Representation of processive functions in Robinson arithmetic ?

Hannes Hutzelmeyer

Summary

In connection with his so-called incompleteness theorem Gödel discovered the beta-function. The beta-function theorem is important for the representation of recursive functions in the concrete calcule ALPHA of Robinson arithmetic. The other features are composition and minimization of primitive recursive functions. Recursive functions are no part of Robinson arithmetic, but they are representable by certain formulae.

The author has developed an approach to logics that comprises, but goes beyond predicate logic. The FUME method contains two tiers of precise languages: object-language Funcish and metalanguage Mencish. It allows for a very wide application in mathematics from recursion theory and axiomatic set theory with first-order logic, to higher-order logic theory of real numbers and so on.

The concrete calcule LAMBDA of a natural number arithmetic with first-order logic has been defined by the author. It includes straight recursion and composition of functions, it contains a wide range of so-called compinitive functions, with processive functions far beyond primitive recursive functions. They include e.g. Ackermann's function and similar constructions. All recursive functions (that are obtained by minimization too) can be represented in LAMBDA . As long as there is no proof that all processive functions are minimitive recursive (recursive but not primitive recursive) one has the problem of representing them in concrete calcule ALPHA of Robinson arithmetic. As long as the challenge of such a proof is not met there is the **conjecture** that there are calculative functions that are **not representable** in Robinson arithmetic.

An abstract calcule alphakappa of Robinson-Crusoe arithmetic shows that there exists an even weaker adequate arithmetic than Robinson's.

Contact: Hutzelmeyer@pai.de
https://pai.de

1. FUME system of object-language and metalanguage

The author has put forward **FUME** a **precise** system of **object-language Funcish** and **metalanguage Mencish** that overcomes certain difficulties of predicate logic and that extends to a full theory of **types**. In order to describe an **object-language** one needs a **metalanguage**. According to the author's principle metalanguage has to be absolutely precise as well, normal English will not do. There are at least three levels of language:

English	supralanguage	natural	talks about everything
Mencish	metalanguage	formalized precise	talks about object-language
Funcish	object-language	formalized precise	language of mathematics

The essential parts of a language are its sentences. A sentence is a **string** of **characters** of a given **alphabet** that fulfills certain rules. This means that metalanguage talks about the strings of the object-language. The essential parts of the metalanguage are the metasentences (that are strings of characters as well). It is important to realize that the metalanguage talks about the strings of the object-language and nothing but. If one wants to comment on a certain mathematical system that is realized with the use of an object-language one has to take refuge to the supralanguage. As supralanguage is not a formal, precise language, there are no restrictions. One can comment on mathematical systems and one can talk in supralanguage specifically about metasentences, just as metalanguage talks about object-language.

On first sight Funcish and Mencish look familiar to what one knows from predicate-logic. However, they are especially adapted to a degree of precision so that they can be used universally for all kind of mathematics. And they lend themselves immediately to a treatment by computers, as they have perfect syntax and semantics. It is not the place to go into details. Both Funcish and Mencish have essentially the same syntax. Mencish, however, has strictly first-order logic. The **fonts-method** allows to distinguish between object-language (Arial and Symbol, normal, e.g. ∀Λ1[ ), metalanguage (Arial and Symbol, boldface italics e.g. ***Axiom***) and supralanguage English (Times New Roman).

Notice that Funcish and Mencish have a context-independent notation, which implies that one can determine the **category** of every language element uniquely from its syntax, 'wherefore by their *words* ye shall know them' (*fruits* according to Mathew 7.20). The reader may be puzzled by some expressions that are either newly coined by the author or used slightly different from convention. This is done in good faith; the reason for the so-called **Bavaria notation** is to avoid ambiguities.

There are some hints on the front of the author's homepage https://pai.de/ . You will find some a short description in chapter 1. of the pdf-download GeoO1.1.pdf that can be started from 'Geometries of O' on the homepage. There is also a description in the pdf-download that can be started from 'Church's thesis …' on the homepage. This publication from 2006, however, is not quite up to date in other respects. A complete description of Funcish and Mencish is forthcoming.

'Calcule' is the name given to a mathematical system with the precise language-metalanguage method FUME . 'Calcule' is an expression coined by the author in order to avoid confusion. The word 'calculus' is conventionally used for real number mathematics and various logical systems. As a German translation 'Kalkul' is proposed for 'calcule' versus conventional 'Kalkül' that usually corresponds to 'calculus'. Calcules are given names using some convention that relates to the Greek **sort** names of a calcule, e.g. concrete calcule LAMBDA with sort Λ .

A **concrete calcul**e talks about a **codex** of concrete **individuals** (given as strings of characters) and concrete **functions** and **relations** that can be realized by 'machines' (called calculators). An **abstract calcule** talks about **nothing**. It only says: if some entities exist with such and such properties they also have certain other properties. Essentially there are only 'if-then' statements. E.g. 'if there are entities that obey the Euclid axioms the following sentence is true for these entities'.

Mencish ih the language of the corresponding metacalcules, metasentences talk about ***sentence*** and other strings of Funcish calcules. It contains many metaproperties that classify strings of Funcish, but there are some metafunctions too. In section 5 it will be made use of metafunction string-replacement ***(Λ ; Λ∫Λ)*** where ***(Λ1 ; Λ2∫Λ3)*** gives the result of replacing all suitable appearances of the second string ***Λ2*** in the first string ***Λ1*** by the third string ***Λ3*** . Mencish contains a few other metafunctions as well as some unary and binary metrelations.

Mencish allows for a precise defintion of what is usually called an ***Axiom*** scheme or **schema**. It is preferred to talk about a ***sentence*** **mater**. In sections 2 and 5 the metalingual expression ***scheme*** will be introduced and treated with a completely **different** meaning. *As mentioned before, so-called Bavaria notation has been chosen for good reasons. Although it may put up some hardship for the reader in the beginning, it will finally be realized that it gives so much more clarity.*

Funcish allows for higher-order logic by means of ***type*** strings, e.g. ***function-type*** Λ(Λ) or ***property-type*** (Λ) that one could e.g. put into∀Λ1(Λ)[... or ∃1(Λ)[... where the ***function-variable*** Λ1(Λ) and the ***relation-variable*** 1(Λ) appear.

It is not absolutely correct to say that first-order logic is sufficient for calcule LAMBDA . Like for many other calcules one needs the **implicit definition of functions** . To this end one has to make a little detour to second-order logic, but one can return from that detour anytime. The detour means that one makes use of the purely logical ***Implicition-axiom*** matres allowing for the **implicit definition of functions**. They state the unique existence of functions so that they can be given names (i.e.***extra-function-constant*** strings); subsequently these functions can be used in normal fashion. Afterwards there occur no omnications with ∀Λ1(Λ)[or entications with ∃Λ3(Λ)[and therefore one again is in the safe world of first-order logic The method is based on ***UNEX-formulo***[1] strings, that have to be introduced now.

As opposed to a ***formula*** that must not include the ***variable*** Λ0 a ***formulo*** must include the ***variable*** Λ0 . ***UNEX-norm-formulo***[2] strings define relations that hold for exactly one value Λ0 for every booking of the input ***variable*** strings Λ1 , Λ2 , … according to the arity of the ***UNEX-formulo*** . It is metadefined as follows in the unary case. This is the first appearance of a metasentence; remember that the boldface italics fonts belong to Mencish that talks about strings of Funcish that uses normal fonts. You also see that the same logic syntax is used in both Funcish and Mencish. Requiring the string ∀Λ0[∀Λ1[***Λ1***]] to be a ***sentence*** means that ***Λ1*** is a ***formulo*** with exactly the free ***variable*** strings Λ0 and Λ1. The second condition means that ***variable*** Λ2 does not appear bound in ***Λ1*** . The following metasentence defines a ***UNEX-formulo*** such that theres exists a value Λ0 for all input, and that this value is unique:

∀Λ1[[[sentence(∀Λ0[∀Λ1[***Λ1***]] ***)] ∧ [sentence(*** ∀Λ0[∀Λ1[∀Λ2[***Λ1***]]] ***)]] →***
[[UNEX-norm-unary-formulo(Λ1)] ↔
[TRUTH(∀Λ1[∃Λ0[[***Λ1***]∧[∀Λ2[[***(*** ***Λ1;*** Λ0∫Λ2***)***]→[Λ2=Λ0]]]]] ***)]]]*** [3]

Talking about the arity of ***UNEX-formulo*** strings the ***variable*** Λ0 is not counted. A nullary ***UNEX-formulo*** string has no other ***variable*** , a unary ***UNEX-formulo*** string has one free, a binary ***UNEX-formulo*** string has two other free ***variable*** strings and so on.

Logical ***Axiom***[4] of **implicit definition of unary functions** by ***UNEX-formulo***

∀Λ1[[[sentence(∀Λ0[∀Λ1[***Λ1***]] ***)] ∧ [sentence(*** ∀Λ0[∀Λ1[∀Λ2[***Λ1***]]] ***)]] →***
[Axiom([∀Λ1[∃Λ0[[***Λ1***]∧[∀Λ2[[***(*** ***Λ1;*** Λ0∫Λ2***)***]→[Λ2=Λ0]]]]]→
[∃Λ1(Λ)[[∀Λ1[***(*** ***Λ1;*** Λ0∫Λ1(Λ1)***)***]]∧[∀Λ2(Λ)[[∀Λ1[***(*** ***Λ1;*** Λ0∫Λ2(Λ1)***)***]]→[Λ2(Λ)=Λ1(Λ)]]]]] ***)]]***

[1] the capital letters indicate that ***UNEX-formulo*** is not a metaproperty that is effectively decidable like e.g. ***formulo***

[2] ***norm*** means ***variable*** strings Λ0 and consecutive Λ1 , Λ2 , Λ3 …

[3] the capital letters indicate that ***TRUTH*** is not a metaproperty that is effectively decidable like e.g. ***sentence***

[4] the only initial capital letter indicates that metaproperty ***Axiom*** is related to ***TRUTH*** but decidable

2. Concrete calcule LAMBDA for pinitive functions

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

Arial 8, petit-number for variables										*Arial 12, normal size numbers for decimal individuals*									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Symbol 12, general logic symbols,															*special calcule symbols*				
=	≠	¬	∨	∧	→	↔	∃	∀	[	]	(	)	;	&	*	#	≤	□	Λ

List of 40 characters of calcule LAMBDA

sort ::	Λ	
sort-array ::	***sort*** ¦ ***sort-array*** ; ***sort***	*the recursivw definition is evident*
decimal :: number ::	0 ¦ 1 ¦ 2 ¦ …	*definition without dot-dot-dot see section 5*
basis-ingredient ::	***sort*** ¦ ***decimal*** ¦ ***basis-function-constant*** ¦ ***basis-relation-constant***	
basis-function-constant ::	Λ( ) ¦ Λ(***sort-array***) ¦ (Λ∗Λ)	*pinitive functions, decimal synaption*
basis-relation-constant ::	#Λ ¦ Λ≤Λ	*pinity, minority*
pinon-catena ::	***pinon*** ¦ ***pinon-catena pinon***	
pinon-array ::	***pinon*** ¦ ***pinon-array*** ; ***pinon***	
pinon ::	0 ¦ 1 ¦ 2 ***pinon pinon*** ¦ 8 ***pinon pinon-catena*** 9	*only 4 cases*

pinon strings are natural numbers that **code** primitive functions, when they replace Λ in ***basis-function-constant*** string Λ( ) or Λ(***sort-array***) resp. : 0 codes the zero function, 1 the succession function. The third case 2 ***pinon 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.

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 its definition among the ***basis-ingredient*** strings is redundant, as it could be given by a ***pinon*** . The same is true for ***basis-relation-constant*** strings #Λ and Λ≤Λ as they can be defined using some ***pinon*** strings Λpiny[1)] and Λemiy resp. .

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. Very few examples for coding of primitive recursive functions by decimal numbers are given here (many are given in the download listed below). It is a funny observation that pinitive functions have a Janus face. They have been designed to produce primitive recursive functions

22011(Λ_1;Λ_2) 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:

Λ_1(0) the value for all codes at 0 where the result is put to 0 if Λ_1 is not a ***pinon*** code.

The strange functions that can be obtained by putting variables into code position can be generalized to so-called **processive** functions. One realizes that ***scheme*** strings that are obtained from ***function-constant*** strings by inserting ***number*** and ***variable*** strings and compositions thereof produce functions (conventionally they are called *general terms)*. The world of processive functions is very rich, e.g. it comprises straightforwardly **Ackermann's function** and other **hyperexponentiation**s that are obtained by the so-called generator technique (see download C6-C7-Pinon.pdf for 'Programming primitive recursive functions and beyond' that can be obtained on homepage https://pai.de/Church-s-thesis/Programming-functions).

[1)] one can introduce ***extra-number-constant*** as names by adding a ***small-medium-letter-word*** subscript to the ***sort*** Λ ; a ***pinon*** string can be referred to both in Mencish and Funcish, e.g.by ***Λupr*** or Λupr resp.

3. Primitive and minimitive recursive functions

Concrete calcule LAMBDA of decimal pinitive arithmetic allows to define what is meant by a recursive unary function by its **representation** as a ***UNEX-recursive-norm-unary-formulo(Λ1)*** . A ***UNEX-norm-unary-formulo Λ1*** contains exactly ***variable*** strings Λ0 and Λ1 and fulfills the condition ***UNEX*** wich means that for every Λ0 there exist exactly one Λ1 ; uniqueness is obtained by choosing the smallest possible value (minimization). It is called ***recursive*** if its either ***primitive*** or ***minimitive*** :

$\forall \Lambda_1[\,[\,\textit{UNEX-primitive-norm-unary-formulo}(\Lambda_1)\,] \leftrightarrow [\,\exists \Lambda_2[\,[\,\textit{pinon}(\Lambda_2)\,] \wedge [\Lambda_1 = \Lambda_0 = \Lambda_2(\Lambda_1)]\,]\,]\,]$

$\forall \Lambda_1[\,[\,\textit{UNEX-minimitive-norm-unary-formulo}(\Lambda_1)\,] \leftrightarrow$
$[\,\exists \Lambda_2[\,\exists \Lambda_3[\,[\,[\,\textit{pinon}(\Lambda_2)\,] \wedge [\,\textit{pinon}(\Lambda_3)\,]\,] \wedge [\,\textit{TRUTH}(\ \forall \Lambda_1[\exists \Lambda_2[\Lambda_2(\Lambda_1;\Lambda_2)=0]]\ )\,]\,] \wedge$
$[\,\Lambda_1 = \exists \Lambda_2[[[\Lambda_2(\Lambda_1;\Lambda_2)=0] \wedge [\forall \Lambda_3[[\Lambda_2(\Lambda_1;\Lambda_3)=0\ ] \rightarrow [\Lambda_2 \leq \Lambda_3]]]] \wedge [\Lambda_0 = \Lambda_3(\Lambda_2)]]\,]\,]\,]$

It was proven by Kleene that **one minimization** suffices. The definition of ***UNEX-minimitive-norm-unary-formulo*** strings shows that they are denumerable (as finite strings of characters) but not enumerable (meaning effectively denumerable), as it cannot be decided in general if the primitive recursive function ***scheme*** $\Lambda_2(\Lambda_1;\Lambda_2)$ has a zero Λ_2 for all arguments Λ_1 . Therefore recursive functions are not enumerable - and thus do not lend themselves to diagonalization. However, one can say e.g. 'for all unary minimitve functions' as they are given by Λ_3 and Λ_4 with **unary-regularity-condition** $\forall \Lambda_1[\exists \Lambda_2[\Lambda_3(\Lambda_1;\Lambda_2)=0]$

It is sufficient to consider ***UNEX-minimitive-norm-unary-formulo*** strings as ***UNEX-primitive-norm-unary-formulo*** with a ***pinon Λ3*** can be expressed as ***UNEX-minimitive-norm-unary-formulo*** strings with the trivial choice: $\Lambda_2 =$ 82201202201201220120 12019 (that is ***pinon*** Λjsub for the primitive function subtraction *x-y*) and the given Λ_3 .

Onr can define corresponding **minimitive functions** with a ***minimitive-norm-unary-function-constant*** using the logical ***Axiom*** of implicit definition of unary functions by a ***UNEX-norm-unary-formulo*** .

4. Processive functions and non-calculative functions

One has to start with an exact definition of ***scheme*** strings which are ***pattern*** strings with at least one ***variable*** , the count of different ***variable*** strings gives the arity of the ***scheme*** .

pattern **::** ***number*** ¦ ***variable*** ¦ ***pattern*** ( ) ¦ ***pattern*** (***pattern-array***)
pattern-array **::** ***pattern*** ¦ ***pattern-array*** ; ***pattern***

A ***scheme*** where every left-paranthesis ' ('is preceded by a ***number*** is a ***primitive-scheme***. All other ***scheme*** strings are called ***processive-scheme*** . Examples:

primitive-scheme	0(Λ1)	201(Λ3)	123(Λ1)	202220120113(1(Λ2);Λ1)
processive-scheme	Λ1(0)	Λ1(Λ1)	Λ1(Λ2)(Λ3)	22011(Λ2(5);Λ4)

Every ***scheme*** defines a **compinitive** function, that is either primitive by a ***primitive-scheme*** or processive by a ***processive-scheme*** (where ***processive-scheme*** may also result in primitive functions, just take the trivial case where the ***scheme***r that precedes the left-paranthesis is evaluated to give 0 or not a ***pinon***). All compinitive functions are calculative. It is easy to define a non-calculative function by a ***UNEX-formulo*** . Take as an example the unary **lazy-slothy-function**[1] ΛLAZY-SLOTHY(Λ) that is given by ***UNEX-formulo*** using ***binary-scheme*** $\Lambda_1(\Lambda_2)$

$\Lambda\textit{lazy-sloth} = [[\exists \Lambda_2[\Lambda_1(\Lambda_2)=0]] \wedge [\Lambda_0=0]] \vee [[\forall \Lambda_2[\Lambda_1(\Lambda_2) \neq 0]] \wedge [\Lambda_0=1]]$

It is almost always 0 and only sometimes gets 1 (the first time for Λ_1=1 , the next time for Λ_1=8109), but it cannot be determined if the binary function given by$\Lambda_1(\Lambda_2)$ is regular (i.e. has a zero). This corresponds to the so-called **busy-beaver-function** of Turing-machines. One should have in mind that the uncalculability of the busy-beaver-function has nothing to do with its tremendous growth but only with the unsolvability of the halting problem. The lazy-slothy-function does not grow at all.

[1] an ***extra-unary-function-constant*** is denoted by ***sort capital-medium-letter-word*** (***sort***)

The existence of non-calculative functions is beyond doubt if one accepts the logical ***Axiom*** of implicit definition of unary functions by a ***UNEX-norm-unary-formulo*** . In this connection there are two interesting questions:

- are there calculative functions that are not recursive or compinitive (***E*** not empty)
  (thus contradicting Church's thesis, see download Snark1.1.pdf ' Snark, …' that can be obtained on homepage https://pai.de/Church-s-thesis/Snark-counterexample)
- are there processive functions that are not recursive (***D*** not empty);
  this will be discussed in section 6 .

<table>
<tr><td colspan="5">calculative functions</td></tr>
<tr><td colspan="3">recursive</td><td></td><td></td></tr>
<tr><td>primitive
A</td><td>minimitive
B</td><td>minimitive and processive
C</td><td>processive
D</td><td>metacursive
E</td></tr>
<tr><td></td><td></td><td colspan="3">transcursive</td></tr>
<tr><td></td><td></td><td colspan="2">compinitive</td><td></td></tr>
<tr><td colspan="4">progressive</td><td></td></tr>
</table>

Diagram: classification of calculative functions with respect to concrete calcule LAMBDA

5. Representation of recursive functions in Robinson arithmetic

The ontological basis of concrete calcule ALPHA of Robinson decimal natural number arithmetic comprises the following ingredients:

sort ::	A	*capital Greek letter, read ALPHA*
cipher ::	1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9	
positive-number ::	***cipher*** ¦ ***positive-number*** 0 ¦ ***positive-number cipher***	
number ::	0 ¦ ***positive-number***	
basis-function-constant ::	A′ ¦ (A+A) ¦ (A×A)	*succession, addition, multiplication*
basis-relation-constant ::	A<A	*minority*

The following ***Basiom*** strings correspond to ***Axiom*** strings of an abstract calcule (*the difference of the two notions is not discussed here*):

B1 $\forall A_1[A_1' \neq 0]$
B2 $\forall A_1[A_2[[A_1'=A_2']\rightarrow[A_1'=A_2]]]$
B3 $\forall A_1[[A_1\neq 0]\rightarrow[\exists A_2[A_1=A_2']]]$
B4 $\forall A_1[(A_1+0)=A_1]$
B5 $\forall A_1[A_2[(A_1+A_2')=(A_1+A_2)']]$
B6 $\forall A_1[(A_1\times 0)=0]$
B7 $\forall A_1[A_2[(A_1\times A_2')=((A_1\times A_2)+A_1)]]$
B8 $\forall A_1[\neg[A_1<0]]$
B9 $\forall A_1[[0=A_1]\vee[0<A_1]]$
B10 $\forall A_1[A_2[[A_1<A_2]\leftrightarrow[[A_1'=A_2]\vee[[A_1'<A_2]]]]]$
B11 $\forall A_1[A_2[[A_1<A_2']\leftrightarrow[[A_1<A_2]\vee[[A_1=A_2]]]]]$

(By the way: the appendix contains an even weaker abstract calcule of natural number arithmetic)

At first sight ALPHA is a very poor calcule. The only functions that can be defined explicitely are multinomials. It is better with ***formula*** and ***formulo*** strings, where at least it can be expressed that a ***number*** is a prime and one can express by a ***sentence*** string that there is no largest prime. However you will run into trouble of proving the Euclid ***THEOREM*** in ALPHA .

But the idea of theory of numbers is to prove it in a stronger calcule and **represent** (see beginning of section 3) the result in ALPHA . Let's exemplify for the unary case what is meant by **representation** of functions by ***UNEX-formulo*** strings as they have been introduced in section 1 for calcule LAMBDA together with the logical ***Axiom*** of **implicit definition of functions** ; everything stays the same in ALPHA or in any other calcule, it is all purely logical (no special features of the calculs do appear).

The standard problem in connection with functions is **composition**. Two unary functions that are represented by two ***UNEX-unary-norm-formulo*** strings A_1 and A_2 produce a new ***UNEX-unary-norm-formulo*** strings A_3 by composition, where ***variable*** strings A_4 is not contained in A_1 or A_2

$$A_3 = \exists A_4[[(A_2 ; A_0 \int A_4)] \wedge [(A_1 ; A_1 \int A_4)]]$$

So far this is applies in every calcule. If it comes to represent the recursive functions of calcule LAMBDA in calcule ALPHA one also has to represent straight recursion and minimization.

Minimization poses no problem in representing except that it is a problem by itself as it rests on the undecidable question if a function is regular. But this is already the problem in LAMBDA . Lets do minimization in the simplest case i.e. for a ***UNEX-binary-norm-formulo*** string A_1 to produce a ***UNEX-unary-norm-formulo*** string A_2 using ***variable*** strings A_3 that is not contained in A_1 .

$$A_2 = [((A_1 ; A_0 \int 0) ; A_2 \int A_0)] \wedge [\forall A_3[[((A_1 ; A_0 \int 0) ; A_2 \int A_3)] \rightarrow [A_0 < A_3']]]$$

Straight recursion is a different story. However, it was Gödel's ingenious invention of so-called **beta-function technique** that laid a way out. There are various ways to code a **suite** (i.e. a finite sequence) of numbers by two numbers, the code-number and the arity-number if some kind of recursion is available. But now we are heading for recursion and have the very limited supply of three Robinson functions only. And yet one can do it. A suite of numbers of arity A_5 can be coded by two number A_1 and A_2 so that the value at position A_3 is given by the number A_0 using a ***UNEX-ternary-norm-formulo*** that represents the beta-function *gbeta(b,c,i)=divisionremainder(b,c(i+1)+1)* in conventional notation (which has to be used for the moment as there is no way talk about suites in ALPHA). The **beta-function lemma** states that the constituents of a suite $x_0, x_1, \ldots, x_{a-1}$ of arity a can be obtained by two code numbers b and c , applying the beta-function $x_i = gbeta(b,c,i)$ for $i<a$. The ***UNEX-ternary-norm-formulo*** is given as

$$Agbeta = \exists A_4[[(A_0+((A_2 \times A_3')' \times A_4))=A_1] \wedge [A_0<(A_2 \times A_3')']] \qquad \text{implicitely limited } \exists A_4[[A_4<A_1] \rightarrow$$

With the inclusion of beta-function technique the window also opens to talk about suites of numbers. This means one can e..g. express the following ***THEOREM*** strings in ALPHA :

- **Fermat's last theorem**
  It needs beta-function technique for expressing x^n strings
- **Euclid's theorem of unlimiuted prime numbers**
  It needs beta-function technique for expressing suites of prime numbers
- **Fundamental theorem of arithmetic prime decomposition**
  It needs beta-function technique for expressing suites of prime numbers and suites of prime-power-products.

By the way: there is a very fundamental difference between minimization and application of beta-function technique. In minimization it is not guaranteed if there exists a zero, whereas in beta-function application there always exist two code numbers, one just cannot give a majorant.

Let's return to ***UNEX-formulo*** strings and straight recursion: treating it in the second simplest, the binary case, again starting in conventional notation. A function *f(x,y)* is to be constructed such that *f(0,n)=g(n* and *f(m+1,n)=h(f(m),m,n)* with a unary starting function *g(x)* and a ternary iteration function *h(x,y,z)*. Let the two functions *g(n)* and *h(x,y,z)* be given by a ***UNEX-unary-norm-formulo*** A_1 and ***UNEX-ternary-norm-formulo*** A_2 , where both of them and ***Agbeta*** do not contain ***variable*** strings A_3 , A_4 , A_5 , A_6 and A_7 . The ***UNEX-binary-norm-formulo*** for straight recursion is constructed with the application of beta-function technique (in the second expression the ***Agbeta*** is expanded) :

$\exists A_3[\exists A_4[[[\forall A_5[[((((Agbeta;\ A_1 \int A_3);\ A_2 \int A_4);\ A_3 \int 0);\ A_0 \int A_5)\ ] \wedge [((A_1;\ A_1 \int A_2);\ A_0 \int A_5)\ ]]] \wedge$
$[[0<A_1] \rightarrow [\forall A_5[[A_5<A_1] \rightarrow [\forall A_6[\forall A_7[[[((((Agbeta;\ A_1 \int A_3);\ A_2 \int A_4);\ A_3 \int A_5);\ A_0 \int A_6)\ ] \wedge$
$[((((A_2;\ A_1 \int A_6);\ A_2 \int A_5);\ A_3 \int A_2);\ A_0 \int A_7)\ ]] \wedge [((((Agbeta;\ A_1 \int A_3);\ A_2 \int A_4\ );\ A_3 \int A_5');\ A_0 \int A_7)\ ]]]]]]] \wedge$
$[(((Agbeta;\ A_1 \int A_3);\ A_2 \int A_4);\ A_3 \int A_1)\ ]]]$

$\exists A_3[\exists A_4[[[\forall A_5[[\exists A_4[[(A_0+(A_4' \times A_4))=A_3] \wedge [A_5<A_4']]] \wedge [((A_1;\ A_1 \int A_2);\ A_0 \int A_5)\ ]]] \wedge$
$[[0<A_1] \rightarrow [\forall A_5[[A_5<A_1] \rightarrow [\forall A_6[\forall A_7[[[\exists A_4[[(A_6+((A_4 \times A_5')' \times A_4))=A_3] \wedge [A_6<(A_4 \times A_5')']]] \wedge$
$[((((A_2;\ A_1 \int A_6);\ A_2 \int A_5);\ A_3 \int A_2);\ A_0 \int A_7)\ ]] \wedge [\exists A_4[[(A_7+((A_4 \times A_5'')' \times A_4))=A_3] \wedge$
$[A_7<(A_4 \times A_5'')']]]]]]]]] \wedge [\exists A_4[[(A_0+((A_4 \times A_1')' \times A_4))=A_3] \wedge [A_0<(A_4 \times A_1')']]]]]$

The **generalization to functions of all arities** is tedious but straightforward. One has to treat the nullary case, the unary case and the multary case separately.This means that one can represent all recursive functions by ***UNEX-norm-formulo*** strings: A ***unex-norm-formulo***[1] strings is a ***UNEX-norm-formulo*** string where the proof of unique existence is trivial. *One can make this definition precise, but for the moment this should be enough.* One starts from the following trivial ***unex-norm-formulo*** strings that represent nullification, succession and projections for all arities:

nullary	unary	binary	ternary	…
$A_0=0$	$[A_0=0] \wedge [A_1=A_1]$	$[A_0=0] \wedge [[A_1=A_1] \wedge [A_2=A_2]]$	$[A_0=0] \wedge [[A_1=A_1] \wedge [[A_2=A_2]] \wedge [A_3=A_3]]]$	
	$A_0=A_1$	$[A_0=A_1] \wedge [A_2=A_2]]$	$[A_0=A_1] \wedge [[A_2=A_2]] \wedge [A_3=A_3]]$	
		$[A_0=A_1'] \wedge [A_2=A_2]$	$[A_0=A_1'] \wedge [A_2=A_2]] \wedge [A_3=A_3]]$	
		$[A_1=A_1] \wedge [A_0=A_2']$	$[[A_1=A_1] \wedge [A_2=A_2]] \wedge [A_0=A_3']$	

If one then successively applies **compositions** and **straight recursion** as defined above for ***UNEX-norm-formulo*** strings (and extends it to all arities ,which is a tedious but feasible job) one obtains the so-called ***unex-primitive-norm-formulo*** strings. They represent the **primitive recursive functions** in ALPHA .

If one successively applies **compositions** , **straight recursion** and **minimization** t***o*** the starting ***unex-norm-formulo*** strings one obtains the so-called ***UNEX-recursive-norm-formulo***[2] strings. They represent the **recursive functions** in ALPHA that besides primitive functions also include minimitive functions that are obtained by minimization.

Now that one can represent the primitive recursive functions and the recursive functions in ALPHA one can ask: can one represent every ***sentence*** string of LAMBDA as a ***sentence*** string of ALPHA ? The answer is astonishing:

Every ***sentence*** string of LAMBDA that comprises only primitive recursive functions and minimizations and compostions thereof can be represented as a ***sentence*** string of ALPHA as everything can be expressed properly with ***UNEX-formulo*** technique.

However, what happens, when processive functions appear in LAMBDA ? The answer is given in the next section. Finally:

How about the ***THEOREM*** strings of LAMBDA that comprise only primitive recursive functions and minimizations. As the corresponding ***sentence*** strings of ALPHA are ***THEOREM*** strings, one can ask, where the ***TRUTH*** of these ***sentence*** strings comes from. It certainly is **not obtained by derivation** from the Robinson ***Basiom*** strings, but rather from the outside through metalingual reasoning. This is a great step and a very deep feature, that needs further discussion (but not in this publication).

[1] small Latin letter indicates that no problem of ***TRUTH*** is involved [2] capital Latin letter indicates that ***TRUTH*** is involved

6. Representation of processive functions in Robinson arithmetic ?

Processive functions of concrete calcule LAMBDA of decimal pinitive arithmetic have been introduced in section 2 and have been discussed in section 4. Now that the concept of representation of functions and sentences of one calcule in another calcule has been introduced for the examples LAMBDA and ALPHA one can discuss what processive functions mean in connection with the concrete calcule ALPHA of Robinson decimal natural number arithmetic.

Representing processive functions in ALPHA would necessitate the construction of adequate ***UNEX-formulo*** strings.

In the preceding section it was shown how to construct adequate ***unex-formulo*** strings for primitive recursive functions, meaning that one can do even better than constructing ***UNEX-formulo*** strings. It was based on trivial starting ***unex-formulo*** strings for nullifications, successions and projections and successive application of compositions and straight recursions, the latter with beta-function technique.

However, there is no correponding way to construct ***UNEX-formulo*** strings for processive functions, as they are defined by ***processive-scheme*** strings, meaning that they can only be talked about metalingually, although every one of them is perfectly admissible and expressible in object-language.

This leads back to the diagram at the end of section 4 and the preceding question if there are processive functions that are not recursive (***D*** not empty). The following is not a conjecture for a ***sentence*** of ALPHA but rather a conjecture in metalanguage.

metaconjecture:	there are processive functions that cannot be represented in ALPHA (***D*** is not empty)

There are two possibilities for a proof of the contrary in metalanguage (a metaproof):

- give a construction of a recursive function for every processive functions
- or show the weaker metatheorem that there exists a recursive function for every processive function.

Given Kleene's normal form these two possibilties amount to the following:

- for every processive functions of arity *a* one has to construct two ***pinon*** numbers, one for a regular characteristic primitive function of arity *a+1* (that has to be minimized) and one for a unary primitive function that is to be applied subsequently
- or one has at least to show the existence of such two ***pinon*** numbers for every processive function.

As long as this **challenge** is in the open one may say that not all calculative functions can be represented in the concrete calcule ALPHA of Robinson decimal natural number arithmetic

Appendix Abstract calcule alphakappa of Robinson-Crusoe natural number arithmetic

Based on the observation that one only needs the ***unex-formulo*** technique for representaion of functions in concrete calcule ALPHA one remembers equation $(x+y)^2=x^2+y^2+2xy$ (in classical notation) to produce an eaven weaker calcule. This time the **abstract** counter piece is introduced;

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

sort ::	$\alpha\kappa$	
basis-individual-constant ::	$\alpha\kappa n$	*nullum*
basis-function-constant ::	$\alpha\kappa'$ ¦ $(\alpha\kappa+\alpha\kappa)$ ¦ $(\alpha\kappa\uparrow)$	*succession, addition, quadration*
basis-relation-constant ::	$\alpha\kappa<\alpha\kappa$	*minority*

with ***Axiom*** strings:

A1	$\forall\alpha\kappa_1[\alpha\kappa_1'\neq\alpha\kappa n]$
A2	$\forall\alpha\kappa_1[\alpha\kappa_2[[\alpha\kappa_1'=\alpha\kappa_2']\rightarrow[\alpha\kappa_1'=\alpha\kappa_2]]]$
A3	$\forall\alpha\kappa_1[[\alpha\kappa_1\neq\alpha\kappa n]\rightarrow[\exists\Lambda_2[\alpha\kappa_1=\alpha\kappa_2']]]$
A4	$\forall\alpha\kappa_1[(\alpha\kappa_1+\alpha\kappa n)=\alpha\kappa_1]$
A5	$\forall\alpha\kappa_1[\alpha\kappa_2[(\alpha\kappa_1+\alpha\kappa_2')=(\alpha\kappa_1+\alpha\kappa_2)']]$
A6	$\forall\alpha\kappa_1[(\alpha\kappa n\uparrow)=\alpha\kappa n]$
A7	$\forall\alpha\kappa_1[(\alpha\kappa_1'\uparrow)=(((\alpha\kappa_1\uparrow)+\alpha\kappa_1)+\alpha\kappa_1)']$
A8	$\forall\alpha\kappa_1[\neg[\alpha\kappa_1<\alpha\kappa n]]$
A9	$\forall\alpha\kappa_1[[\alpha\kappa n=\alpha\kappa_1]\vee[\alpha\kappa n<\alpha\kappa_1]]$
A10	$\forall\alpha\kappa_1[\alpha\kappa_2[[\alpha\kappa_1<\alpha\kappa_2]\leftrightarrow[[\alpha\kappa_1'=\alpha\kappa_2]\vee[[\alpha\kappa_1'<\alpha\kappa_2]]]]$
A11	$\forall\alpha\kappa_1[\alpha\kappa_2[[\alpha\kappa_1<\alpha\kappa_2']\leftrightarrow[[\alpha\kappa_1<\alpha\kappa_2]\vee[[\alpha\kappa_1=\alpha\kappa_2]]]]$

extra-individual-constant ::	$\alpha\kappa u=\alpha\kappa n'$	*unus*

One uses the following ***binary-norm-unex-formulo*** for the introduction of multiplication

$((\alpha\kappa_1+\alpha\kappa_2)\uparrow)=((((\alpha\kappa_1\uparrow)+(\alpha\kappa_2\uparrow))+\alpha\kappa_0)+\alpha\kappa_0)$

It is achieved by an

extra-function-constant ::	$(\alpha\kappa\times\alpha\kappa)$	*multiplication*

which is introduced by application of the logical ***Axiom*** of implicit definition of unary functions by ***UNEX-formulo*** as given in section 1 :

$\forall\alpha\kappa_1[\alpha\kappa_2[(\alpha\kappa_1+\alpha\kappa_2)\uparrow)=((((\alpha\kappa_1\uparrow)+(\alpha\kappa_2\uparrow))+(\alpha\kappa_1\times\alpha\kappa_2))+(\alpha\kappa_1\times\alpha\kappa_2))]]$

All the ***UNEX-formulo*** strings get a little lengthier as compared to section 5. But everything is representable: Gödel's beta-function, straight recursion and so on.

No tremendous progress but funny.