 1 codes succession. The third case of pinon codes straight recursion, where the left pinon of intrinsic arity m gives the initial value and the right pinon of intrinsic arity n gives the iteration function (the intrinsic arity of the new pinon is max(m+1,n-1)). The last case 8 pinon pinon-catena 9 codes composition of functions with any intrinsic arity: the left pinon is the function where the pinon strings of the pinon-array are plugged in. The PINITOR calculator that does the calculating is not described here, neither the basic true sentences, that include a schema of sentences (or as the author prefers to call it a mater of sentences) meaning that they are enumerably infinite many (by the way: for a proper introduction of sentence schemata one has to use metalanguage).

The basis-function-constant (λ*λ) gives the decimal synaption of two strings, which is basically concatenation, except that no leading 0 is admissible. Actually the definition among the basis-ingredient strings is redundant, as it can be given by a pinon. The same is true for basis-relation-constant #A and A≤A as they can be defined using pinon Apiny and Aemiy resp. as codes of characteristic functions.

Primitive recursive functions are obtained by pinon strings, these precede as codes the basis-function-constant strings λ( ) and λ(sort-array) . If a number is not a pinon string the primitive function with this code is simply put to 0 for all input.. Many examples are given in the publication ‘Programming primitive recursive functions and beyond’ that can be downloaded as file C6-C7-Pinon.pdf on the homepage https://pai.de of the author. Very few examples for coding of primitive recursive functions by decimal numbers are given here:

It is a funny observation that pinon functions have a Janus face. They have been designed to represent primitive recursive functions, e.g.

22011(λ1;λ2) the addition of two numbers with pinon λadd=22011 e.g. 22011(1;1)=2

But the following is defined too and gives a funny function:

λt(0) the value for all codes at 0 where the result is put to 0 if λt is not a pinon code.

By the way: it will turn out that one can talk about number-array strings within LAMBDA ; however, this calcule has the shortcoming that it necessitates enumerably many basis-function-constant strings, as there is no limit on the arity for the sort-array strings of primitive recursive functions.
The strange functions that can be obtained by putting variables into code position can be generalized to so-called **processive** functions. Composition of functions produces so-called **scheme** strings (not to confuse with **schemata** (or matres) of sentences, conventionally they are called 'general terms'). One realizes that **scheme** strings are obtained from **function-constant** strings by inserting **number** and **variable** strings and compositions thereof represent functions. The world of processive functions is very rich, e.g. it comprises straightforwardly **Ackermann function** and other **hyperexponentiations**.

There is a straightforward way in calculc **LAMBDA** to talk about **number-array** strings $A_3$ of arity given by **number** $A_1$. They can be represented by code **number** string $A_2$, is expressed by the metatheorem:

$$\forall A_3 \forall A_4 [ [[[ \text{number}(A_3) ] \land [0 << A_3 ]] \land \text{number-array}(A_4) ] \land [\text{LAMBDA}(A_4) = A_3] ] \rightarrow [ \exists A_1 [ \text{number}(A_1) ] \land [ \forall A_2 [ [[[ \text{number}(A_2) ] \land [A_2 << A_3 ]] \land \text{number}(A_5) ] \land \lambda_\text{true}(\lambda_\text{true}(A_4; A_3)) ] ] ] ] ] ]]

The proof is quite trivial, one can program a unary primitive recursive function, given any finite count of values of a given arity for the low end of the value table.

Based on this **metatheorem** one can talk about **number-array** strings of any arity in the following way within concrete calculc **LAMBDA** of decimal positive arithmetic:

$$\forall A_1 \forall A_2 [ \forall A_3 [[0 < A_3] \land [A_2 < A_3]] ] \rightarrow [ \ldots \lambda_1(A_2) \ldots ] ] ]]

As opposed to the preceding section one can talk about the constituent of an **number-array** string in a direct way. The reason for this is that concrete calculc **LAMBDA** allows for primitive recursion and one does not have to take refuge to representation of functions using Gödel's beta-function technique.

But still one has to go the detour in metalanguage in order to correctly refer to **number-array** strings as one can only express in metalanguage what is meant by a **number-array** string.

### 5 Little Gauss's theorem

Everybody knows the anectode of **little Gauss** reinventing the method of summing up numbers that was found by Indian mathematician Aryabhata in 499 AD: conventionally written with dot-dot-dot:

\[ (1+2+3+4+\ldots+n)=n(n+1)/2 \]

How to express it in connection with concrete calculc **ALPHA** of decimal Robinson arithmetic? And another question is, how to prove it? It is not a **THEOREM** but a schema (or as the author prefers to call it 'mater') of **THEOREM** strings. Therefore it has to be expressed differently:

a) **metatheorem** of Little Gauss

that is producing successively the trivial **THEOREM** strings:

\[
\begin{align*}
(2\times(1+2)) &= (2\times(2+1)) \\
(2\times((1+2)+3)) &= (3\times(3+1)) \\
(2\times(((1+2)+3)+4)) &= (4\times(4+1)) \\
\end{align*}
\]

\[
\forall A_1 \forall A_2 [ [[[ \text{number}(A_1) ] \land \text{number-array}(A_2) ] ] ] \\
\forall A_3 [ A_2 = 1; A_3 ] ] \land [ \exists A_3 [ A_2 = A_3; A_1 ] ] ] \\
\forall A_5 [ \forall A_4 [[ \text{number}(A_3) ] \land \text{number}(A_4) ] ] \land \forall A_6 [ [[[ A_2 = A_3; A_4 ] ] ] ] \lor [ \forall A_2 [ A_2 = A_5; A_3; A_4 ] ] ] ] \\
The proof is based on induction for the **scheme** \((A_1 \times (A_1 + 1))\) where the start is \(A_1 = 1\) and the induction is based on \(((A_1 + 1) \times ((A_1 + 1) + 1)) = ((A_1 \times (A_1 + 1)) + (2 \times A_1))\).

b) **THEOREM** with Gödel's beta-function

One can give a representation of the **Successive-number-array** starting from 1 up to arity \(A_4\) using Gödel's beta-function-technique (the existence of \(A_1\) and \(A_2\) are guaranteed by Gödel's beta-function metatheorem) (it may be made unique by choosing the smallest \(A_1\)). The first auxiliary **THEOREM** states that one can represent the ascending array **Successive-number-array** by Gödel's beta-function codes:

\[
\forall A_4[\exists A_1[\exists A_2[\forall A_3[[A_3 < A_4] \rightarrow \gamma((AXFOgbeta; A_0 \beta A_3')])]]]
\]

and a representation of the successive-sum array thereof

\[
\forall A_4[\exists A_1[\exists A_2[\forall A_3[[A_3 < A_4] \rightarrow \gamma(A_0((AXFOgbeta; A_3 \beta A_3') ; A_0 \beta (A_0 + A_3')))]]]]
\]

And one can thus state **THEOREM** of little Gauss:

\[
\forall A_4[\exists A_1[\exists A_2[\forall A_3[[A_3 < A_4] \rightarrow \gamma(A_0((AXFOgbeta; A_3 \beta A_3') ; A_0 \beta (A_0 + A_3')))]]]]
\]

And one can prove it based on Gödel's beta-function metatheorem and the induction for the **scheme** \((A_3 \times (A_3 + 1))\).

c) **THEOREM** in a concrete calcule with recursive arithmetic

It is a different story in the concrete calcule **LAMBDA** of decimal primitive arithmetic where one has the tools of primitive recursion. The **number-array** 1;2;3;4; … :\(A_1\) is coded by arity \(A_1\) and **pinon** \(A_2=1\). Given two strings \(^{1)}\) and **Axllisu** and **Azrlisp** one can construct a **pinon** for every \(A_2\) by concatenating them to **Axllisu** \(A_2\) **Azrlisp**. For a given arity this **pinon** sums up the constituents and there is a **pinon** \(\text{Acarl}\) for carlation, conventionally written as \((x(x+1))/2\). The **THEOREM** of little Gauss reads:

\[
\forall A_1[\text{Axllisu } A_2 \text{Azrlisp}(A_1)] = \text{Acarl}(A_1)]
\]

This means: once one has realized that **number-array** strings can be represented by their arity and a code, one can express the **THEOREM** of little Gauss perfectly in **LAMBDA** and it can be proven within **LAMBDA** too.

---

1) the **extra-individual-constant** strings are again used like a makro in programming languages just names for strings that are to be expanded wherever they appear in synaptions.
6 Euclid’s theorem of unlimited primes

Contrary to the preceding section it is not problem to express the **THEOREM** of unlimited primes properly in concrete calcule ALPHA of decimal Robinson arithmetic. One starts off with unary formula $\text{AFAPrime}$

$$\text{AFAPrime} = [1 < A_1] \land [\forall A_3[\forall A_2[[A_1 = (A_3 \times A_3) \rightarrow [A_3 = 1] \lor [A_3 = A_1]]]]]$$

that defines prime number strings and then one can express the **THEOREM**:

$$\forall A_1[[\text{AFAPrime}] \rightarrow \exists A_2[[\langle\text{AFAPrime}; A_1/A_2 \rangle \land [A_1 < A_2]]]]$$

However the proof needs **arrays of open arity**. This means that for a proof one has to use the second step and move from object-language to metalanguage (and back). The translation of the **THEOREM** into a metatheorem and the arrangements for the proof are a bit tedious but trivial. **Successive-prime-array** strings come handy, example 2;3;5;7;11;13

$$\forall A_1[[\text{Successive-prime-array}(A_1)] \leftrightarrow [[\text{number-array}(A_1)] \land [\exists A_7[ A_1 = 2; A_7]]] \land$$

$$[\forall A_2[\forall A_3[[[\text{number-array}(A_2)] \land [\exists A_3]]] \land [\forall A_4[\forall A_5[[[A_1 = A_2; A_3]] \lor$$

$$[A_1 = A_2; A_3; A_5]] \lor [A_1 = A_4; A_2; A_5]]] \lor [A_1 = A_4; A_2; A_3; A_5]]])]] \rightarrow [[[[ \text{TRUTH}(\text{AFAPrime}; A_1/A_2)] \land [\text{TRUTH}(\text{AFAPrime}; A_1/A_3)] \land [A_2 < A_3]]] \land [\forall A_6[\text{TRUTH}(\text{AFAPrime}; A_1/A_6)]]]$$

a) metatheorem

$$\forall A_1[[[\text{number-array}(A_1)] \land [\text{TRUTH}((\text{AFAPrime}; A_1/A_1))]] \rightarrow$$

$$[\exists A_2[[[\text{number-array}(A_2)] \land [\text{TRUTH}((\text{AFAPrime}; A_1/A_2))]] \land [A_1 < A_2]]]$$

For the proof construct $A_4$ from **Successive-prime-array** as successor of the product of its constituents. **Metalingual proofs can be lengthy (and a bit boring in its details), so just a sketch is given as usual:**

$$[\text{Successive-prime-array}(A_3)] \land [[A_3 = 2; 3] \lor$$

$$[\exists A_5[[\text{Successive-prime-array}(A_5)] \land [A_3 = A_2 A_2]]]$$

$$A_4 = ((((((((A_3 A_9) A_8) A_7) A_6) A_5) A_4) A_3) A_2) A_1) A_0 ; /((1 \times (A_2 ; ; /)) A_0)’$$

b) **THEOREM** with Gödel’s beta-function

The idea is to use **number-array** strings as e.g. in conventional notation: 1, 2, 6, 30, 210, 2310 that are are generated by successive products of prime **number** strings. For a given prime **number** $A_1$ one can find the corresponding **number-array** that ends with the constituent that is the product of all preceding primes, its successor is a prime number greater than the considered one. This can be done using Gödel’s beta-function technique with codes $A_4$ , $A_5$ and arity $A_6’$ with the quaternary formula :

$$\text{AFASupr}= [[[((\text{AFAXgamma}; A_1/A_4); A_2/A_5); A_3/A_0); A_0/1]] \land$$

$$[\exists A_7[[[[([\text{AFAXgamma}; A_1/A_4); A_2/A_5]; A_3/A_6); A_0/A_7]]] \land$$

$$[\exists A_8[[[[([\text{AFAXgamma}; A_1/A_4); A_2/A_5]; A_3/A_6); A_0/A_8]]] \land [A_8 = (A_7 \times A_1)]]]]] \land$$

$$[\forall A_9[[A_9 < A_6] \rightarrow [\forall A_{10}[\forall A_{11}[[[[[([\text{AFAXgamma}; A_1/A_4); A_2/A_5]; A_3/A_9]; A_0/A_{10}] \land$$

$$[\forall A_{11}[[[[[([\text{AFAXgamma}; A_1/A_4); A_2/A_5]; A_3/A_9]; A_0/A_{11}] \rightarrow [\forall A_{12}[[[[[([\text{AFAXgamma}; A_1/A_4); A_2/A_5]; A_3/A_9]); A_0/A_{11} \times A_{12}] \land$$

$$[\forall A_{13}[[[A_1 < A_{12}] \land [(\text{AFAPrime}; A_1/A_3)] \rightarrow [[A_1 < A_{13}]]]]]]]]]]]]]] \land$$

For the proof take the construction of a prime number $A_0’$ greater than $A_1$ :

$$\forall A_1[[\text{AFAPrime}] \rightarrow \exists A_3[\exists A_5[A_6][[\text{AFAPrime}; A_1/A_4); A_2/A_5]; A_3/A_6]] \land$$

$$[\forall A_0[[[[[([\text{AFAXgamma}; A_1/A_4); A_2/A_5]; A_3/A_6)] \rightarrow [[[\text{AFAPrime}; A_1/A_0’)] \land [A_1 < A_0’]]]]]]]]]$$

c) It is a different story in the concrete calcule LAMBDAb of decimal pinitive arithemec where one has the tools of primitive recursion. There one can express the **Successive-prime-array** by means of code and perform the proof within the calcule.
7 Fundamental theorem of natural arithmetic

The Fundamental theorem of natural arithmetic (canonical representation of a natural numbers by unique prime-power decomposition) is illustrated by the example \(504 = (((((2\times2)\times2)\times3)\times3)\times7)\). It cannot be expressed immediately in concrete calcule ALPHA of decimal Robinson arithmetic as a THEOREM. First one has to take refuge to the corresponding metatheorem:

a) fundamental metatheorem of natural arithmetic

**Ascending-prime-array** strings come handy, example 2;2;2;3;3;7, for expressing the metatheorem:

\[
\forall A_1[[\text{Ascending-prime-array}(A_1)] \iff [[\text{number-array}(A_1)] \land [[(\text{AFAprime}; A_1/A_1)] \lor \\
[(\forall A_2[ \forall A_3[[\text{number-array}(A_2)] \land [\text{TRUTH}(A_1 = ((\text{FApripopair}, A_1/A_2)] \land [(\forall A_3[\text{Ascending-prime-array}(A_3)] \land [\text{TRUTH}(A_1 = ((((((A_3\theta_9)\theta_8)\theta_7)\theta_6)\theta_5)\theta_4)\theta_3)2\theta_1)\theta_0) ; \sigma (1 \times (A_2 ; \sigma) \times \) \theta_0))]) \land \\
[A_2 = A_3])]]]]]
\]

The first part states the existence and the second part takes care of uniqueness: The proof necessitates induction, preferably in the form of infinite descent.

b) fundamental THEOREM in Robinson natural arithmetic with Gödel's beta-function

The idea is to use number-array strings of products of successive powers of ascending primes, for the above example: 1;8;72;504, the last one being the number in question. Firstly the binary formula prime-power-pair AFApripopair is defined which is true if the first argument \(A_1\) is a prime number and the second argument \(A_2\) is a power thereof, e.g. 5 and 125 are such a pair.

\[
\text{AFApripopair} = [[1 < A_1] \land \forall A_3[[\forall A_2[[A_1 = (A_3 \times A_3) \rightarrow [A_3 \equiv 1] \lor [A_3 = A_1]]]]] \land \\
[\forall A_3[\forall A_3[[A_2 = (A_3 \times A_3) \rightarrow [1 < A_3] \rightarrow [\exists A_3[A_3 = (A_3 \times A_1)]]]]]]
\]

The fundamental THEOREM of natural arithmetic in concrete calcule ALPHA looks a little bit complicated (and extends to about 30 lines if one expands formula strings AXFObeta and AFApripopair), where the first part states the existence and the second part takes care of uniqueness:

\[
\forall A_1[[1 < A_1] \rightarrow [\exists A_4[\exists A_5[\exists A_6[[[[[\text{AXFObeta}; A_1/A_4]; A_2/A_5]; A_3/0]; A_0/1]]] \land \\
[[\text{AXFObeta}; A_1/A_4]; A_2/A_5]; A_3/A_6]; A_0/A_1]]] \land \\
[\forall A_7[[A_7 < A_6] \rightarrow [\forall A_8[\forall A_9[[[[[\text{AXFObeta}; A_1/A_4]; A_2/A_5]; A_3/A_7]; A_0/A_8]]] \land \\
[[\text{AXFObeta}; A_1/A_4]; A_2/A_5]; A_3/A_7]; A_0/A_9]]] \rightarrow \\
[\forall A_1\forall A_1[[[[\text{AFApripopair}; A_1/A_10]; A_2/A_8]]] \land [\forall A_1\forall A_1[[[[\text{AFApripopair}; A_1/A_11]; A_2/A_9]]] \land \\
[[0 < A_7] \rightarrow [A_0 < A_1]]]]]]]]]]]]]
\]

[\forall A_24[\forall A_25[\forall A_26[[[[[\text{AXFObeta}; A_1/A_24]; A_2/A_25]; A_3/0]; A_0/1]]] \land \\
[[\text{AXFObeta}; A_1/A_24]; A_2/A_25]; A_3/A_26]; A_0/A_1]]] \land \\
[\forall A_7[[A_7 < A_6] \rightarrow [\forall A_8[\forall A_9[[[[[\text{AXFObeta}; A_1/A_24]; A_2/A_25]; A_3/A_7]; A_0/A_8]]] \land \\
[[\text{AXFObeta}; A_1/A_24]; A_2/A_25]; A_3/A_7]; A_0/A_9]]] \rightarrow \\
[\forall A_10[\forall A_11[[[[\text{AFApripopair}; A_1/A_10]; A_2/A_8]]] \land [\forall A_10[\forall A_11[[[[\text{AFApripopair}; A_1/A_11]; A_2/A_9]]] \land \\
[[0 < A_7] \rightarrow [A_0 < A_1]]]]]]]]]]]]] \rightarrow [A_6 = A_26] \land [\forall A_{15}[A_{15} < A_6] \rightarrow [\forall A_0[ \\
[[\text{AXFObeta}; A_1/A_4]; A_2/A_5]; A_3/A_15]]] \land [\forall A_{13}[[[\text{AXFObeta}; A_1/A_24]; A_2/A_25]; A_3/A_26]]] \rightarrow [A_9 = A_26] ]
\]

version 1.0 | Open Arity | 10
c) fundamental **THEOREM** of natural arithmetic in a concrete calcule with recursive arithmetic

It is a different story in the concrete calcule LAMBDA of decimal pinitive arithmetic where one has the tools of primitive recursion. There one can express the **Ascending-prime-array** by means of its arity and a **pinon** code and perform the proof within the calcule where one has the possibility of limited sums and products as was mentioned at the end of section 5.

8 **Open arity in other areas of mathematics and conclusion**

Open arity and related features are needed in many other areas of mathematics, e.g.

- axiom schemata of **separation** and **replacement** of axiomatic set theory
- **induction** and **recursion** for functions of **any arity** in number theories.
- an infinite count of functions for proper definition of recursive functions
- geometrical space of **unspecified dimension** (how to express n-tuples)
- definition and use **polynomials**, say for integer, rational or algebraic arithmetics
- systems of **unspecified finite cardinality** (e.g. finite groups and Galois fields).
- finite and infinite **graph theories** and many more.

All of them can be treated properly by the FUME-method with the two-tiers of languages Funcish and Mencish. Of course common English can be used as an unprecise supralanguage to talk about everything. However, it is important to know about the shortcomings of unprecise language. Supralanguage English (or any other natural language) is but a means to express comments and to reason in a plausible fashion. The precise talking has to be done in Mencish and Funcish:

<table>
<thead>
<tr>
<th>supra</th>
<th>English</th>
<th>talks about</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta</td>
<td>metacalcule <strong>sigma</strong></td>
<td>Mencish</td>
</tr>
<tr>
<td>object</td>
<td>abstract calcule <strong>sigma</strong></td>
<td>Funcish</td>
</tr>
<tr>
<td>infra</td>
<td><strong>nothing</strong></td>
<td>codex <strong>ALPHA</strong></td>
</tr>
</tbody>
</table>

*Figure 1  Hierarchy of languages and codices pertinent to the FUME-method for two example calcules, an abstract and a concrete one*

A logic with only one tier is not sufficient for the foundation of mathematics. Extending predicate logics to theory of types, introducing axiomatic set theory and other constructions does not solve the problem. One needs at least two tiers, a precise object-language together with a precise metalanguage.
Appendix A  Selected basic metaindividuals, metarelations and metafunctions

syntactic metaproperties in general (sort $\phi$)
- **petit-number** string with only $0,1,2,3,4,5,6,7,8,9$ (for convention decimals are used)
- **number** string with only $0,1,2,3,4,5,6,7,8,9$ (for convention decimals are used)
- **number-array** array of **number** strings separated by semicolon
- **variable** **formulo** string followed by **petit-number**
- **variable-array** array of **variable** strings separated by semicolon
- **omnia** multiple distinct **omnia** strings e.g. $\phi_2[\phi_1][\phi_3]\phi$
- **pattern** built up from **function-constant** strings with **number** and **variable** strings
- **term** **pattern** with **number** strings only
- **scheme** **pattern** with at least one **variable** strings only
- **phrase** built up from equalities of **pattern** strings and from **relation-constant** strings using full predicative logic
- **sentence** **phrase** with no free **variable** strings
- **formula** **phrase** with at least one **variable** (arity is count of distinct **variable** strings), no $\phi$
- **formulo** like **formula** but with $\phi$ (which is left out for arity count)

**Successive-prime-array** array of **number** strings, that are successive primes

**Ascending-prime-array** array of **number** strings, that are ascending (not necessarily successive) primes

alethic metaproperties in general
- **UNEX-formulo** representing a function by a **formulo** with unique existence of output for input
- **TRUTH** any alethic **sentence**
- **THEOREM** quantive alethic **sentence** that is not basic
- **Axiom . Basiom** **sentence** introduced as basic **TRUTH** (in abstract or concrete calcule resp.)

metaindividuals in calcule **ALPHA** \{ sort $A$ \}
- **AXFOgbeta** ternary **UNEX-formulo** representing Gödel's beta-function,
- **AXFOadp** binary **UNEX-formulo** representing antidiagonal pair coding
- **AXFOada** unary **UNEX-formulo** representing auxiliary function for antidiagonal pair coding
- **AXFOadr** unary **UNEX-formulo** representing row decoding function of antidiagonal pair
- **AXFOadc** unary **UNEX-formulo** representing column decoding function of antidiagonal pair
- **AFPrime** unary **formula** characterizing **number** strings
- **AFAeurpr** unary **formula** of products of successive primes, ending at the given argument $A_1$
- **AFApripopair** binary **formula**, so that $A_1$ is a prime and $A_2$ is a power thereof

syntactic binary metarelations in general and in calcule **ALPHA**
- $\phi \equiv \phi$ matching length of strings
- $\phi \neq \phi$ smaller length of strings
- $\phi \supset \phi$ souting (suitably containing, i.e. in a way that avoids disambiguities)
- $\phi / \phi$ suitably-free-in
- $\phi / \phi$ suitably-bound-in
- $\phi \sim \phi$ compatible (no collision of bound **variable** strings in constructing **phrase** strings)
- $A \ll A$ natural-minority, smaller with respect to numbering by **number**

syntactic metafunctions in general and in calcule **ALPHA**
- $(\phi;\phi)$ synaption (concatenation except for leading $\phi$)
- $(\phi;\phi)$ character-deletion
- $(\phi;\phi)$ string-replacement
- $A''(A)$ succession with respect to **number** (10 characters)
- $A^n(A)$ length as **number**, e.g. $A^n(\forall A_1 \ldots ) = 4$
- $A^\infty(A)$ arity as **number**, e.g. $A^\infty(A_1;A_2;A_3;A_4;A_5) = 5$
- $A^\forall(A;A)$ projection: substring of **array** in second place at position with **number** in first place

version 1.0  Open Arity  12
Appendix B Gödel's beta-function and more in abstract Robinson-Crusoe arithmetic

Based on the observation that one only needs the **UNEX-formula** technique for representation of functions in concrete calculate **ALPHA** of Robinson decimal natural number arithmetic one remembers equation $(x+y)^2=x^2+y^2+2xy$ (in classical notation) to produce an even weaker calculate. This time the **abstract** counter piece is introduced. The interesting feature is that one can leave away the binary function **multiplication**; unary **quadrature** is sufficient.

The ontological basis of abstract calculate **alphakappa** of Robinson-Crusoe natural number arithmetic comprises the following ingredients:

| sort :: | ακ |
| basis-individual-constant :: | ακn |
| basis-function-constant :: | ακ' ; (ακ+ακ) ; (ακ↑) |
| basis-relation-constant :: | ακ<ακ |
| extra-individual-constant :: | ακu=ακn' |

**Axiom** strings

A1  ∀ακ1[ακ1≠ακn]
A2  ∀ακ1[ακ2[ακ1=ακ2']→[ακ1'=ακ2]]
A3  ∀ακ1[[ακ1+ακn]→[∃ακ2[ακ1=ακ2']]]
A4  ∀ακ1[(ακ1+ακn)=ακ1]
A5  ∀ακ1[ακ2[(ακ1+ακ2')=(ακ1+ακ2)]]
A6  ∀ακ1[(ακ1↑)=ακn]
A7  ∀ακ1[(ακ1')=((ακ1↑)+ακ1)+ακ1']
A8  ∀ακ1[−[ακ1<ακn]]
A9  ∀ακ1[[ακn=ακ1]∨[ακn<ακ1]]
A10 ∀ακ1[ακ2[ακ1<ακ2]→[[ακ1'=ακ2]∨[[ακ1'<ακ2]]]]
A11 ∀ακ1[ακ2[[ακ1<ακ2]→[[ακ1<ακ2]∨[[ακ1=ακ2]]]]]

**Axiom** matres for the unary and multary case of induction:

∃ακ1[ [sentence(∀ακ1[ακ1])] →
[ Axiom(∃[ακ1↑/ακn])∧[∀ακ1[[ακ1]→((ακ1↑)(ακ1'/ακ1')]]]→[∀ακ1[ακ1]] ) ] ]

∃ακ1[∃ακ2[∃ακ3[ [ [formula(ακ1) ]∧[ omny(ακ2) ] ]∧ [sentence( ακ2∧ακ1[ακ1][ακ3] ) ] ]→
[ Axiom( ακ2[ [ακ1; ακ1'/ακ1] ]∧[∀ακ1[[ακ1]→[ (ακ1; ακ1'/ακ1') ]]]]→[∀ακ1[ακ1][ακ3]] ) ] ]

One uses the following binary **UNEX-formula** for the introduction of multiplication:

$$\alpha κXFOMul = ((\alpha κ1+\alpha κ2)↑)=((\alpha κ1↑)+(\alpha κ2↑))+(\alpha κ0+\alpha κ0))$$

One has a unary **formula** in Robinson-Crusoe arithmetic to express that a **number** string is prime:

$$\alpha κXFAPrime = ∀\alpha κ30[∀\alpha κ3[[[\alpha κu<\alpha κ30][\alpha κ30<\alpha κ31']][\alpha κ31<\alpha κ1]]→
[((\alpha κ30+\alpha κ31)↑)≠((\alpha κ30↑)+(\alpha κ31↑))+(\alpha κ1+\alpha κ1)]]]$$

As well one can represent Gödel's beta-function in Robinson-Crusoe arithmetic by a ternary **UNEX-formula** using auxiliary bound **variable** strings ακ21 and ακ22 that are limited by $((\alpha κ1+\alpha κ2)↑)$:

∃ακ21[((ακ1+ακ2)↑)=((ακ1↑)+(ακ2↑))+(ακ21+ακ21))]

∃ακ22[((ακ21↑+ακ20)↑)=((ακ21↑)+(ακ20↑))+(ακ22+ακ22)]

$$\alpha κXFObeta = ∃ακ20[∃ακ21[∃ακ22[ [[((\alpha κ2+\alpha κ2')↑)=((\alpha κ2↑)+(\alpha κ3↑))+(\alpha κ21+ακ21)]]∧
[((\alpha κ21'+\alpha κ20)↑)=((\alpha κ21↑)+(\alpha κ20↑))+(\alpha κ22+\alpha κ22)]]][[\alpha κ0<((\alpha κ2×\alpha κ3')')]]]]$$