PRACE IPI PAN ICS PAS REPORTS

Stanisław Ambroszkiewicz

Types and Operations

Nr 1030

INSTYTUT PODSTAW INFORMATYKI POLSKIEJ AKADEMII NAUK
INSTITUTE OF COMPUTER SCIENCE POLISH ACADEMY OF SCIENCES
ul. Jana Kazimierza 5, 01-248 Warszawa http://www.ipipan.waw.pl

Stanisław Ambroszkiewicz

Types and Operations

Nr 1030

Prof. Jerzy Łoś (1920 - 1998)

The work is dedicated to the memory of Professor Jerzy Łoś.

Institute of Computer Science
Polish Academy of Sciences

Warsaw, December 2014

Pracę zgłosił Prof. dr hab. inż. Wojciech Penczek

Adres autora:	Instytut Podstaw Informatyki PAN ul. Jana Kazimierza 5 01-248 Warszawa Polska
E-mail:	`sambrosz@ipipan.waw.pl`

Symbol klasyfikacji rzeczowej MSC 2000: 03D99, 68Q05

Na prawach rękopisu
Printed as manuscript

Nakład 100 egzemplarzy. Papier kserograficzny klasy III. Oddano do druku w grudniu 2014 r. Wydawnictwo IPI PAN. ISSN 0138-0648

Abstract

Types and Operations

A revision of the basic concepts of type, function (called here operation), and relation is proposed. A simple generic method is presented for constructing operations and types as concrete finite structures parameterized by natural numbers. The method gives rise to build inductively so called Universe intended to contain all what can be *effectively constructed* at least in the sense assumed in the paper. It is argued that the Universe is not yet another formal theory but may be considered as a grounding for some formal theories.

Keywords: types, semantics, foundations

Streszczenie

Typy i Operacje

Zaproponowana została rewizja podstawowych pojęć typu i funkcji (nazywanej tutaj operacją). Typy, obiekty tych typów oraz operacje są konstruowane za pomocą prostej i uniwersalnej metody jako skończone struktury parametryzowane liczbami naturalnymi. Metoda ta pozwala na budowanie tzw. Uniwersum, które, w zamierzeniu, ma zawierać wszystko co jest *efektywnie konstruowalne* przynajmniej w sensie, jaki jest przyjęty w tej pracy. Przedstawione są argumenty, że Uniwersum nie jest jeszcze jedną formalną teorią, lecz może służyć jako ugruntowanie dla pewnych formalnych teorii.

Słowa kluczowe: typy, semantyka, podstawy

1 Introduction

It is a continuation of the work of Professor Andrzej Grzegorczyk [16] (who was inspired by the System T of Kurt Gödel [12]) concerning recursive objects of all finite types.

The phrase *effectively constructed objects* may be seen as a generalization of the notion of *recursive objects*. Objects can be represented as finite (usually parameterized) structures. Universe is understood here as a collection of all generic constructible objects.

In the Universe, constructability is understood literally, i.e. it is not definability, like general recursive functions (according to Gödel-Herbrand) that are defined by equations in Peano Arithmetic along with proofs that the functions are global, that is, defined for all their arguments. Objects are not regarded as terms in lambda calculus or in combinatory logic.

Most theories formalizing the notion of effective constructability (earlier it was computability) are based on the lambda abstraction introduces by Alonzo Church that in principle was to capture the notion of function and computation. Having a term with a free variable, it is easy to make it a function by applying lambda operator. Unlimited application of lambda abstraction results in contradiction (is meaningless), i.e. some terms cannot be reduced to the normal form. This very reduction is regarded as computation. Introduction of types and restricting lambda abstraction only to typed variables results in a very simple type theory.

Inspired by System T, Jean-Yves Girard created system F [11], [10]; independently also by John C. Reynolds [31]. Since System F uses lambda and Lambda abstraction (variables run over types as objects), the terms are not explicit constructions. System F is very smart in its form, however it is still a formal theory with term reduction as computation; it has strong normalization property.

Per Martin-Löf Type Theory (ML TT for short) [28] was intended to be an alternative foundation of Mathematics based on constructivism asserting that to construct a mathematical object is the same as to prove that it exists. This is very close to the Curry-Howard correspondence *propositions as types*. In ML TT, there are types for equality, and a

cumulative hierarchy of universes. However, ML TT is a formal theory, and it uses lambda abstraction. Searching for a grounding (concrete semantics) for ML TT by the Author long time ago, was the primary inspiration for the Universe presented in this work.

Calculus of Inductive Constructions (CoIC), created by Thierry Coquand and Gérard Huet [5] and [6], is a lambda calculus with a rich type system like the System F. It was designed for *Coq Proof Assistant* [4], and can serve as both a typed programming language and as constructive foundation for Mathematics. Agda is a dependently typed functional programming language based also on ML TT; it is also a proof assistant, see at www wiki.portal.chalmers.se/agda/

ML TT, System F, and CoIC are based on lambda and Lambda abstraction, so that in their syntactic form they correspond to the term rewriting systems.

In this work lambda and Lambda abstractions are challenged. It is an attempt to show that the same (as in System F), and perhaps more, can be achieved by explicit and concrete constructions, even though these constructions are not so concise and smart as the corresponding terms in System T. The proposed method relates rather to the approach where explicit constructors are used. In this sense, it continues the idea of Grzegorczyk's combinators [16], and in some sense also combinators in Haskell B. Curry [8] combinatory logic. Note that combinatory logic is the theoretical basis for functional programming language Haskell.

Although the whole Informatics can be reduced to processing of streams of bytes (it is still a common view), in programming more and more sophisticated data structures are used that cannot be identified with syntactical operations on strings of signs (terms) according to fixed simple (term rewriting) rules. It seems that lambda calculus is a way to only roughly describe such sophisticated structures. From the constructivist point of view, changing of all occurrences of a free variable (in a term) with another term of the same type as the variable, is not obvious; something of this intuition was captured by the Girard's linear logic. Arbitrary application of Lambda abstraction and substitution may result with terms that have no computational meaning, i.e. cannot be reduced to the normal form.

Effective construction of an object cannot use actual infinity. If it is an inductive construction, then the induction parameter must be shown explicitly in the construction. For any fixed value of the parameter the construction must be a finite structure. The Universe presented in this paper is supposed to consist only of such objects. Objects are not identified with terms whereas computations are not term rewritings. Although, in computations all can be reduced to the primitive types, higher order types and their objects correspond in programming to sophisticated data structures and their instances.

Two primitive types are considered: natural numbers and Continuum. It seems that the Continuum as a primitive type is novel in Informatics. The inspiration comes from quite recent (November 2013) Homotopy Type Theory: Univalent Foundations of Mathematics (HoTT) [39]; a formal theory based on ML TT and CoIC. HoTT aspires to be another foundation for Mathematics alternative to set theory (ZFC), by encoding general mathematical notions in terms of homotopy types. According to Vladimir Voevodsky [40](one of the creators of HoTT) the univalent foundations are adequate for human reasoning as well as for computer verification of this reasoning. Generally, any such foundations should consist of three components. The first component is a formal deduction system (a language and rules); for HoTT it is CoIC. The second component is a structure that provides a meaning to the sentences of this language in terms of mental objects intuitively comprehensible to humans; for HoTT it is interpretation of sentences of CoIC as univalent models related to homotopy types. The third component is a way that enables humans to encode mathematical ideas in terms of the objects directly associated with the language.

The above phrases: *mental objects* and *mathematical ideas* are not clear. Actually, in the univalent foundations, these mental objects (as homotopy types) are yet another formal theory. It seems that the main problem here is the lack of a grounding (concrete semantics) of these mental objects and mathematical ideas. The concept of equality (relation) plays extremely important role in ML TT and HoTT. However, a formal axiomatic description of the notion of equality of two object of the same type, and then higher order equality is not sufficient to com-

prehend the essence of the notion of equality and in general of the notion of relation. The homotopy origins of HoTT are interesting and will be discussed in the Section 6, whereas a grounding for the notion of relation is proposed in Section 9.

The proposed Universe is not yet another formal theory of types. It is intended to be a grounding for some formal theories as well as a generic method for constructing objects corresponding to data structures in programming.

It seems that the same idea was investigated at least since the beginning of the XX century. However, it was done in formal ways by Church lambda calculus, Curry combinatory logic, Gödel System T, Grzegorczyk System, Martin Lof TT, Girard System F, and Coquand CoIC to mention only the most prominent works. The Universe is an attempt to understand these formal theories. Hence, once it is presented, it should be grasped as simple and obvious.

Universe is strongly related to computable functionals (Stephen C. Kleene [23][24][25], Georg Kreisler [26], Grzegorczyk [13] and [14], as well as to Richard Platek & Dana Scott PCF^{++} [35, 34]) and to Scott Domain [36].

2 Foundations

This section and the next one may be seen as naive because there are no technicalities here. However, the aim is to present the most primitive notions as simple as possible. These notions (as basic elements for constructions) are *types*, *objects*, and *operations* that process input objects into output objects. Operation is a synonym for function.

Object a of type A is denoted by $a : A$. The type of operation is determined by the type of its input (say A) and the type of its output (say B), and is denoted by $A \to B$.

Primitive type may be interpreted as data link (communications channel) whereas object of that type as a signal transmitted in this channel. The interpretation may be extended for complex types.

Simple operation has one input and one output, however in general, operation may have multiple inputs as well as multiple outputs, see Fig.

1. The type of operation g having multiple inputs and multiple outputs is denoted by $g : (A_1; A_2; ...; A_k) \rightarrow (B_1; B_2; ...; B_l)$

Operation $f : A \rightarrow B$ may be applied to its argument, i.e. input object $a : A$. The output of this application is denoted by $f(a)$. For operations with multiple input, application may be partial, i.e. only for some of the inputs (say a_i and a_j). Then it is denoted by $g(a_j; a_i; *)$. Application is amorphous, however, if the type of the operation and the types of arguments are fixed, then application may be considered as an operation. There are no variables in our approach. In lambda calculus

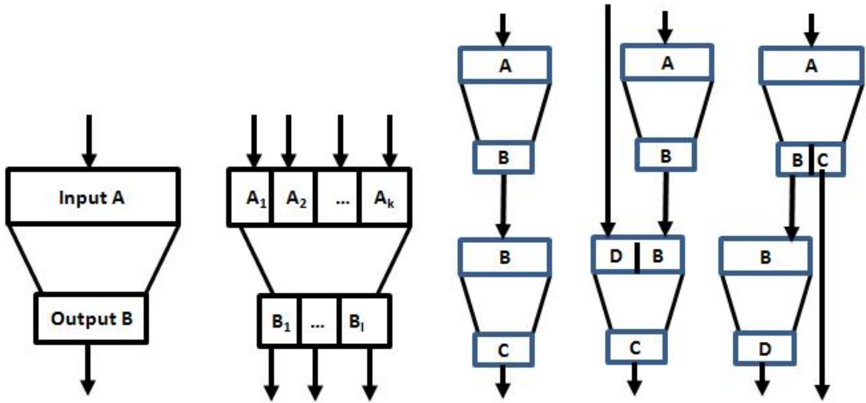

Figure 1: On the left, graphical schema of operation; on the right, composition of two operations.

variables serve to denote inputs. In combinatory logic any combinator has exactly one input. Operation having many inputs can be (equivalently in some sense) transformed (by *currying*) into operation having one input. It will turn out in Section 4.3 that currying is an operation.

Composition of two operations consists in joining an output of one operation to an input of another operation. The type of the output and the type of the input must be the same, see Fig. 1. Composition is amorphous, however if the operation types are fixed, then composition may be considered as an operation.

The Universe will be developed inductively (actually, by transfinite

induction) starting with primitive constructors, destructors and primitive types. At each inductive level, new primitives will be added. The primitives are natural consequences of the constructions methods from the previous levels and give rise to new methods. Each level is potentially infinite. The Universe is never ending story. Once construction methods are completed for one level, it gives rise to the construction of the next level and new methods. There are always next levels that contribute essential and qualitatively new constructions to the Universe.

This constitutes a bit intuitive and informal foundation for the Universe. In the next sections the idea is developed fully and precisely.

3 Level zero

Level 0 of the Universe consists of primitive constructors of types, primitive types, and related primitive operations. On the level 1, the types form level 0 will be treated as objects, analogously for higher levels. The levels of the Universe correspond to an infinite well-founded typing hierarchy of sorts in CoIC [4].

3.1 Type constructors

Keeping in mind the interpretation of types as telecommunication links, there are three basic type constructors. Let A and B denote types.

- $\times$ product of two types $A \times B$; as one double link consisting of A and B. Signals (objects) are transmitted in $A \times B$ simultaneously.

- $+$ disjoin union $A + B$; two links are joined into one single link. Signal (object) transmitted in this link is preceded by a label indicating the type of this object.

- $\rightarrow$ arrow, operations type $A \rightarrow B$; A is input type, whereas B is output type.

These three basic constructors are independent of primitive types. On the level 1 these constructors will be considered as operations on types, and new type constructors will be introduced.

Product and disjoint union are natural and their nesting, like $(A \times B)+(C \times D)$, has obvious interpretation. The meaning of operation type is a bit more harder to grasp, especially if input is again an operation type, like $(A \to B) \to C$. The problem is how to interpret operation as an object. Actually this problem can be reduced to grasping properly what are input types and output types in an operation. Fig. 2 may help. Input is interpreted as socket board where each socket corresponds to a link that is a single type. The same for the output.

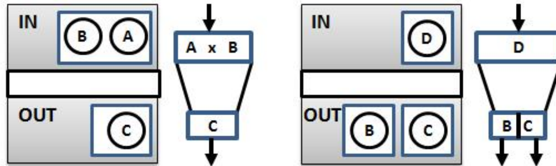

Figure 2: Another pictorial presentation of operations: on the left, there is operation with input socket board consisting of $A \times B$ and one output board consisting of type C; on the right, output socket board consists of two independent types B and C.

Type of operation is again a socket board consisting of two parts. The upper part is a socket board as the input type. The bottom part is a socket board as the output type, see Fig. 3.

Putting an object $a : A$ into a socket of type A is interpreted as a transmission of the object via this socket. For an operation $f : A \to C$ putting an object $a : A$ into the input socket of type A means the application $f(a)$. So that the object $f(a)$ will appear at the output socket of type B.

For operation $f : (B; A) \to C$ the application $f(b, a) : C$ is just an object at the output socket C, see Fig. 4. However, $f(b, *)$ is still an operation of type $A \to C$.

Once this is clear, it is also easy to grasp what does it mean to apply operation $F : (A \to B) \to C$ to an input object $h : A \to B$. The input socket board of the operation F is of type $A \to B$. Putting object h into the input socket board of F means connecting all the sockets of input and output of h to the input board of the operation F, see Fig.

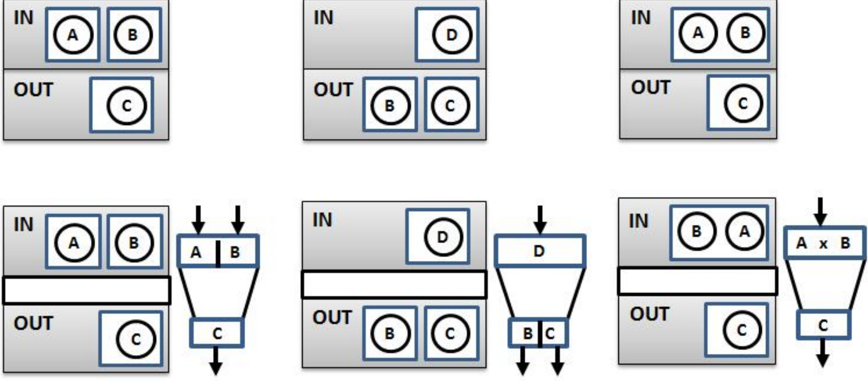

Figure 3: Operations and their types: operations are presented in the bottom row whereas their types in the top row.

5, where input socket of h (of type A) is connected to the input socket (of type A) of the input board of F, whereas output socket of h (of type B) is connected to the output socket (of type B) of the input board of operation F. Once it is done, the result of the application $F(h)$ is at the socket C of the output board of the operation F. This is the crucial point of our approach. Types are interpreted as sockets whereas input types and output types as socket boards. Operation type is interpreted as a complex board consisting of input board and output board. This gives rise to interpret types as objects at the level 1 of the Universe.

Constructors of objects corresponding to product, disjoin union, and arrow (operation type) are as follows. Let $a : A$ and $b : B$.

- for product: $join_{A,B}$ is an operations of type $(A; B) \to (A \times B)$ such that $join_{A,B}(a; b)$ is an object of type $A \times B$ denoted as a pair by (a, b).

- for disjoin union: $plus^A_{A,B} : A \to (A + B)$ and $plus^B_{A,B} : B \to (A + B)$. For $a : A$ and $b : B$, $plus^A_{A,B}(a)$ and $plus^B_{A,B}(b)$ are objects of type $A + B$.

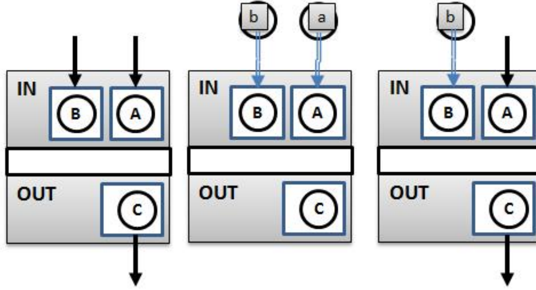

Figure 4: An operation applied to objects.

- for arrow: for any $a : A$ there is the constant operation of type $B \rightarrow A$, such that for any $b : B$, it returns a as its output. More generally, $const_{A,B} : A \rightarrow (B \rightarrow A)$, such that operation $const_{A,B}(a) : B \rightarrow A$ returns always a as its output.

It is important that the equality symbol "=" in not used even for description. The reason is that the equality as relation will be constructed in Section 9.

Destructors for product, disjoin union, and arrow.

- $proj_{A,B} : (A \times B) \rightarrow (A; B)$. For any (a, b) of type $A \times B$, projection returns two output objects denoted by $proj^A_{A,B}((a, b)) : A$ and $proj^B_{A,B}((a, b)) : B$. Composition of $join_{A,B}$ and $(proj^A_{A,B}; proj^B_{A,B})$ gives two identity operations: $id_A : A \rightarrow A$ and $id_B : B \rightarrow B$, that return the input object as the output. Although identity operation (say for type A) may be identified with a link of type A, it is useful in constructions.

- $get_{A,B} : (A + B) \rightarrow (A; B)$. For an input object of type $A + B$, it returns only one output: either $get^A_{A,B}$ or $get^B_{A,B}$. Although this operation has two outputs, once it is applied, only one output has object; it is determined by the input object.

- $apply_{A \rightarrow B,A}$. Application as operation indexed by types A i B is of type $((A \rightarrow B); A) \rightarrow B$. For any input objects $f : A \rightarrow B$ and

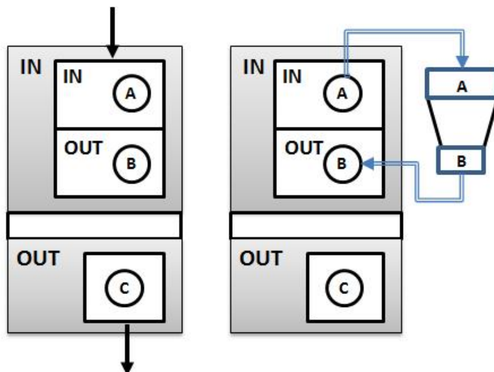

Figure 5: Application of operation $F : (A \to B) \to C$ to object $h : A \to B$.

$a : A$, it returns as the output $apply_{A \to B,A}(f; a)$, i.e. the same as the amorphous application $f(a)$ (being also a destructor for arrow type). Applications as operations are used in constructions.

Operation $apply_{A \to B,A} : ((A \to B); A) \to B$ is interpreted as a specific board consisting of linked sockets, see Fig. 6. Generally, operation *apply* may be more complex, i.e. may have multiple inputs for example, it maybe of type $(((C; A) \to B), C) \to (A \to B)$, see Fig. 6.

3.2 Composition as operation

Composing two operations $f : A \to B$ and $g : B \to C$ means to link the output sockets of f to the input sockets of g. This is done for two fixed operations. If these operations are not fixed, the composition becomes an operation, i.e. $compose_{A,B,C} : ((A \to B); (B \to C)) \to (A \to C)$ and is realized as a specific board with appropriate links between sockets, see Fig. 7.

The simplest composition is of the form: $compose_{A,A,A} : ((A \to A); (A \to A)) \to (A \to A)$; it will be used to construct iteration.

Like application also composition has its more complex variants depending on the numbers of inputs and outputs of the composed operations. An example with two inputs is shown in Fig. 7.

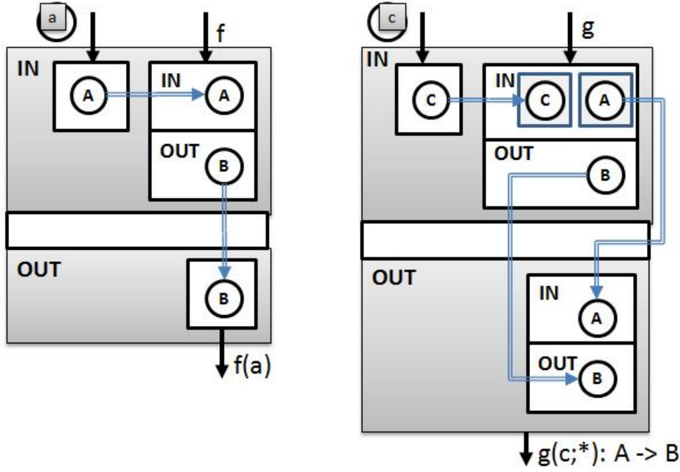

Figure 6: Applications as operations: simple $apply_{A \to B,A} : ((A \to B); A) \to B$, and complex $apply_{(((C;A) \to B),C)}$

For two outputs it maybe of the following form. $compose_{A,(B;D),B,C} : ((A \to (B; D)); (B \to C)) \to (A \to (C; D))$.

3.3 Operation *Copy*

Object is given by its construction. Repeating this construction means to copy this object. For already constructed object a, $Copy(a)$ returns two outputs, the first one, denoted by $Copy^1(a)$, is the original object a, whereas the second output, denoted by $Copy^2(a)$, is a copy of a. $Copy$ is amorphous, however in constructions it may be used as operation $Copy_A : A \to (A; A)$. It is a specific operation determined by what has already been constructed.

Note that once an object was used in a construction, it is an inherent part of this construction, so that it can not be used again in another construction. Operation $Copy$ allows to produce copies that may be used in another constructions. This restriction concerns also primitive operations and types. In this paper we abuse somewhat the notation, and use the same symbols (occurring more than once) that in fact are

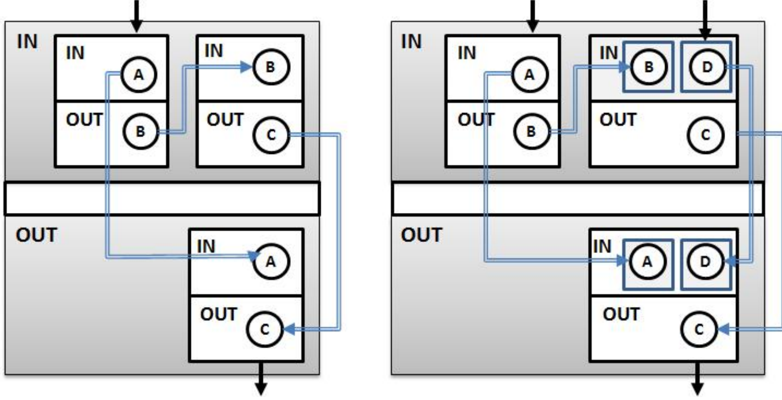

Figure 7: Composition $compose_{A,B,C} : ((A \to B);(B \to C)) \to (A \to C)$, and complex composition $compose_{A,B,(B;D),C} : ((A \to B);((B;D) \to C))) \to ((A;D) \to C)$

different copies of the same type, primitive operation, or object. It is confusing especially if multiple copies of the same type are in input or in output. To distinguish these types the following convention is used. If A denotes a type, then its copies are denoted by A', A'', and so on. Also copies of object a will be denoted by a', a'', a''', and so on. Actually types from level 0 are considered also as objects on level 1.

Although what was presented above is simple and obvious, it constitutes the strictly finitary basis. Next important step is to introduce primitive types.

4 Natural numbers

Copy and add to the previous result is the primeval intuition of natural numbers.

Telecommunication interpretation of this intuition is as follows. Copy the unit signal from the transmission channel, and the result put in the channel at the beginning. Repeating this means natural numbers. Starting with a single unit signal (denoting number 1), the consecutive repetitions give next natural numbers, i.e. the first repetition results in

two unit signals, the second one in three signals, and so on. This very repeating is the successor operation denoted by $Succ$.

If the intuition is applied to an operation of type $A \to A$ (here A is an arbitrary type)) instead of the unit signal, then it is exactly the approach to define natural numbers proposed by Church in his lambda calculus and also the one used in System F. Natural number (say n) is identified with the amorphous iteration, i.e. it can be applied to any operation (with input and output of the same type), and returns n-times composition of this operation. Let us accept the first interpretation.

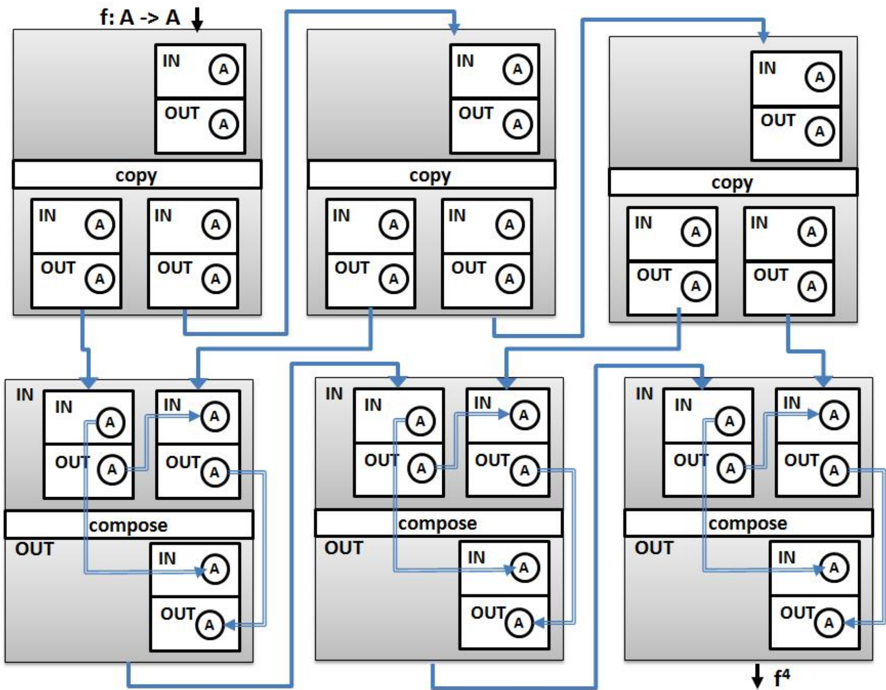

Figure 8: Operation $Iter(4; *)$ applied to f

Then the primitive operations successor $Succ$ and predecessor $Pred$ have natural interpretation. $Succ$ consists in coping the original unit signal and join the result to what has already been done. $Pred$ is interpreted

as removing from the channel the first unit signal, if the channel is not empty.

Denote the type of natural numbers by N.

4.1 Iteration

The type of natural numbers is crucial to construct sophisticated objects.

Iteration (for a fixed type A) is the operation $Iter_A : (N; (A \rightarrow A)) \rightarrow (A \rightarrow A)$ that for a given $n : N$ and operation $f : A \rightarrow A$, returns $Iter_A(n; f)$ that is n-times composition of f.

The parameter $n : N$ determines how many copies of f and copies of $compose_{A,A,A}$ must be used to produce the output by $Iter_A$. Hence, the parameter $n : N$ determines a collection of linked socket boards that are to be assembled into operation $Iter_A(n; *) : (A \rightarrow A) \rightarrow (A \rightarrow A)$. An example for $Iter_A(4; *)$ is shown in Fig. 8.

4.2 Operation *Change*

For a sequence of objects of type A, i.e. operation $q : N \rightarrow A$, and a fixed $a : A$, change n-th element (i.e. $q(n)$) to a. The operation $Change_A$ is of type $(N; A; (N \rightarrow A)) \rightarrow (N \rightarrow A)$, such that $Change_A(n; a; q)(n)$ is a. For k different than n, $Change_A(n; a; q)(k)$ is $q(k)$.

Change corresponds to `if-then` and `case` constructs in programming.

Interpretation of *Change* in the terms of links consists in considering link of the type N with signal n, and the link of type A with object a, together with an operation of type $N \rightarrow A$, and checking the input of the operation. If the input *is the same* as n, then change the output of the operation to a, else do nothing. The phrase *the same* corresponds to a primitive relation on type N that will be constructed in Section 9.

In Section 5, *Change* is needed to construct operations that correspond to primitive recursion schemata on higher types, i.e. Grzegorczyk's iterators.

Change, *Iter* and *Copy* are not simple and their realizations depend on the input objects. The rest of the primitive operations have interpretations as static socket boards dependent only of the types, not on the

input objects.

4.3 Currying

Currying is a syntactical rule to transform a term denoting function with two or more variable (inputs) to equivalent (nested) term with one outer variable; the other variables are hidden inside the term. It was introduced by Moses Schönfinkel in 1924 and later developed by H. Curry.

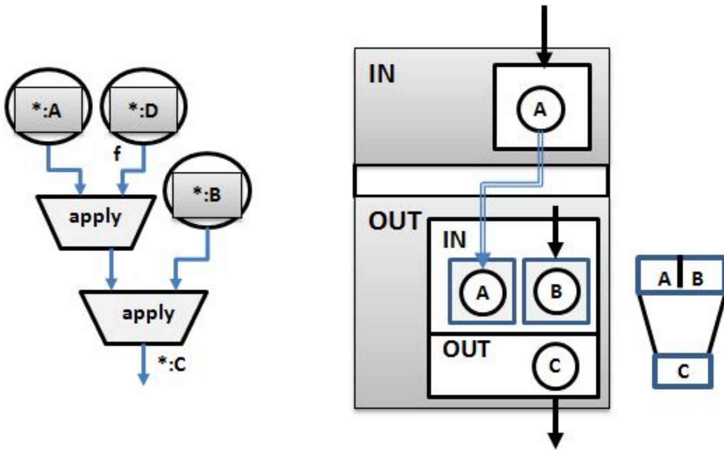

Figure 9: On the left, the construction of operation h corresponding to *uncurrying*; here D denotes $A \to (B \to C)$. On the right, the construction of *currying*.

Operation $f : (A; B) \to C)$ is transformed by currying into operation $g : A \to (B \to C)$, such that $f(a; b)$ denotes the same object as $g(a)(b)$.

Currying as well as *uncurrying* (i.e. the reverse transformation dual to currying) can be represented as operations, i.e. interpreted as socket boards.

The construction of *uncurrying* as an operation of type $(A \to (B \to C)) \to ((A; B) \to C)$ is presented in Fig. 9. Compose $apply_{(A \to (B \to C)),A}$ and $apply_{(B \to C),B}$ by linking the output (of type $B \to C$) of the first application to the one input (of type $B \to C$) of the second application.

Denote this composition by h; it is of type $((A \to (B \to C)); A; B) \to C)$. It has three inputs. Once h is applied only to the first input (say $f : A \to (B \to C)$), it returns $h(f; *; *) : (A; B) \to C$.

currying as an operations of type $((A; B) \to C) \to (A \to (B \to C)$ is a bit strange construction shown in Fig. 9. It is the operation consisting of the input socket of type A, and the output socket board of type $(A; B) \to C$. Input socket A is linked to the socket A in the output board. Putting operation $f : (A; B) \to C$ into the *output board*(!) results in equivalent operation of type $A \to (B \to C)$. Note that usually input object is put into input. However, here the object is put into output.

Currying and uncurrying transform an operation into equivalent operation of different type. This equivalence will be discussed in a broader context in the Section 9.7.

5 Constructability and primitive recursive objects of all finite types

The schema of primitive recursion for operations of the first order (from natural numbers into natural numbers) is clear. However, it is not so obvious for operations of higher types, where input objects as well as output objects may be operations. The recursion schema for second order operations was introduced by Rózsa Péter [30].

Gödel System T [12], and Grzegorczyk System [16] are based on the recursion on higher types. Grzegorczyk's iterators (as primitive recursion schemata indexed by types) are considered as objects.

H. Curry [7] defined Grzegorczyk's iterators as terms in combinatory logic using pure iteration combinator corresponding to the operation *Iter* introduced in Section 4.1.

Girard [10] defined higher recursion schemata as terms in his System F.

The higher-order recursion is still of interest mainly because of its application in programming. However, recent works are based on formal approaches. For the Gödel-Herbrand style approach (see L. C. Paulson [29]), it is still not clear what is the meaning of equality for objects of higher types; this problem will be discussed in Section 9.

In M. Hofmann [19] and J. Adamek et al. [1] a category-theoretic semantics of higher-order recursion schemes is presented. In programming, see Ana Bove et al. [3], recursive algorithms are defined by equations in which the recursive calls do not guarantee a termination.

Finally, Carsten Schurmann, Joelle Despeyroux, and Frank Pfenning [33] propose an extension of the simply typed lambda-calculus with iteration and case constructs. Then, primitive recursive functions are expressible through a combination of these constructs. Actually, they did the same as the construction of Grzegorczyk's iterator presented below, however at the level of abstract syntax, that is, in the similar manner as Girard did earlier.

5.1 Grzegorczyk's iterator

Although the Grzegorczyk System was intended to be constructive, it is still a formal theory.

Grzegorczyk's iterator denoted here by R^A is a primitive (in Grzegorczyk System) object of type

$A \rightarrow ((N \rightarrow (A \rightarrow A)) \rightarrow (N \rightarrow A))$

that satisfies the following equations:

for any $a : A$, $\quad c : N \rightarrow (A \rightarrow A)$, and $k : N$

$R^A(a)(c)(1) = a$

$R^A(a)(c)(k+1) = c(k)(R^A(a)(c)(k))$

A notational convention is introduced for application () to simplify presentation, i.e. $f(a)(c)(k)$ is the same as $((f(a))(c))(k)$.

The problem is with the equality for objects of type A. In System F equality of two terms means their reduction to the same normal form. In a formal theory, the axioms of equality for all types must be added to the theory. In Section 9, relations corresponding to the equality on higher types are constructed.

By applying currying and uncurrying, R^A can be interpreted equivalently as operation of type

$(N \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow (N \rightarrow A))$

and then as operation $\bar{R}^A$ of type

$(N \rightarrow (A \rightarrow A)) \rightarrow (N \rightarrow (A \rightarrow A))$

Now, the definitional equations above can be rewritten as:

$\bar{R}^A(c)(1)(a) = a,$
$\bar{R}^A(c)(k+1)(a) = c(k)(\bar{R}^A(c)(k)(a))$
where $\bar{R}^A(c)(k)(a)$ is the same as $R^A(a)(c)(k)$.

In this new form the iterator is an operation that for input object (sequence) $c : N \rightarrow (A \rightarrow A)$ produces object (sequence) $\bar{c} : N \rightarrow (A \rightarrow A)$. First element of this sequence, i.e. $\bar{c}(1)$, it the identity operation on A, i.e. id_A. The element $\bar{c}(k+1)$ (i.e. $\bar{R}^A(c)(k+1)$) is the composition of $\bar{c}(k)$ (i.e. $(\bar{R}^A)(c)(k)$) and $c(k)$. In fact, $\bar{c}(k+1)$ is the composition of the first k elements of the sequence c.

Girard's recursion operator (indexed by type A) is defined as a term (denoted by R) in System F. The index is omitted. Definition of R is based on interpretation of natural numbers as operators for iterating operations. Applying number n to arbitrary operation (having the same type for input and output) means to compose n-times the operation with itself.
The recursion operator R is of type $A \rightarrow ((A \rightarrow (N \rightarrow A)) \rightarrow (N \rightarrow A))$, and has the following property.

For any $a : A$, $v : A \rightarrow (N \rightarrow A)$ and $k : N$,
$R(a)(v)(1) = a$
$R(a)(v)(k+1) = v(R(a)(v)(k))(k)$
The equalities above must be understood as term reduction to the same normal form.

Apply currying and uncurrying in the similar way as for the Grzegorczyk's iterator.
$(A \rightarrow (N \rightarrow A))$ and A can be swapped, so that
$(A \rightarrow (N \rightarrow A)) \rightarrow (A \rightarrow (N \rightarrow A))$
Then, in the first and the second segment, N and A can be swapped, so that
$(N \rightarrow (A \rightarrow A)) \rightarrow (N \rightarrow (A \rightarrow A))$
Hence, R may be rewritten equivalently as
$\bar{R}$ of type $(N \rightarrow (A \rightarrow A)) \rightarrow (N \rightarrow (A \rightarrow A))$,
such that
$\bar{R}(\bar{v})(0)(a) = a$, where $\bar{v} : N \rightarrow (A \rightarrow A)$ satisfies $\bar{v}(k)(a) = v(a)(k)$,

$\bar{R}(\bar{v})(k+1)(a) = \bar{v}(k)(R(\bar{v})(k)(a))$
In this form $\bar{R}$ is exactly the same as Grzegorczyk's iterator, i.e. for a sequence of operations of type $A \to A$ as input, it returns the output sequence where its $n+1$-th element is the composition of the first n elements of the input.

However, this cannot be taken literally that the input as a sequence is taken as whole (as actual infinity) by the Grzegorczyk's iterator (and Girard's operator) and returns a complete sequence as its output. From the syntactical point of view it is acceptable as a definition of a term, however not as a construction. Actually, the parameter $n : N$, that refers here to the n-th sequence element, must refer to construction parameter. It will be clear in the following construction.

5.2 Construction of Grzegorczyk's iterator

Let's come back to the notation describing constructions and let C denote the type $(N \to (A \to A))$.

We are going to construct operation $iterator : C \to C$, such that (informally) for any input object (sequence) $c : C$ returns object (sequence) $\bar{c} : C$, such that n-th element of $\bar{c}$ is the composition of the first n elements of sequence c. Although literally it is the same as for Grzegorczyk's iterator, it will be clear that at any construction step, *iterator* is a finite structure, that is, the parameter $n : N$ in the construction of *iterator* refers always to this finite structure, i.e. to the first n elements of c and of $\bar{c}$.

The detailed and explicit construction of this operation is simple, however a bit laborious. It is worth to carefully analyze this construction in order to grasp the full meaning of the constructability.

The construction needs two operations op and Rec_A constructed in Fig. 10.

The operation op takes number n and a sequence c as the input and returns new sequence $\underline{c}$ that differs from c only on the (n+1)-th element, that is, $\underline{c}(n+1)$ is the composition of two operations $c(n)$ and $c(n+1)$.

To explain the construction, let $n : N$ and $c : C$ be considered as input parameter (not as concrete objects), i.e. as pair (n, c) of type $N \times C$.

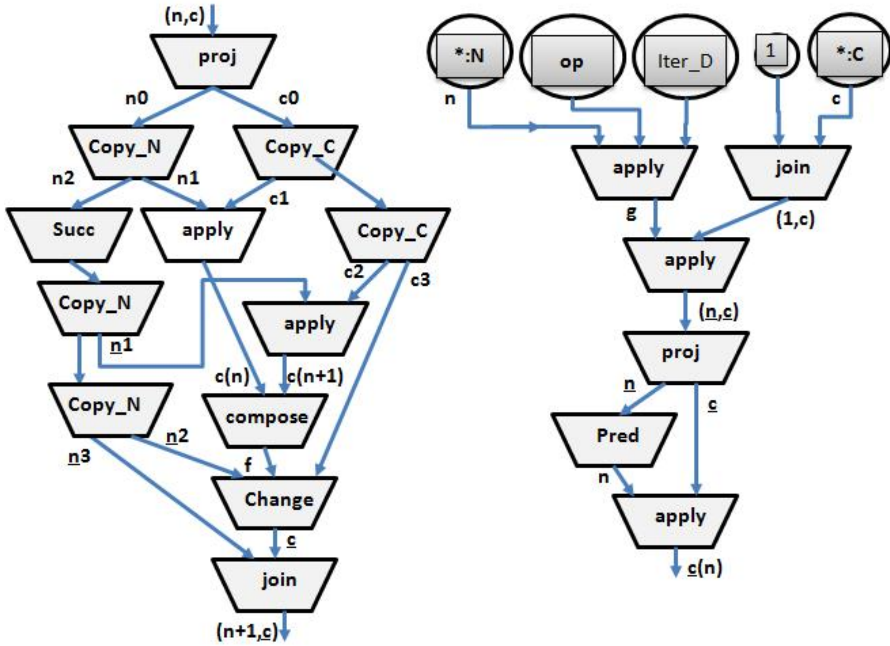

Figure 10: Construction of operations $op : N \times C \to N \times C$, and operation $Rec_A : (N; C) \to (A \to A)$

- Operation $proj_{N,C}$ applied to (n, c) returns two outputs: $proj^N_{N,C}(n, c)$ denoted by n^0, and $proj^C_{N,C}(n, c)$ denoted by c^0.
- $Copy_N$ applied to n^0 returns also two outputs: $Copy^1_N(n^0)$ denoted by n^1, and $Copy^2_N(n^0)$ denoted by n^2.
- $Copy_C$ applied to c^0 returns: $Copy^1_C(c^0)$ denoted by c^1, and $Copy^2_C(c^0)$ that is used again for coping.
- $Copy^1_C(Copy^2_C(c^0))$ is denoted by c^2, and $Copy^2_C(Copy^2_C(c^0))$ is denoted by c^3.
- c^1, c^2, c^3 are copies of c^0, and n^1, n^2 as copies of n^0, actually they are copies of c and n respectively.

- Apply $Succ$ to n^2, i.e. $Succ(n^2)$. Then copy it twice, that is, applying $Copy_N$ to $Succ(n^2)$ returns: $Copy^1_N(Succ(n^2))$ denoted by $\underline{n}^1$, and $Copy^1_N(Copy^2_N(\underline{n}^1))$ denoted by $\underline{n}^2$, and $Copy^2_N(Copy^2_N(\underline{n}^1))$ denoted by $\underline{n}^3$.
- $\underline{n}^1$, $\underline{n}^2$. and $\underline{n}^3$ are three copies of $Succ(n^2)$, that is, the same as $n+1$.
- Denote $apply_{((C;N)\to(A\to A)),(C;N)}(c^1;n^1)$ as $c^1_{n^1}$; it is the same as n-th element in the sequence c, i.e. $c(n)$.
- Denote $apply_{((C;N)\to(A\to A)),(C;N)}(c^2;\underline{n}^1)$ as $c^2_{\underline{n}^1}$; it is the same as $c(n+1)$.
- $compose_{A,A,A}(c^1_{n^1};c^2_{\underline{n}^1})$ is the same as composition of $c(n)$ and $c(n+1)$. Denote the composition by f.
- Change in the sequence c^3 the $\underline{n}^2$-th element to f, i.e. $Change_C(\underline{n}^2;f;c^3)$, and denote it by $\underline{c}$. In fact, the $(n+1)$-th element of c was changed to f.
- Join $\underline{n}^3$ and $\underline{c}$ into a pair, that is, $join_{N,C}(\underline{n}^3;\underline{c})$, and denote it by $(n+1,\underline{c})$
- Starting with (n,c) as the input in the construction we get $(n+1,\underline{c})$ as the output. Actually, the only change that was made in the original sequence c was to replace the $(n+1)$-th element of c by the composition of two operations $c(n)$ and $c(n+1)$.

The description of the construction of the operation $op : N\times C \to N\times C$, in Fig. 10, is completed.

Let $N\times C$ be denoted by D, then $op : D \to D$. Now, operation $Iter_D$ can be applied to op.

Note that $Iter_D(n)(op)(1,c)$ is the n-th iteration of operation op that for the input $(1,c)$ returns $(n+1,\underline{c})$ such that for any $k = 1,\ ...\ n+1$, the element $\underline{c}(k)$ is the composition of the first k elements of c. The elements $\underline{c}(m)$, for m geater than $n+1$, are the same as $c(m)$. Note that $n : N$ is the parameter of the construction. In order to construct Grzegorczyk's

iterator from op and $Iter$, the operation Rec_A, shown in Fig. 10, must be constructed first.

Let the type $(N;(D \to D)) \to (D \to D)$ be denoted by E. Operation Rec_A is constructed in the following way.

- Apply $Iter_D$ only to the operation op leaving the input $n : N$ open, i.e.
  $apply_{E,N;(D \to D)}(Iter_D; n; op)$ is operation of type $D \to D$. Let this operation be denoted by g. Note that the operations op has been already constructed in Fig. 10, whereas input n is a parameter.

- $join_{N,C}(1; c)$ is denoted by $(1, c)$. Note that c is a parameter.

- $apply_{(D \to D),D}(g; (1, c))$ is denoted by $(\underline{n}, \underline{c})$.
  Let $proj^N_{N,C}(\underline{n}, \underline{c})$ be denoted by $\underline{n}$, and $proj^C_{N,C}(\underline{n}, \underline{c})$ by $\underline{c}$.

- $\underline{n}$ is the same as $n + 1$, and for any $k = 1, 2, \ldots, n + 1$, $\underline{c}(k)$ is the composition of the k first operations in the original sequence c.

- $Pred(\underline{n})$ is the same as n. Finally, $apply_{C,N}(\underline{c}, Pred(\underline{n}))$ is operation of type $A \to A$. It is the composition of the n first elements (operations) of the original sequence c.

This completes a description of the construction of the operation $Rec_A : (N; C) \to (A \to A)$ in Fig. 10.

For the inputs n and c, the output, i.e. $Rec_A(n; c)$, is the composition of the first n elements (operations) from the input sequence c. This is the exact meaning of the construction of Rec_A.

However, applying currying, the operation Rec_A may be presented equivalently as the operation $\overline{Rec}_A$ of type $C \to (N \to (A \to A))$, i.e. of type $C \to C$. This may suggest that the operation $\overline{Rec}_A$ takes as its input a complete infinite sequence and returns as the output also a complete infinite sequence. It is not true. By the construction of Rec_A it is clear that $n : N$ is the parameter for this construction, i.e. for any $n : N$ the construction is a finite structure.

Operation $\overline{Rec}_A$ corresponds exactly to the Grzegorczyk's iterator. As an object, it can be used in more and more sophisticated constructions.

The conclusion is as follows. The level 0 can be viewed as a grounding (concrete semantics) for Grzegorczyk System as well as for Gödel System T. As to a grounding for Girard System F, higher levels of the Universe must be introduced. It seems that the Grzegorczyk's idea of *primitive recursive objects of all finite types* is fully explored on the level 0. However, the general recursive objects (in Gödel-Herbrand definition) have grounding (as constructions) on higher levels.

6 Continuum as a primitive type

In the XIX century and at the beginning of the XX century there was a common view among the mathematicians that Continuum is different than natural numbers and cannot be reduced to them, that is, Continuum cannot be identified with the set of the real numbers, or in general with a compact connected metric space. Real numbers are defined on the basis of rational numbers (for example, as Cauchy sequences), and rational numbers on the basis of natural numbers.

The following citations support this view.

- D. Hilbert [18]: *the geometric continuum is a concept in its own right and independent of number.*

- E. Borel [38]: *had to accept the continuum as a primitive concept, not reducible to an arithmetical theory of the continuum [numbers as points, continuum as a set of points].*

- L. Brouwer [38]: *The continuum as a whole was intuitively given to us; a construction of the continuum, an act which would create all its parts as individualized by the mathematical intuition is unthinkable and impossible. The mathematical intuition is not capable of creating other than countable quantities in an individualized way. [...] the natural numbers and the continuum as two aspects of a single intuition (the primeval intuition).*

In the mid-1950s there were some attempts to comprehend the intuitive notion of Continuum by giving it strictly computational and constructive sense, i.e. by considering computable real numbers and com-

putable functions on those numbers, see Grzegorczyk[13] [14], [15] and Lacombe [27]. These approaches were mainly logical and did not find a ubiquitous interest in Mathematics.

It seems that rather the homotopy theory describes the Continuum in a proper way. Recently, see HoTT [39], a type theory was introduced to homotopy theory in order to add computational and constructive aspects. However, the type theory is based on Martin Löf's type theory that still is a formal theory invented to provide intuitionist foundations for Mathematics. The authors of HoTT admit that there is still no computational grounding for HoTT.

HoTT is too serious enterprise and realized by so distinguished mathematicians to ignore it. The interpretation of the Continuum presented below was partially inspired by HoTT.

6.1 Informal introduction of Continuum

Intuitively continuum (as an object) can be divided finitely many times, so that the resulting parts are of the same type as the original continuum. Two adjacent parts can be united and the result is of the same type as the original continuum. Based on this simple intuition, continuum may be interpreted as an analog signal transmitted in a link. Here the link corresponds to the type Continuum whereas the signal corresponds to an object of the type Continuum. In telecommunication there are natural examples for link division, like frequency division and time division, as well as for link merging especially in the optical networks.

While dividing a link into two (or more) parts it may happen that in some parts there are no signals. These parts are deactivated (blinded) and are not taken to next divisions.

As a result of divisions and deactivations, a structure of active parts is emerged where each part is of the type Continuum, some parts are adjacent, and the structure as a whole is also of the type Continuum. This very structure may be interpreted as an approximation of the signal in the link, more accurate as the divisions are finer.

If interpreting Continuum as a physical link, there are many kinds of Continuum, of different dimensions, of different methods of division, as well as of many criteria for deactivation of blind parts of a link. However,

there are common features that constitute together the essence of the Continuum. This very essence is formalized below.

The concept of the Continuum is a result of the discussion held by the Author with Krzysztof Cetnarowicz (KI AGH) in July 2-4, 2014 in Kudowa Zdrój.

6.2 Cubical complexes and Continuum

Since the closed unit interval $[0, 1]$ (a subset of real numbers) is an example of continuum, let us follow this interpretation, however with some restrictions. Let the n-th dimensional continuum be interpreted as the unit n-th dimensional cube, that is, the Cartesian product of n-copies of the unit interval.

Let us fix the dimension and consider a unit cube. The cube may be divided into parts in many ways. However, the most uniform division is to divide it into 2^n the same parts, where n is the dimension of the cube. Each of the part may be divided again into 2^n the same sub-parts, and so on, finitely many times. Some of the parts may be removed as they are interpreted as empty (blind) in the link. The resulting structure is exactly a uniform cubical complex (see for example [22]) consisting of elementary cubes all of them of the same dimension. There is a natural relation of adjacency between the elementary cubes. Two cubes (parts) are adjacent if their intersection is a cube of dimension $n-1$. Since our intention is that the parts are not sets of points, the adjacency relation between the parts must be given as primitive.

Let a uniform cubical complex (complex, for short) be denoted by e. Two adjacent parts of e may be united into one part. After uniting some adjacent parts, the resulting structure is called a manifold by the analogy to manifolds in algebraic topology. See Fig. 11 for an example of 2-dimensional unit cube that is divided, and then some of the parts removed, and adjacent parts are united.

A complex may be arbitrary large, i.e. may contain a large number of parts, however, usually its essential structure (information it contains) is relatively small and does not depend directly on the number of the parts. For this very reason the uniting, preserving this essential structure and reducing significantly the number of parts, is of great importance.

Denote by $\hat{e}$ the manifold resulting from uniting the all adjacent parts in e.

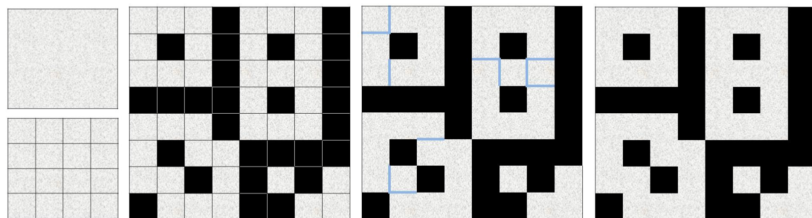

Figure 11: Divisions, deactivations, and uniting

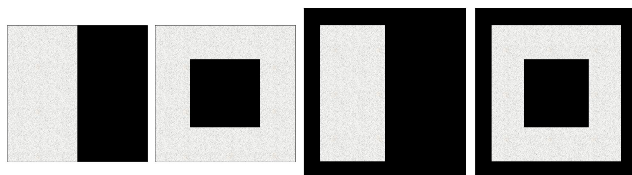

Figure 12: Examples of $\hat{e}$ and $\hat{e}^{-1}$ without black border and with black border

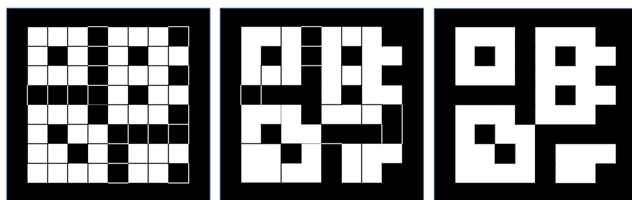

Figure 13: Objects with one black outer border

The final manifold $\hat{e}$ consists of disconnected components, see Fig. 11 the first example from the right.

For complex e, let its complement (consisting of empty black cubes (parts)) be denoted by e^{-1}. It is dual to e and it is also the subject to uniting its parts. The white final manifold $\hat{e}$, and its dual black final

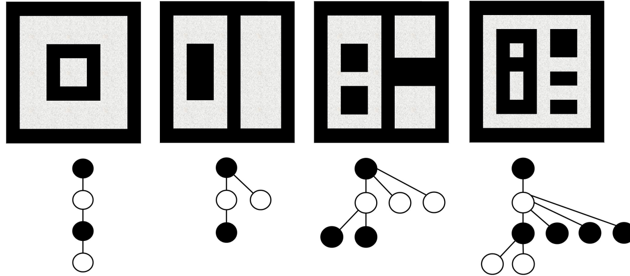

Figure 14: More final dual manifolds and the corresponding trees

manifold $\hat{e}^{-1}$ contain some essential information of the original object i.e. the complex e. The adjacency relation between the white components of $\hat{e}$ and the black components of $\hat{e}^{-1}$ reflects to some extent the structure of e and e^{-1}. However, this relation is not complete, that is, it can not distinguish between two different cases shown in Fig. 12, see the first two examples. Note that the adjacency between the white components (as well as black components) to the border of the original unit cube is important and can not be neglected. The adjacency to the border can be eliminated if the border is fixed as the one black component, see Fig. 12 and Fig. 13. This simplifies (by aggregation) the adjacency relation between black and white components. Denote this simplified aggregated relation by R_e.

From now on, let the original unit cube have the black border as the outer component.

The new primitive type *Continuum* is introduced. Object of this type is any uniform cubical complex c together with with its dual complex c^{-1} as the pair (c, c^{-1}), and with adjacency relations between elementary cubes, black and white ones. For short, let (c, c^{-1}) be denoted as c.

Note that the number of white components and the number of black components alone are not sufficient to express important information of an object of type *Continuum*, see Fig. 14 and the two first examples where the number of white components and the number of black components are the same.

The adjacency relation R_c contains more essential information of the object (c, c^{-1}).

Equivalence relation for objects of type $Continuum$. Any two objects c_1 and c_2 are *similar* if the relations R_{c_1} and R_{c_2} are isomorphic.

More interesting relations on $Continuum$ may be introduced by using homomorphisms between relations R_{c_1} and R_{c_2}.

Any black component (except the one that contains the border) it is externally adjacent exactly to one white component. Also any white component is externally adjacent exactly to one black component. Hence, R_e can be represented as the following tree. The `root` of the tree is the black segment containing the border. The `node`s of depth 1 (children of the root) are the white segments adjacent (according to R_e) to the black root. Each of the white `node` of depth 1 has inner adjacent black segments as its children (`node`s of depth 2). Each of the black `node`s of depth 2 has inner adjacent white segments as its children (`node`s of depth 3). And so on. See Fig. 14 for examples for complexes of dimension 2. However, for higher dimensions the relation R_e is not sufficient to distinguish different manifolds, like tori and sphere in dimension 3. i.e they have the same simple tree ($black \rightarrow white \rightarrow black$) corresponding to their adjacency relations between black and white segments. Hence, the aggregated adjacency relation R_c reflects only partial information contained in objects of type $Continuum$.

It is clear that there must be different and more subtle uniting criteria that reduce the number of parts and adjacency relation for white complex as well as for its dual black complex. Final result of such uniting should represent important information contained in objects of type $Continuum$. The problem is to find out the criterion that gives the all essential information. The term *essential* should correspond to homotopy theory.

6.3 Some remarks on the Continuum

The Continuum as a primitive type is a link whereas an object of Continuum is an analog signal transmitted in the link. This signal may be

approximated and recognized (classified). It is done by dividing the link into parts and deactivating those parts that are blind (there is no signal in those parts). The result may be represented (digitalized) as a well known finite combinatorial structure, that is, a uniform cubical complex and its dual complex. These very complexes may be subjects for computations consisting in uniting some adjacent parts and resulting in a classification of the complexes (actually, of the signal in question).

The investigations presented above still need to be verified, especially how is this approach related to the homotopy theory.

Cubical complexes are used in computational homology (see for example [22]) where the cubes are considered as sets of points. However, computations and construction on the type *Continuum* as a primitive type seems to be novel in Informatics.

Equivalence relation on the type Continuum is important because it may be considered as the equality; this will be discussed in Section 9. Equality is the key notion for Martin Löf TT and HoTT that still does not have computational grounding. It is an important and missing aspect of HoTT as stated by Robert Harper [17] one of the creators of HoTT.

In homotopy type theory each type can be seen to have the structure of an weak ∞-groupoid. There is a correspondence between the ∞-groupoids and topological spaces. Simplicial sets, cubical sets, and cubical complexes are between them. Recent works [2] in HoTT may indicate that the interpretation of the objects of Continuum as cubical complexes makes sense.

7 Summary of the level zero

The primitive types, constructors, and operations described above constitute the level zero of the Universe. What can be constructed on this level? Since the primitive recursion schema corresponds to operations, it is clear that all primitive recursive objects of all finite types can be constructed as it is in System T and Grzegorczyk System. Since types are used as terms in System F, it describes more than the constructions at level 0.

It is not clear what the Continuum contributes to constructions. It is a hard problem, and concerns a computational grounding of HoTT as well as of a large part of Mathematics.

The interpretation of types, their constructors, and operations as links, signals, and socket boards is important. This gives rise to comprehend the key notion of object and its construction as a parameterized finite structure. This interpretation is in opposition to formal theories, where object is described as a term, construction amounts to substitution and lambda abstraction whereas computation to beta reduction.

Level 0 is the basis for building the higher levels of the Universe.

8 Level 1

Passing to the level 1 consists in apprehending the level 0 as a generic construction method where types can be considered as objects, type constructors as operations and particular primitive operations (indexed by types, for example *compose*) as a polymorphic operation that takes types as its input and returns a particular primitive operation.

Since types from level 0 are to be considered as objects, a super type for these objects must be constructed. Denote this super type by $Types^0$; the superscript 0 suggests that there will be also super super type at level 1, and generally $Types^n$.

Level 1 is an extension of level 0, so that, all types, constructors, operations and constructions from level 0 are also on level 1. Generally, level n+1 is an extension of level n.

At level 1 and higher, new type constructor are introduced, i.e. dependent types and polymorphic operations as it is in ML TT, System F, and CoIC. Actually, dependent types emerge from the level 0 in a natural way as the necessary consequence of the generic construction method of level 0.

What are types as objects? At level 0, they are interpreted as socket boards constructed by product ($\times$), disjoin union ($+$), and arrow ($\rightarrow$ by separating input types and output types). $Types^0$ (as the super type of all types at level 0) can be constructed inductively.

At level 1, the following three type constructors at level 0, i.e. $\times, +, \rightarrow$,

are in fact operations that given two types as input objects return another type as the output object. As operations they are:

$\times^1 : (Types^0; Types^0) \rightarrow_1 Types^0$
$+^1 : (Types^0; Types^0) \rightarrow_1 Types^0$
$\rightarrow^1 : (Types^0; Types^0) \rightarrow_1 Types^0$

Note that the type constructor $\rightarrow_1$ above is the extension of the type constructor $\rightarrow$. Analogously, $\times_1$ is the extension of $\times$, and $+_1$ is the extension of $+$. The general convention is that subscripts denote extensions whereas superscripts denote levels of operations and types. The subscripts and superscripts are omitted if there is no confusion.

There is one common destructor for the operations $\times$, $+$ and $\rightarrow$. It is the operation $des : Types^0 \rightarrow (Types^0; Types^0)$ such that for a primitive type as the input it returns two the same primitive types as the output; for a composite type it returns the components either of the product, or disjoint union, or arrow. Although this destructor is not used in the constructions presented in this work, it is introduced for the completeness of the level 1.

$Types^0$ is an inductive type, that is, an operation of type $N \rightarrow Types^0$ can be constructed at level 1 (actually in many ways) that enumerates all types from level 0. Let us fix one such operation and denote it by $Ind^1 : N \rightarrow Types^0$.

Note that types from level 0 can still be used as types at level 1, see Ind^1 above.

For all primitive operations at level 0, its extensions to level 1 is marked in their type indexes; for example $join_{A;B}$ is extension to level 1 if at least one of the type A and B is a type at level 1.

Operation *Copy* is (by its nature) amorphous and extensible for all construction of the Universe, i.e. once an object was constructed, its construction can be repeated.

Primitive operations from level 0 are indexed by types of level 0. At the level 1 these operations are outputs of higher order operation for which these types are input objects. For example, $plus^1$ is a polymorphic operation at level 1. Its input is of type $(Types^0; Types^0)$ whereas the type of output object depends (is determined) by the input object. For input $(A; B)$ the operation $plus^1$ returns operation $plus_{A,B} : (A; B) \rightarrow$

$(A + B)$ as the output.

The same is for all primitive operations at level 0, i.e. $join, proj, plus, get, compose, apply, const, Copy, Iter, Change, currying, uncurrying$.

Their higher order versions (as operations at level 1) are denoted by adding superscript 1, i.e. $join^1, proj^1, plus^1, get^1, compose^1, apply^1, const^1, Copy^1, Iter^1, Change^1$, $currying^1$, $uncurrying^1$.

All these operations are polymorphic, i.e. input object determines the type of output object. For any of the operations above, this very determination is an operation of type $A \to Types^0$ where A is, for example, $(Types^0; Types^0)$ for $join^1$ and $plus^1$.

The types of these polymorphic operations are the well known dependent types ([11] and [28]), with constructor Π as a generalization of product and arrow, and constructor Σ as a generalization of disjoint union.

For operation $F : A \to Types^0$, an object of the type ΠF is a polymorphic operation g that for input object $a : A$ it returns output object $g(a) : F(a)$. If F is constant, i.e. its output is B independent of the input, then ΠF is reduced to the type $A \to B$.

Objects of type ΣF are of the form $(a; b)$ where $a : A$ and $b : F(a)$.

Operation F may have multiple inputs, for example, $F : (Types^0; Types^0) \to Types^0$, and the output $F(A; B)$ is the type $(A; B) \to (A + B)$. Note that $plus^1 : \Pi F$, and $plus^1(A; B)$ is the same as $plus_{A,B}$.

According to the principle, on which the Universe is being build, dependent types and polymorphic operations must be parameterized by $n : N$. In fact, this parameter is implicit in the construction of $Types^0$ as inductive type by operation Ind^1. At the level 2, $Types^0$ will be considered also as an object of $Types^1$.

Constructor for objects of type ΣF is as follows. For any operation $F : A \to Types$ and operation $f : \Pi F$, (i.e. for all $a : A$, $f(a) : F(a)$), the constructor σ_F is of type $\Pi F \to (A \to \Sigma F)$, such that $\sigma_F(f)$ is operation such that for any input $a : A$, it returns output $(a; f(a) : F(a))$ of type ΣF.

The dependent type constructors are interpreted as logical quantifiers in Section 9.

Dependent type constructors are at level 1. They are extended (analogously as the product, disjoin union and arrow) to all higher levels of the Universe.

Dependent type constructors are interpreted as operations at level 2 and higher. Π^1_A and Σ^1_A are operations (at level 2) of type $(A \rightarrow Types^0) \rightarrow Types^1$, whereas Π^2_A and Σ^2_A are (at level 3) of type $(A \rightarrow Types^1) \rightarrow Types^2$. These operations will be used in constructions.

The super types $Type^1$ and generally $Type^n$ were introduced implicitly. However, it is clear that $Types^1$ is the type of all types at level 1 that are considered at level 2 also as objects. Analogously for $Types^n$. This cumulative type hierarchy is inductive. Also each $Types^{n+1}$ is an inductive type, i.e. operation Ind^n can be constructed that enumerates all objects of this type.

9 Relations

The concept of relation is not easy to comprehend. Usually relation is defined as is a collection of ordered pairs of objects. Actually, this set theoretical definition is not sufficient. Relation is (like operation) a primitive notion. It seems that it corresponds to a primeval generic method of comparing two objects.

For any primitive type there is at least one elementary binary relation between objects of this type. For the type Continuum it is the similarity relation determined by the method for comparing two objects and stating that they are either similar or not. For natural numbers the relation is simpler. Elementary relations are the basis to construct more sophisticated relations.

So far the types were inhabited (not empty), i.e. primitive types are inhabited and type constructors produce inhabited types from inhabited types. Introducing relations enforces some types to be uninhabited (empty).

9.1 Elementary relations for natural numbers

There are the following three interrelated elementary relations on N: $Equal_N$, $Lesser_N$, and $Greater_N$ constructed below.

There are two links of type N, one for n, and one for k. It is supposed that for each of the links the state of the link can be recognized as either empty or not empty. This may be considered as the most primitive relation (property) for natural numbers.

Put the signals $n : N$ and $k : N'$ into the two links. i.e. N and N'.
Procedure N. Check the two links. If each of them is empty, then this is the witness (object) for the type $Equal_N(n; k)$ to be inhabited. If the link N is not empty and the link N' is empty, then it is the witness for type $Greater_N(n; k)$ to be inhabited. If the link N is empty and the link N' is not empty, then it is the witness for the type $Lesser_N(n; k)$ to be inhabited. If both links are not empty, then apply to each of them $Pred$, that is, $Pred(n)$, and $Pred(k)$ (it means to remove from each of the links one elementary signal) and go to the beginning of the procedure.

For any $n : N$ and $k : N$, $Equal_N(n; k)$ (n is equal to k) is a primitive type (at level 1) corresponding to the states of the two links. Analogously for $Lesser_N(n; k)$ (n is lesser than k), and for $Greater_N(n; k)$ (n is greater than k).

If a type is inhabited, it corresponds to *truth*, whereas the opposite corresponds to *false*.

These three elementary relations are operations of type $(N; N') \rightarrow Types^1$. So that for any $n : N$ and $k : N'$, $Equal_N(n; k)$, $Lesser_N(n; k)$, and $Greater_N(n; k)$ are types at level 1, called *parameterized primitive relational types*. If a primitive relational type is inhabited, then a primitive object of this type must be introduced. This will be done in Section 9.4 for relations, not for any particular type separately.

The elementary relations can be used in constructions of more sophisticated well known relations like, *two numbers are relatively prime*, or *n multiplied by k is m*.

Generally, an elementary relation R is an operations (at level 2) of type $(A_1; ...; A_k) \rightarrow Types^1$ such that for any $a_1 : A_1$, ... , $a_k : A_k$,

$R(a_1; ...; a_n)$ is a primitive relational type at level 1.

Relation is a primitive notion identified with parameterized primitive types. The primitive types correspond to evaluation of the relation for parameters (input), i.e. checking if the relation is true for these parameters. Parameterized primitive types are at level 1, and treated as objects at level 2.

Dependent type constructors Π and Σ may be applied to relations. They correspond to the quantifiers: for all, and exists.

In CoIC [4] there is a separate class *Prop* for types that correspond to parameterized primitive types. It seems that there is no reason to separate relations from the rest of operations. However, parameterized primitive relational types are used only for elementary relations.

9.2 Conjunction, disjunction and negation

Product of two types corresponds to conjunction, whereas disjoin union to disjunction. For two relation $R_1 : A \rightarrow Types^1$, and $R_2 : B \rightarrow Types^1$ and fixed objects a, b and two primitive types $R_1(a)$, and $R_2(b)$,

- conjunction is $R_1(a) \times R_2(b)$,
- disjunction is $R_1(a) + R_2(b)$.

Negation as type constructor (denoted by $\neg$) is not easy to comprehend. If type A is inhabited (this corresponds to *true*), then $\neg A$ should be empty and correspond to *false*. However, if A is empty, then how to interpret that $\neg A$ is inhabited? To state that $\neg A$ is *true* means to construct an object of type $\neg A$. What are objects of type $\neg A$? What can be inferred from an empty type? Is there only one empty type? Negation applied to a separate type has no sense. This is clearly seen in the following example. What is $\neg Equal_N(1; 2)$? It has sense (grounding) only if the complements $Greater_N$ are $Lesser_N$ are also considered. Hence, the negation can be grounded only if the complete relation is considered. i.e. $Equal_N$, $Greater_N$ and $Lesser_N$ together are complementary parts of the complete relation. Negation of one part is grounded in its complement.

The conclusion is that the inherent part of a relation construction is the construction of its complement. This very complement may be considered as its negation.

The next problem concerns interpretation of the type constructor $\rightarrow$ as logical implication. Consider $A \rightarrow B$ and cases where A and/or B are empty. Interpreted as implication means that it is empty only if A is inhabited and B is empty. That is, if A and B are empty, then the type $A \rightarrow B$ is inhabited! What are operations that have input and output empty? Logically it is *true*. Such operations have no sense, that is, there are no such operations at all. Hence, the constructor $\rightarrow$ cannot be interpreted as logical implication $\implies$, that is, $(A \implies B)$ is defined as the disjunction $(\neg A + B)$.

Logically, the constructor $\rightarrow$ means that if the type $A \rightarrow B$ is inhabited (operation g can be constructed) then for any proof (object) $a : A$, operation g returns output $g(a)$ as a proof (object) of type B. Hence, arrow corresponds to the deriving rules in formal logic.

9.3 Complex relations

Once the elementary complete relations for primitive types are given, the next problem is how to construct complex relations and types (as propositions).

For natural numbers the complete elementary relation consists of three parts: $Equal_N$, $Greater_N$ and $Lesser_N$.

Relations of the first order are operations of type $(A_1; ...; A_k) \rightarrow Types^1$. Higher order relations, i.e. of types $(A_1; ...; A_k) \rightarrow Types^n$, will arise as multiple quantifiers are used in constructions. The superscript n is omitted.

For relation $R_1 : A \rightarrow Types$ and $R_2 : B \rightarrow Types$, the following constructions

$compose(R_1; R_2; \times)$
$compose(R_1; R_2; +)$
$compose(R_1; R_2; \rightarrow)$

correspond to conjunction, disjunction and arrow of two relations R_1 and R_2. The superscripts for $+$, $\times$, $\rightarrow$ are omitted as well as the indexes of *compose*.

Informally these operations will be denoted by $[R_1 \times R_2]$, $[R_1 + R_2]$ and $[R_1 \to R_2]$; they are of type $(A; B) \to Types$. $[R_1 \times R_2](a; b)$ denotes the same as $R_1(a) \times R_2(b)$. Analogously for $+$ and $\to$.

If A is the same as B, then the relation $\bar{R} : A \to Types$, such that $\bar{R}(a)$ is the same as $R_1(a) \times R_2(a)$ is constructed as the composition of $Copy_A$ with $[R_1 \times R_2]$, i.e. $compose(Copy_A; [R_1 \times R_2])$.

Negation of relation R (i.e. $\neg R$) is inherently built into the construction of the relation R. It is convenient to consider the negation $\neg$ as an operation of type $Types \to Types$, however, it can be applied only to parameterized primitive types (complete elementary relations) and complex relations constructed from these elementary relations. Application to other types has no grounding and no sense. Construction of the extension of the negation to complex relations will be explained in the next Section.

For any relation R of type $A \to Types$, there are dependent types ΠR and ΣR. The first one corresponds to logic formula (proposition) $\forall_{a:A} R(a)$, whereas the second one to logic formula (proposition) $\exists_{a:A} R(a)$. Notice that we abuse somewhat the notation because R denotes object whereas in formal logic it is just a symbol without interpretation. In Section 11 the distinction between objects and symbols will be clarified as well as an explicit grounding (interpretation) of terms and formulas of a formal theory will be given.

If a relation has multiple arguments (inputs), say $R : (A; B) \to Types^n$, then ΠR corresponds to $\forall_{a:A} \forall_{b:B} R(a; b)$, the same for Σ. However, formula $\forall_{a:A} \exists_{b:B} R(a; b)$ corresponds to the type that is constructed as follows. Relation R must be transformed by currying to the equivalent operation $\bar{R} : A \to (B \to Types^n)$. In order to apply a dependent type constructor, its version as operation is needed, i.e. Π^{n+1} or Σ^{n+1} of type $(B \to Types^n) \to Types^{n+1}$. For example, logic formula $\exists_{b:B} R(a, b)$ (with free variable $a : A$) corresponds to the composition $compose(\bar{R}; \Sigma^{n+1})$; it is one input relation of type $A \to Types^{n+1}$. Then, $\Pi compose(\bar{R}; \Sigma^{n+1})$ corresponds to the logic formula (proposition) $\forall_{a:A} \exists_{b:B} R(a, b)$.

For relation $R : A \to Types$, operation $Copy$ applied to R, i.e. $Copy(R)$

returns two outputs $Copy^1(R)$ and $Copy^2(R)$.

$compose(Copy_A; [Copy^1(R)+\neg Copy^2(R)])$ (denoted by $LEM_R : A \to Types$) corresponds to the law of excluded middle. For any input object $a : A$ the resulting output type $LEM_R(a)$ is $R(a) + \neg R'(a')$. Recall that by the notational convention a' is a copy of a, and R' is a copy of R; the same is for types. In Section 9.4, it will be clear that if the negation operator $\neg$ is extended for already constructed relation R, then either $R(a)$ is inhabited or $\neg R'(a')$ is inhabited. Generally the type ΠLEM_R is inhabited, that is, object of this type can be constructed on the basis of the construction of R.

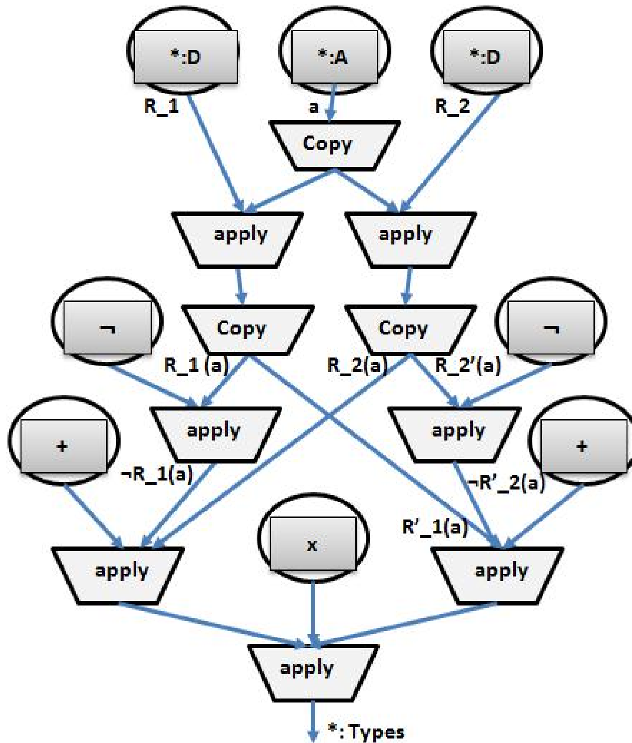

Figure 15: The construction of operation $Equiv_D$ modulo currying

Let us construct the relation that corresponds to logical equivalence

of two relations $R_1 : A \to Types$ and $R_2 : A' \to Types$, i.e. informally for any $a : A$, $(\neg R_1(a)+R_2(a'))\times(\neg R_2'(a'')+R_1'(a'''))$. Let the type $A \to Types$ be denoted by D. See Fig. 15 for the construction of operation that is of type $(D; A; D') \to Types$. Applying currying we get the required $Equiv_D$ of type $(D; D') \to D$ such that for any R_2, R_2 and any $a : A$, the type $Equiv_D(R_1; R_2)(a)$ corresponds to the above logical equivalence.

For already constructed relation R, the double negation $\neg\neg R$ is logically equivalent to R. That is, the type $Equiv_D(\neg\neg Copy^1(R); Copy^2(R))$ is inhabited, so that object of this type can be constructed on the basis of the construction of R.

Negation of quantified relations. Let $R : A \to Types$ be a relation. Logical formulas $\neg\forall_{a:A}R(a)$ and $\exists_{a:A}\neg\ R(a)$ are logically equivalent, i.e. $\neg\forall_{a:A}R(a) \implies \exists_{a:A}\neg\ R(a)$ and $\exists_{a:A}\neg\ R(a) \implies \neg\forall_{a:A}R(a)$. Also the logical formulas $\neg\exists_{a:A}F(a)$ and $\forall_{a:A}\neg\ F(a)$ are logically equivalent. For already constructed relation R, these equivalences correspond to the following inhabited types:

$Equiv_D(\neg\Pi Copy^1(R); \Sigma\neg Copy^2(R))$

and

$Equiv_D(\neg\Sigma Copy^1(R); \Pi\neg Copy^2(R))$.

Note that the relations as well as equivalences between relations are considered as operations, that is, as objects of the Universe that are parametrized finite structures.

The types constructed on the basis of parametrized primitive relational types correspond to propositions (atomic formulas) of a formal language. This correspondence may be viewed as a grounding (meaning) for these propositions. Some of these types correspond to axioms, i.e. there are primitive objects of the types associated with the constructions of complete elementary relations.

9.4 Relations on natural numbers

In this section the relations, types, and the primitive objects describe the Procedure N, see Section 9.1. They are basis for more sophisticated

constructions. It resembles axiomatic (however, not formal) approach in Mathematics where the types and the primitive objects correspond to axioms. From this point of view it is interesting to give minimal collection of such relations, types and objects that describe the Procedure N completely.

Consider two relations F_1 and F_2 of type $(N; N') \to Types$. Let us change notation and now let $[F_1 + F_2]$ denote the following construction $compose(Copy_N; Copy_{N'}; compose(F_1; F_2; +))$, i.e. $[F_1 + F_2](n, k)$ is the same as $F_1(n, k) + F_2(n, k)$. Analogously for $\times$. The basic inhabited types

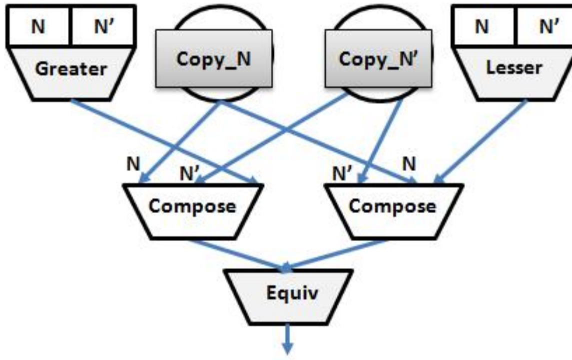

Figure 16: The equivalence of $Greater(n; k)$ and $Lesser(k; n)$

(axioms) for the complete elementary relations $Equal_N$, $Greater_N$ and $Lesser_N$ are as follows.

- For all $n : N$ and $k : N'$: $[[Equal_N + Greater_N] + Lesser_N](n; k)$. The type $\Pi[[Equal_N + Greater_N] + Lesser_N]$ states that the relations together are complete.
- The types $\Pi\neg[Equal_N \times Greater_N]$ and $\Pi\neg[Equal_N \times Lesser_N]$ and $\Pi\neg[Lesser_N \times Greater_N]$ state that the relations are disjoint.
- $compose(Copy_N; Equal_N)$ is of type $N \to Types^1$, it is the same as $n : N$ and $Equal_N(n; n)$. The type $\Pi compose(Copy_N; Equal_N)$ corresponds to the reflexivity of relation $Equal_N$.

- It follows from the Procedure N that *(n : N and k : N′ and Greater(n; k))* is the same as *(k : N and n : N′ and Lesser(k; n))*, i.e. they are equivalent. Both relations are of type $(N; N') \to Types$ where N refers to the first link, whereas N' to the second link. The corresponding operation is constructed in Fig. 16. Note that in the construction, in the second composition from the left, the inputs are swapped.

- Primitive objects of the types above are to be introduced. For these complete elementary relations as well as for relations constructed from them, the types for double negation law and the type for LEM as propositions follow from these axioms. It means that objects of these types can be constructed from the primitive objects. Also for logical equivalence for negation of quantified formulas, the objects of the corresponding types can be constructed from the primitive objects.

Note, that this approach may be generalized to any complete elementary relations.

$Equal_N$ is an equivalence relation, so that it satisfies reflexivity, symmetry and transitivity. The relations $Lesser_N$ and $Greater_N$ satisfy antisymmetry, transitivity, and totality. All these conditions can be constructed as inhabited types, i.e. for each of these type, the primitive object (interpreted as an axiom) should be given.

There are also inhabited types related to $Succ$ and $Pred$:

- For any $n : N$, $Equal_N(n; Pred(Succ(n)))$. The corresponding type is
  $\Pi compose(Copy^2_N; compose(compose(Copy^1_N; Succ); Pred); Equal_N)$.

- For all $n : N$ and $k : N$: $Equal_N(n; k)$ and $Equal_N(Succ(n); Succ(k))$ are equivalent. Let $compose(Succ; Succ'; Equal_N)$ be denoted by E^{Succ}.
  The corresponding type is $\Pi Equiv_D(Equal_N; E^{Succ})$, where D is $(N; N') \to Types$.

- For all $n : N$ and $k : N$,
  $Greater_N(n; k)$ and $Greater_N(Succ(n); Succ(k))$ are equivalent.
- For all $n : N$ and $k : N$, $Greater_N(Succ(n); Succ(k))$ and $Greater_N(Pred(Succ(n)); Pred(Succ(k)))$ are equivalent.
- For any $n : N$, $Greater_N(Succ(n); n)$,
  and for any $n : N$, $Greater_N(Succ(n); Pred(Succ(n)))$.
  The corresponding types can be easily constructed.
- Relation $Equal_N$ is the equality on N, so that, the types corresponding to substitution of n for m in $Greater_N$ and $Lesser_N$ if $Equal_N(n; m)$ should be inhabited.

For each of the above types a primitive object should be introduced. This corresponds to axioms in arithmetic. It should be stressed once again that these axioms describe the Procedure N. The primitive objects of the types (corresponding to the axioms) are used in more sophisticated constructions.

9.5 More inhabited types for natural numbers

A detailed construction of an inhabited type (corresponding to a theorem in a formal theory), and an object of this type (a proof) is presented below.

Informally (as a proposition) the inhabited type in question can be described as: for all $k : N'$ there exists $n : N$, such that $Greater_N(n; k)$. Relation $Greater_N$ is of type $(N; N') \rightarrow Types^1$.

Apply currying (see Section 4.3); the result is operation of type $N' \rightarrow (N \rightarrow Types^1)$ denoted by $Greater^c_N$.
$\Sigma^2 : (N \rightarrow Types^1) \rightarrow Types^2$ can be composed with with $Greater^c_N$, i.e. $compose(Greater^c_N; \Sigma^2)$, is denoted by G, and is of type $N \rightarrow Types^2$. The relation G corresponds to the logic formula $\exists_{n:N}\ (n > k)$.

The type ΠG is inhabited and corresponds to the logical proposition (theorem) $\forall_{k:N} \exists_{n:N}\ (n > k)$.

To prove this theorem, an object of type ΠG must be constructed. Intuitively the successor operation $Succ$ should be of this type. However,

it is of type $N \to N$ but the object must be of type ΠG, i.e. it is a polymorphic operation (denote it by *op*) such that for any $k : N$, $op(k)$ is an object of type $G(k)$, i.e. $op(k)$ is of the form $(n; a)$ and $a : Greater(n; k)$.

It is enough to substitute $Succ(k)$ for n. However, it is a syntactic operation.

The appropriate construction is shown in Fig. 17. For input $k : N$,

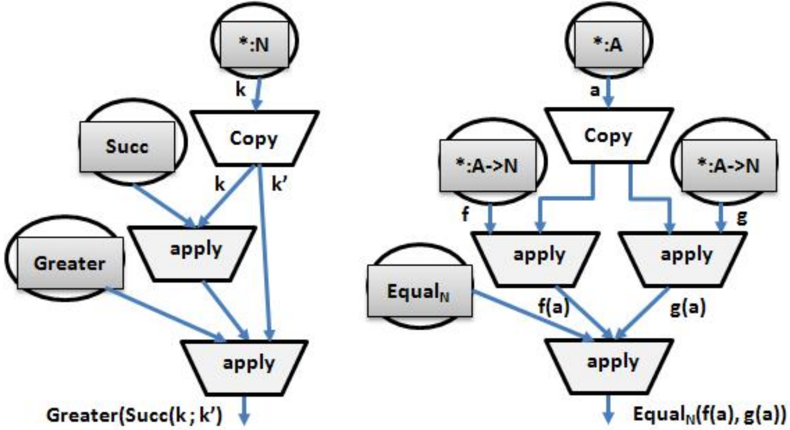

Figure 17: The construction of operation F, and operation $\bar{H}$

$Copy_N$ returns two outputs. The first output $Copy^1(k)$ is applied to $Succ$, i.e. $apply(Succ; Copy^1(k))$. Then, the result and $Copy^2(k)$ is applied to $Greater_N$. The result is $Greater(Succ(k); k)$. Denote this construction by F; it is an operations of type $N \to Types^1$.

The type ΠF is inhabited (it is an axiom, see the previous Section 9.4), and there is a primitive object of this type; denote it by f. For any $k : N$, $f(k) : F(k)$.

Now we are going to use the constructor σ_F; see the end of Section 8. Apply σ_F to f, i.e. $\sigma_N(f)$ is such that for any $k : N$, $\sigma_F(f)(k)$ is the same as $(k; f(k))$ and $f(k) : Greater(Succ(k); k)$.
Hence, $\sigma_F(f)$ is the required operations *op* of type ΠG.

9.6 Relations on higher types

Two operations of the same type (say f and g of type $A \to N$) are *equal* if for any input $a : A$, $Equal_N(f(a), g(a))$. This is an extensional notion of equality for functionals. Appropriate relation $Eq_A(f, g)$ as operation $Eq_A : ((A \to N); (A \to N)) \to Types$ is constructed below.

First, for two fixed f and g let us construct operation H of type $((A \to N); (A \to N)) \to (A \to Types^1)$, such that for any $a : A$, $H(f; g)(a)$ is the same as $Equal_N(f(a), g(a))$.

- The construction of a bit different (modulo currying) operation $\bar{H}$ is in Fig. 17. It is of type $(A; (A \to N); (A \to N)) \to Types^1$ such that $\bar{H}(a; f; g)$ is the same as $Equal_N(f(a), g(a))$.
- By applying currying twice to $\bar{H}$, the operation $H : ((A \to N); (A \to N)) \to (A \to Types^1)$ is constructed.
- In order to construct relation $Eq_A(f, g)$ operation Π^2 of type $(A \to Types^1) \to Types^2$ is needed.
- The composition $compose(H, \Pi^2)$ is the construction of the required relation $Eq_A : ((A \to N); (A \to N)) \to Types^2$.

Eq_A is an equivalence relation so that appropriate inhabited types corresponding to symmetry, reflectivity and transitivity can be constructed as well as objects of these types.

This relation may be the subject for dependent type constructors. For example, for any f there is g such that $Eq_A(f; g)$ (evidently a true proposition), or there exists f, that for any g such that $Eq_A(f; g)$ (evidently a false proposition). The construction of the first type is as follows.

- By applying currying to Eq_A we get $Eq^c_A : (A \to N) \to ((A \to N) \to Types^2)$
- The operation Σ^3 of type $((A \to N) \to Types^2) \to Types^3$ is needed.
- Compose Eq^c_A with Σ^3, i.e. $compose(Eq^c_A; \Sigma^3)$; it is of type $(A \to N) \to Types^3$; denote this operations by Eq^e_A.

Π can be applied to Eq^e_A, i.e. ΠEq^e_A is the type corresponding to the logic proposition
$\forall_{g:A\to N}\exists_{f:A\to N}\ (g = f)$.

- ΠEq^e_A is inhabited, so that an object of this type can be constructed. Intuitively, it should be the identity operations $id_{(A\to N)}$. The construction of the required object is analogous to the construction in the previous section, and is reduced to the reflectivity of relation Eq_A.

A construction of the type (denoted by B) corresponding to the proposition $\exists_{f:A\to N}\forall_{g:A\to N}\ (g = f)$ is analogous. B is empty, i.e. not inhabited. Hence, the type $\neg B$ is inhabited. So that an object of type $\neg B$ should be constructed. $\neg B$ corresponds to the proposition
$\forall_{f:A\to N}\exists_{g:A\to N}\ (g \neq f)$.

Intuitively, it is clear that an object of type $\neg B$ is an operation that for any f, it returns $compose(f, Succ)$ as the output. However, to be precise, the type $\neg B$ must be constructed first, and then, the object of this type corresponding to this operation can be constructed.

To conclude the relations on natural numbers and functionals, note that the grounding for them consists of the complete elementary relations $Equal_N$, $Lesser_N$, and $Greater_N$ along with primitive objects that correspond to axioms. Actually this grounding is the complete description of the Procedure N.

9.7 Summary of the relations

Since the equality can be constructed for functionals (i.e. operations of type $A \to N$), it can be extended to any higher order operations of type $B \to (A \to N)$, and then to any type.

In this context operation *Copy* is important. Given an object as input, the output consists of two object, the first is the original object, whereas the second one is a copy of the original. It is natural to say that these two objects are equal. There are also different constructions that are essentially the same, see Fig. 18 as an example.

There is also equality between objects of different types as a result of *currying* and *uncurrying*. Operation *Equiv*, see Fig. 15 may serve

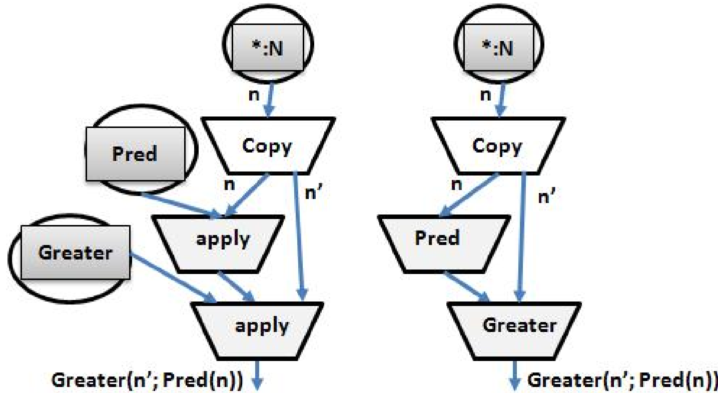

Figure 18: Different equivalent constructions

as equality for relations. Generally, what was presented above does not capture the complete intuitive meaning of the concept of equality.

The question is if *"object a is of type A"* may be considered as a relation. Generally, the type of an object is determined by its construction. In Section 11, high order relation for types as objects is considered, i.e. informally it is *"type A is inhabited"*.

Something similar to lambda abstraction can be performed on constructions by removing some objects that are, in fact, inputs. See Fig. 18 and objects *Pred*, *Greater*. After removing a concrete input object (say, of type A) from the construction, its place becomes an input of type A in this construction. Hence, a (de)constructor corresponding to this removal may be introduced.

For the type Continuum similar investigations concerning complete elementary relations are needed. Analogously as Procedure N for natural numbers, the *Procedure C* should be introduced as the basis for construction of the parameterized primitive relational types that determine the similarity relation and its completion. Although interesting, the similarity introduced in Section 6.2 is too weak to capture all homotopic properties of the objects of type Continuum.

10 Relation verifications and the constructor if_then_else

Verification whether a quantified relation (as a type) is inhabited or not, is not easy because the quantification scope may be potentially infinite and even not inductive. However, for particular cases the scope of quantification is restricted and, in fact, finite.

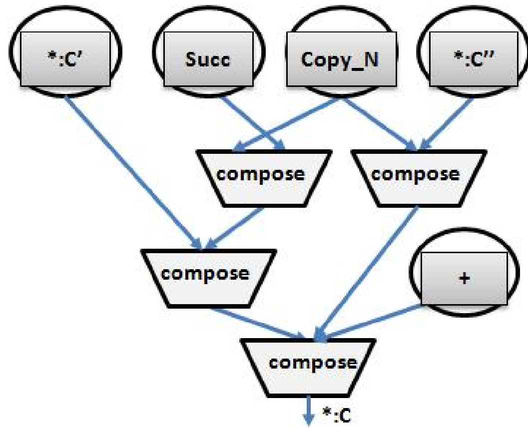

Figure 19: Operations Q^{+}

In programming there is a kind of quantifications over finite domains, that informally have a form of conditions, like *(for all* $k < n$ *there exists* $i < f(n)$ *such that* $R(k;i)$*)*. Although *for all* and *exists* above may resemble the constructors Π and Σ, they correspond to different constructions where Π and Σ are not used.

Let the name *condition* denote a type that consists of disjunctions and conjunctions of primitive relational types and their negations. The disjunctive normal form (disjunction of conjunctions) is very convenient for verification, i.e. once one component of the disjunction is verified as true, then the condition is true; if one component of a conjunction is false then the conjunction is false. A generic verification method can be constructed on the basis of primitive relational types.

To construct conditions, disjunction $+$, conjunction $\times$, and negation

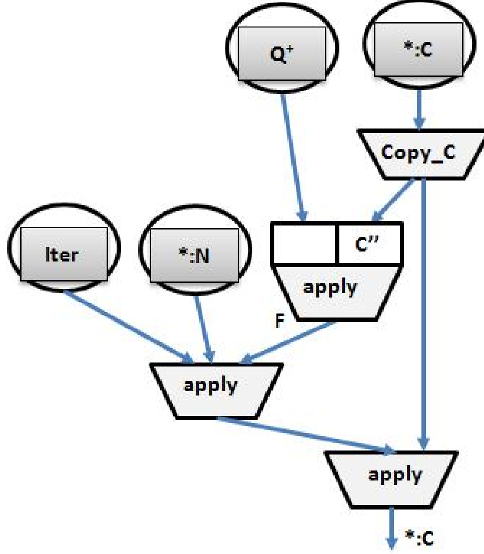

Figure 20: Operations $\bar{L}^+$

$\neg$ are used as operations. Consider an informal condition *exists* $i \leq n$*:* $R(i)$*)*, where $R : N \to Types$. Let C denote $N \to Types$.

Operation Q^+ of type $(C'; C'') \to C$ is constructed in Fig. 19. For R_1, R_2 and $i : N$, $\quad Q^+(R_1; R_2)(i)$ is the same as $R_1(i+1) + R_2(i)$.

We are going to construct operation L^+ of type $(C; N) \to (N' \to Types)$ such that for any $R : C$, $k : N$, and $n : N$, $\quad L^+(R; k)(n)$ corresponds to $(R(k)+R(k+1)+R(k+2)+...+R(k+n))$, i.e. informally *(exists* i *such that* $k \leq i \leq (k+n)$ *and* $R(i)$*)*.

In Fig. 20 operation $\bar{L}^+$ of type $(C; N) \to (N' \to Types))$ is constructed such that $\bar{L}^+(R; n)(k)$ is the same as $L^+(R; k)(n)$. The construction of $\bar{L}^+$ in Fig. 20 is explained below.

- Relation $R : C$ is copied, and $Copy^1(R)$ is applied to Q^+ to the input C'', and the result is denoted by F; it is of type $C \to C$. Applied to $Iter_C$, i.e. $Iter_C(*; F)$ is of type $N \to (C' \to C)$.

- By using currying and uncurrying (not shown in Fig. 20) we get

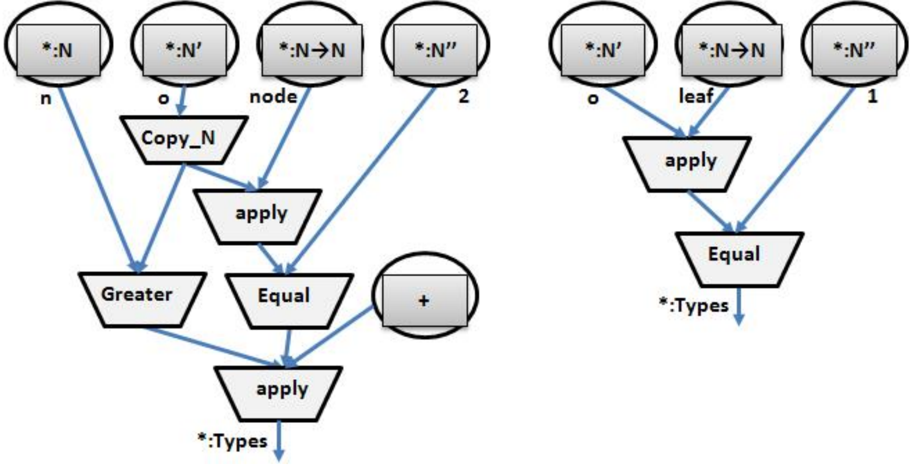

Figure 21: Relations R_{11} and R_{12}

equivalent operation of type $C' \to (N' \to C)$ that can be applied to $Copy^2(R)$.

- As the result, in the Fig. 20, the operation $\bar{L}^+$ of type $(C; N') \to C')$ is constructed. Since C' denotes $N \to Types$, $\bar{L}^+ : (C; N') \to (N \to Types)$. Using currying and uncurrying (swapping N and N'), we get finally the required operation L^+.

If in the construction of L^+ operation $+$ is changed to $\times$ (actually in Q^+), then the resulting operation is denoted by $L^\times$. Then $L^\times(R; 1)(n)$ corresponds to $(R(1) \times R(2) \times ... \times R(n))$, i.e. informally *(for all $i \leq n$: $R(i)$)*.

The condition corresponding to *(exists $i \leq n$ such that for all $j \leq k$, $R(i; j)$)* is constructed as follows.

Now R is of type $(N; N') \to Types$ where N corresponds to i, and N' to j. Apply currying to get the operation R^c of type $N \to (N' \to Types)$.

Operations $L^\times(*; 1)$ and $L^+(*; 1)$ are of type $(N' \to Types) \to (N \to Types)$, and are used in the very similar way as the constructors Π and

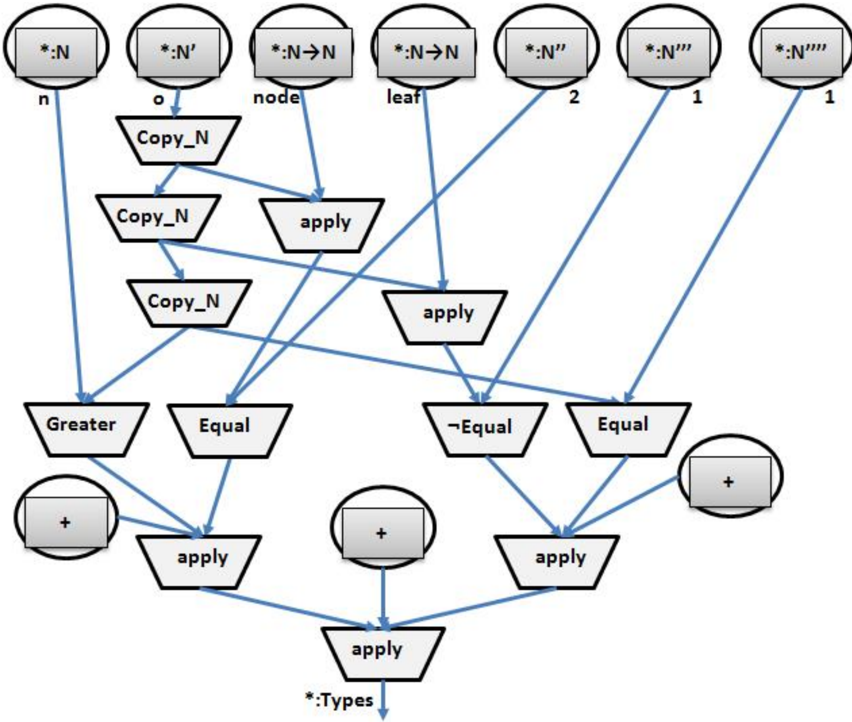

Figure 22: Relation R_{21}

Σ.

Compose R^c with $L^\times(*;1)$, i.e. $compose(R^c; L^\times(*,1))$ is of type $N \to (N \to Types)$. For i and k, $compose(R^c; L^\times)(i)(k)$ corresponds to $(R(i;1) \times R(i;2) \times ... \times R(i;k))$.

Compose $compose(R^c; L^\times(*;1))$ with $L^+(*;1)$, i.e. $compose(compose(R^c; L^\times(*;1)); L^+(*;1))$ is denoted by P; it is of type $N \to (N \to Types)$.

For n and k, $P(n)(k)$ corresponds to $((R(1;1) \times R(1;2) \times ... \times R(1;k)) + (R(2;1) \times R(2;2) \times ... \times R(2;k)) + ... + (R(n;1) \times R(n;2) \times ... \times R(n;k)))$.

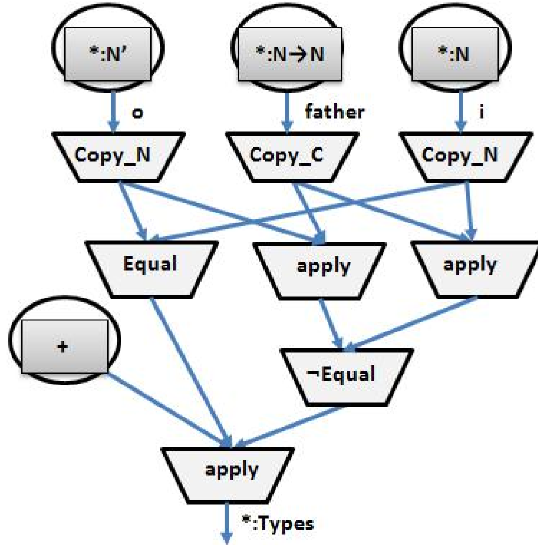

Figure 23: Relation R_{22}

10.1 Example

The following example serves to introduce a new operation constructor and a new type constructor.

Let $const_N$ denote the operation of type $N \to (N \to N)$ such that for any $c : N$ and any $k : N$, $const_N(c) : N \to N$, and $const_N(c)(k)$ is c.

The following operations: `node`, `father` and `leaf` together are interpreted as a data structure called tree. The parameter $n : N$ denotes the current scope of the data structure.

- `node` : $N \to N$. For any $i : N$, `node`(i) is either 1 (denotes an already constructed `node`), or 2 (denotes deleted `node`), or 3 (and greater) denotes unspecified node outside the current scope of the construction. Initially, `node` is $Change_N(1; 1; const_N(3))$, i.e. `node`$(1)$ is set as 1 (it is the root), and for i grater than 1, `node`(i) is set as 3.

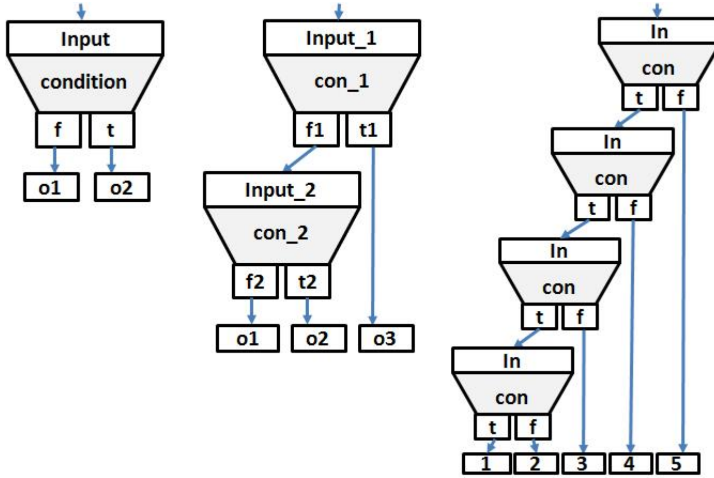

Figure 24: Operation `if_then_else`, composition, and iteration $\texttt{while}_B(4; con; t)$

- `father` : $N \to N$. `father`(i) is interpreted as the node that is the father of node i. Initially, it is the constant operation $const_N(1)$. The `node` and the input parameter $n : N$ determine which inputs of `father` have intended meaning, i.e. for node k greater than n or if k is a removed node, `father`(k) is ignored.

- `leaf` : $N \to N$. For any $i : N$, `leaf`(i) is either 1 (it is a leaf if `node`(1) is also 1), or 2 (is not a leaf), or 3 (and grater) as not constructed or deleted. Initially, `leaf` is $Change_N(1; 1; const_N(3))$.

The parameter $n : N$ denotes the number of the last constructed `node`; initially it is set as 1. The next natural number $Succ(n)$ is for the next `node` to be constructed in the tree.

In the construction of the operations, the parameter $n : N$ determines the current scope of the operations. For a parameter grater than n (outside of the current scope), the objects are still not constructed, so that the reference to them does not have the intended meaning.

Let A denote $(N \to N)$, and B denote $(N; N'; A; A'; A'')$ Two operation are constructed to modify a tree; *add* and *del* both of type

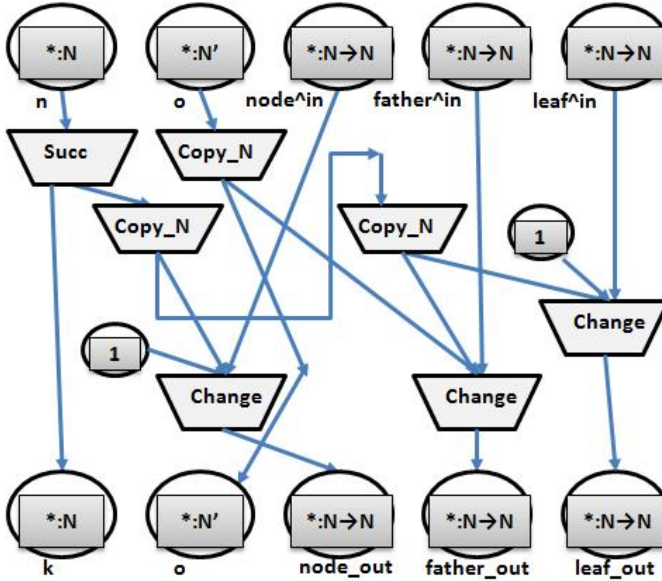

Figure 25: Operation f_{11} corresponding to *add*

$B \to B$. For input $(n; o; \texttt{node}^{in}; \texttt{father}^{in}; \texttt{leaf}^{in})$ they return output $(k; o; \texttt{node}_{out}; \texttt{father}_{out}; \texttt{leaf}_{out})$.

Operation *add* adds a node to the tree. The new node is given number $(n + 1)$ and its father is an already existing node o.

Operation *del* removes node o if it is a leaf.

The following pseudo-codes describe the operations.

Operation *add*:

1. **if** $(Greater_N(o; n)) + Equal_N(\texttt{node}^{in}(o); 2))$ is true, i.e. $o : N$ is either outside of the current scope or it is a deleted node
   then do nothing;
   else

   (a) Construct a new node $Succ(n)$ to be a child of the node o. That is, $\texttt{node}(Succ(n))$ becomes 1, i.e.
   $Change_N(Succ(n); 1; \texttt{node}^{in})$

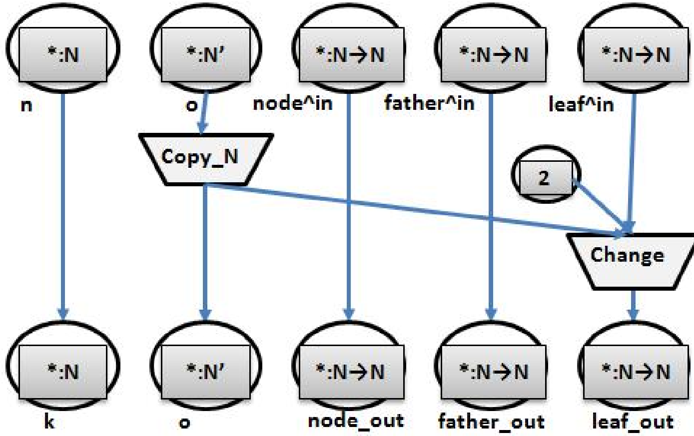

Figure 26: Operation t_{12} corresponding to *add*

(b) $\texttt{father}^{in}(Succ(n))$ becomes o, i.e. $Change_N(Succ(n); o; \texttt{father}^{in})$

(c) $\texttt{leaf}^{in}(Succ(n))$ becomes 1, i.e. $Change_N(Succ(n); 1; \texttt{leaf}^{in})$ denoted by $\texttt{leaf}^{in'}$

2. **if** $Equals_N(\texttt{leaf}^{in'}(o); 1)$, i.e. o was a leaf in the tree
   then

   (a) $\texttt{leaf}^{in'}(o)$ becomes 2, i.e. $Change_N(o; 2; \texttt{leaf}^{in'})$

   else do nothing.

Note that the phrase *'do nothing'* corresponds to id_B.

The constructor **if-then-else** is a new primitive. It needs a condition, and two operations. The input of the condition and the inputs of the two operations are the same.

The first condition (denoted by is S_{11}) of *add* corresponds to $(Greater_N(o; n)) + Equal_N(\texttt{node}^{in}(o); 2))$. To construct it, the relation R_{11} is needed that is in Fig. 21. It is of type $(N; N'; (N \to N); N'') \to Types$ where N corresponds to n, N' to o, $(N \to N)$ to $\texttt{node}^{in}$, and N'' to 2. $R_{11}(*; *, *; 2)$ is the required S_{11}.

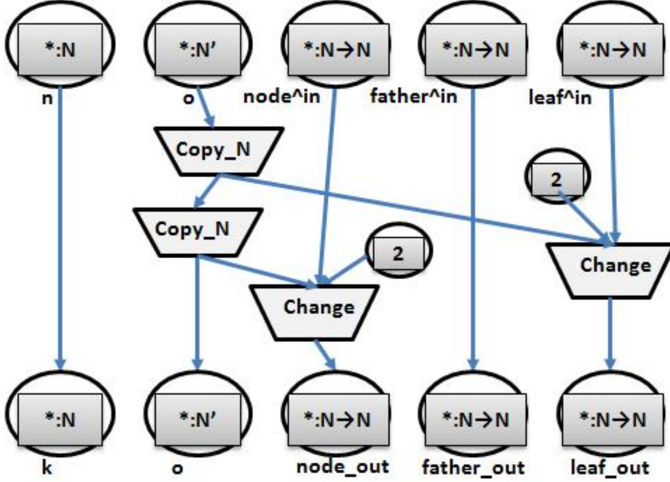

Figure 27: Operation f_{21} corresponding to *del*

The second condition (denoted by S_{12}) of *add*, corresponds to $Equals_N(\texttt{leaf}^{in}(o);1)$. In Fig. 21, relation R_{12} is constructed. It is of type $(N';(N \to N);N'') \to Types$, where N' corresponds to o, $(N \to N)$ to $\texttt{leaf}^{in}$, and N'' to 1. $R_{12}(*;*;1)$ is the required S_{12}.

Operation *del*:

1. **if** the condition $(Greater_N(o;n)+Equal_N(\texttt{node}^{in}(o);2)+\neg Equal_N(\texttt{leaf}^{in}(o);1)+Equal_N(o;1))$, is true, i.e. o is either outside of the scope, or is a removed node, or is not a leaf, or is the root of the tree,
   then do nothing
   else
   (a) $\texttt{node}(o)$ becomes 2, i.e. $Change_N(o;2;\texttt{node}^{in})$
   (b) $\texttt{leaf}(o)$ becomes 2, i.e. $Change_N(o;2;\texttt{leaf}^{in})$ is denoted by $\texttt{leaf}^{in'}$
2. **if** node o is the only child of its father, i.e. for all $i : N$ such that $Lesser_N(i;n)$,
   $(Equal_N(o;i) + \neg Equal_N(\texttt{father}^{in}(i);\texttt{father}^{in}(o)))$,

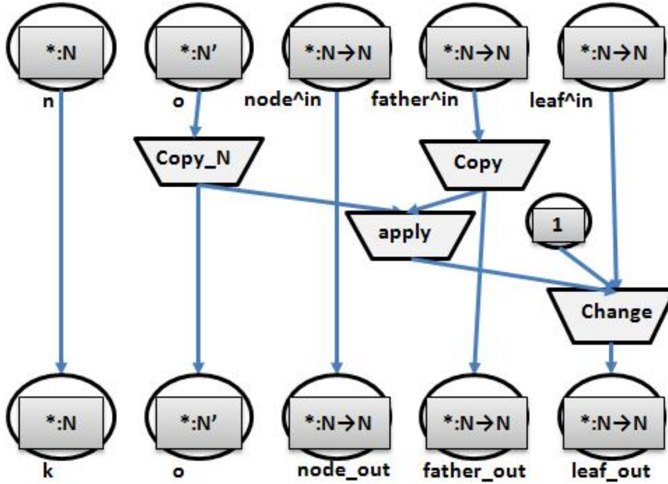

Figure 28: Operation t_{22} corresponding to *del*

i.e. either the nodes o and i are the same, or they have different fathers
then

(a) $\texttt{leaf}^{in'}(\texttt{father}^{in}(o))$ becomes 1, i.e.
$Change_N(\texttt{father}^{in}(o); 1; \texttt{leaf}^{in'})$

else do nothing

The first conditions in *del* (denoted by S_{21}) corresponds to $(Greater_N(o; n) + Equal_N(\texttt{node}^{in}(o); 2) + \neg Equal_N(\texttt{leaf}^{in}(o); 1) + Equal_N(o; 1))$. The auxiliary relation R_{21} is constructed in Fig. 22. It is of type $(N; N'; (N \to N); (N \to N); N''; N'''; N'''') \to Types$ where N corresponds to n, N' to o, the first $(N \to N)$ to $\texttt{node}^{in}$, the second $(N \to N)$ to $\texttt{leaf}^{in}$, N'' to 2, N''' to the first 1, and N'''' to the second 1. $R_{21}(*; *; *; *; 2; 1; 1)$ is the required S_{21}.

In order to construct the second condition of *del* (denoted by S_{22}), the operation R_{22} is constructed in Fig. 23. It is of type $((N \to N); N'; N) \to Types$ where $(N \to N)$ corresponds to $\texttt{father}^{in}$, N' to o, and N to i.

$R_{22}(\texttt{father}^{in}; o; i))$ corresponds to $(Equal_N(o; i) + \neg Equal_N(\texttt{father}^{in}(i); \texttt{father}^{in}(o)))$.
By currying we get operation R^c_{22} of type $(N \to N) \to (N' \to (N \to Types))$.
Operations $L^{\times}(*; 1)$ is of type $(N \to Types) \to (N'' \to Types)$.
Compose R^c_{22} with $L^{\times}(*; 1)$, i.e. $compose(R^c_{22}; L^{\times}(*; 1))$, is of type $(N \to N) \to (N' \to (N'' \to Types))$. By uncurrying we get the required S_{22}.
That is, for o, and n, $S_{22}(\texttt{father}^{in}; o; n)$ is the same as
$(R_{22}(\texttt{father}^{in}; o; 1) \times R_{22}(\texttt{father}^{in}; o; 2) \times ... \times R_{22}(\texttt{father}^{in}; o; n))$.

10.1.1 Operation `if_then_else` and `while` loop

It is clear that the `if_then_else` can not be constructed using the primitives introduced by now. It needs two operations $t : B \to C$ and $f : B \to D$ having the same input, and a condition $R : B \to Types$ also with the same input. Then, depending on $R(b)$, it returns either $t(b)$ (if the condition is true) or $f(b)$ (else). Let the constructor be denoted by $\texttt{if_then_else}_{(B,C,D)}$; see Fig. 24, where also a composition of the two constructors corresponding to operations *add* and *del* is shown. The constructor also needs a generic operation to evaluate conditions in their disjunctive normal form. This generic verification operation is supposed to be implicit in the constructor.

Note that B may denote a multiple input, i.e. $(B_1; ...; B_k)$, analogously for C and D.

The resulting operation has the type B as its input whereas C and D are its mutually exclusive outputs. The phrase *mutually exclusive types* gives rise to introduce a new type constructor denoted by $||$, so that $C||D$ denotes the output of the operation in question. It has also operational version, that is, $|| : (Types; Types) \to Types$.
$\texttt{if_then_else}_{(B,C,D)}$ is of type $((B \to Types); (B \to C); (B \to D)) \to (B \to (C||D))$. That is, for any $b : B$, $\texttt{if_then_else}_{(B,C,D)}(R; t; f)(b)$ is either $t(b)$ if $R(b')$, or $f(b'')$ otherwise. This means that the input object b is copied twice to get the same three objects b, b', b''.

If f (or t) is id_B, then it means *do nothing*, i.e. return the input as the output.

Note that $\texttt{if_then_else}_{(B,B,B)}$ cannot be reduced to operation of type $B \to B$.

The primitive operation $get_{A,B}$ (see Section 3.1) is, in fact, of type $(A + B) \to (A||B)$.

Operation corresponding to `while` loop in programing can be constructed using $\texttt{if_then_else}_{(B,B,B)}$ and a modified version of the iterator constructor. Informally, operation t is iterated if the condition is true. Let this conditional iteration be denoted by $\texttt{while}_B$; it is of type $(N; (B \to Types); (B \to B)) \to (B \to (B||B))$. For any $n : N$, $con : B \to Types$, and $t : B \to B$, $\texttt{while}_B(n; con; t)$ is the n times composition of the operation $\texttt{if_then_else}_{(B,B,B)}(con; t; id_B)$. Construction of $\texttt{while}_B(4; con; t)$ is shown in Fig. 24.

Actually the above construction of the `while` loop is rather inefficient. Only one or two of the mutually exclusive outputs are active. Active output means that there is an object in the output. Since the next composition depends on the current evaluation of the condition, it should be done only if the condition is true. It it is false then the iteration should be completed.

The operation *while* will be used in Section 11.

10.1.2 Constructions of *add* and *del*

Recall that B denotes $(N; N'; A; A'; A'')$ where A denotes $(N \to N)$. The constructions of the operations *add* and *del* use $\textbf{if_then_else}_{(B,B,B)}$.

Since the conditions for *add* and *del* are already constructed, only the corresponding operations t and f are to be constructed. By introducing dumb input types the conditions become of type $B \to Types$.

For *add*:

- The first `if_then_else`
  - Condition S_{11} is $R_{11}(*; *; *; 2)$; see Fig. 21.
  - Operation t is id_B.
  - Operation f is denoted by f_{11} and is constructed in Fig. 25.
- The second `if_then_else`

  - the Condition S_{12} is $R_{12}(*; *; 1)$; see Fig. 21.
  - Operation t is denoted by t_{12} and is constructed in Fig. 26.
  - Operation f is id_B.

For *del*:

- The first `if_then_else`
  - the Condition S_{21} is constructed using relation $R_{21}(*; *; *; *; 2; 1; 1)$; see Fig. 22.
  - Operation t is id_B.
  - Operation f is denoted by f_{21} and is constructed in Fig. 27.
- The second `if_then_else`
  - the Condition S_{22} is constructed using relation R_{22} (see Fig. 23) by currying, composition with $L^{\times}$ and uncurrying.
  - Operation t is denoted by t_{22} and is constructed in Fig. 28.
  - Operation f is id_B.

11 Formal theories and the Universe: Gödel's First Incompleteness Theorem

Grounding of a formal theory in the Universe is presented on the basis of the Gödel theorem. The proof presented below is essentially the same as the one by Barkley Rosser [32]. It is clear and obvious because the *logic of ordinary discourse* for reasoning about a formal theory (see the first page of [32]), can be constructed rigorously as a (relatively small) part of the Universe. The problem of consistency of a formal theory is discussed at the end of the Section.

Primitive types that correspond to a formal language and formal theories are introduced below.

Formal language L is an extension of the first order logic language with additional symbols for functions, relations and constants specific to a formal theory.

For the simplicity of the presentation, let terms of L be objects of primitive type denoted by $Term_L$, and well formed formulas of L be objects of primitive type denoted by $Form_L$. Actually, rather L should be considered as the primitive type along with primitive operations for constructing objects of $Term_L$ and objects of $Form_L$.

$Form_L$ is an inductive type, i.e. its objects can be enumerated by operation $\bar{G}_L : N \to Form_L$. For the reason to be clear later on, the enumeration starts from the number 2. Number 1 is reserved for unspecified formula. Natural numbers will be used to denote formulas.

There is also $G_L : Form_L \to N$, a Gödel numbering, that is, each formula is given the unique natural number called Gödel number.

From the constructivist point of view presented in this work, any formal theory T (in the formal language L) is identified with operation P_T that is a construction (enumeration) of all formulas that are provable in this theory, i.e. they have proofs in T. P_T is of type $N \to Form_L$. Actually the construction of P_T is based on the axioms of T and inference rules. The final formula of a proof is just the formula that has been proved.

Hence, given G_L and $\bar{G}_L$, any formal theory T of language L is identified with P_T. The question about semantics of a formal theory is about an interpretation of the language in some well established domain. An interpretation, as a grounding in parameterized finite structures of the Universe, will be presented after the proof of Gödel Theorem. The composition of P_T with a Gödel numbering G_L, i.e. $compose(P_T; G_L)$ is the operation that enumerates all Gödel numbers (formulas) that are provable in T. Denote this composition by S; it is of type $N \to N$. The composition of $\bar{G}_L$ with a Gödel numbering G_L, i.e. $compose(\bar{G}_L; G_L)$ enumerates all Gödel numbers. Denote this composition by Q; it is of type $N \to N$. Now, the natural numbers denote formulas, whereas $Q(n)$ is the Gödel number of formula n.

If the type $Equal_N(Q(n); S(k))$ is inhabited, then formula n is provable, i.e. its Gödel number $Q(n)$ is equal to $S(k)$ a Gödel number of a provable formula.

For all $k : N$, $\neg Equal_N(Q(n); S(k))$ means that the formula n having Gödel number $Q(n)$ is not provable.

To construct the corresponding type, compose Q and S with $\neg Equal_N$, i.e. $compose(Q; S; \neg Equal_N)$; it is of type $(N'; N) \rightarrow Types^1$.

Apply currying, i.e. $currying(compose(S; Q; \neg Equal_N))$; denote it by H, it is operation of type $N' \rightarrow (N \rightarrow Types^1)$ with the input corresponding to Q. Operational version of constructor Π is needed; i.e. $\Pi^2 : (N \rightarrow Types^1) \rightarrow Types^2$.

Compose H with Π^2, i.e. $compose(H; \Pi^2)$. It is relation of type $N' \rightarrow Types^2$. Denote it by F. $F(n)$ means that formula n is not provable in T.

Let $V : (N; N'; N'') \rightarrow N$ denote the relation such that for any $k : N$, $i : N'$ and $j : N''$, $V(k; i; j)$ is inhabited if and only if k is the number of the formula resulting from replacing in formula i all occurrences of its free variables with the term $[j]$ corresponding to the number j. Although details of the construction of V are not presented here, the following problem is important. How to construct the operation (denoted by g) such that for a formula (say ψ) as input it returns its number k such that $\bar{G}_L(k)$ is *the same* as ψ. Note that *the same* corresponds to the equality relation on $Form_L$. This very operation must be used in construction of V.

Actually the operation g must have additional parameter $n : N$ as the upper bound of verification if $\bar{G}_L(k)$ (where k is lesser than n) is *the same* as the input ψ. Hence, g must be of type $(Form_L; N) \rightarrow N$. Its construction requires the equality relation on the type $Form_L$, so that it must be constructed first. Let this equality be denoted by Eq_L.
For any formulas f_1, f_2, $Eq_L(f_1, f_2)$ is the same as
$Equal_N(G_L(f_1); G_L(f_1))$. Hence, Eq_L can be easily constructed.
The construction of g uses `while` loop with the condition $Lesser_N(k; n) \times Eq_L(f, \bar{G}_L(k))$. Here $n : N$ refers to number of maximum iteration times. For all $n : N$ and $k : N$ such that k is equal or less than n, if $Eq_L(f, \bar{G}_L(k))$, then $g(f; n)$ is set as k, otherwise $g(f; n)$ is set as 1 (unspecified). Although the details of the construction of V are important, it will be done elsewhere due to the limit of space.

Suppose that Q, and S can be represented as terms, and V as a formula in L, i.e. $[S]$, $[Q]$ are objects of the type $Term_L$, and $[V]$ is of type $Form_L$. This is the crucial point of the Gödel theorem. In Peano

Arithmetic these terms and this formula can be defined.

(*) There exists $n : N$ such that $F(n)$ and $V(n; m; m)$ corresponds to the following formula:
(**) $\exists_{[n]}\forall_{[k]}([Q]([n]) \neq [S]([k]) \wedge [V]([n],[m],[m])$; where $[m]$ is a free variable and symbols in square brackets denote corresponding terms in L. Denote this formula by $\phi([m])$.

Hence, $\phi([m])$ is an object of type $Form_L$.

The construction of the relation corresponding to the formula (*) is shown in Fig. 29; it is operation $\bar{K} : (N'; N) \to Types^1$ such that for any $n : N'$ and $m : N$, $\bar{K}(n; m)$ is the same as $F(n) \times V(n; m; m)$.

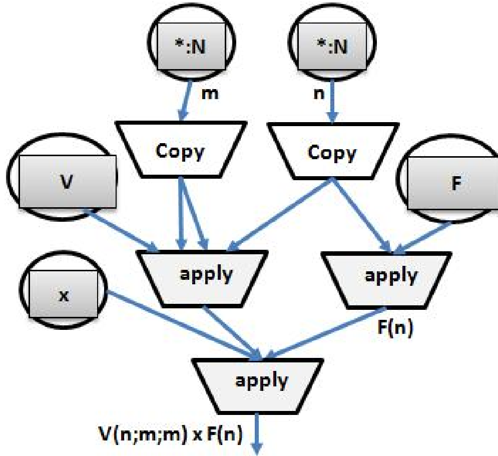

Figure 29: Construction of operation $\bar{K}$

- Apply currying to $\bar{K}$, and denote the result by K^c; it is of type $N \to (N' \to Types^1)$, and has input corresponding to $m : N$.
- The operation $\Sigma^2 : (N' \to Types^1) \to Types^2$ is needed now.
- Let *compose*$(K^c; \Sigma^2)$ be denoted by K; it is of type $N \to Types^2$, and corresponds to the formula $\phi([m])$, which is an object of type

$Form_L$. For $l : N$, $K(l)$ is the grounding (meaning) of the formula $\phi([l])$. So that if $K(l)$ is inhabited ($\phi([l])$ is true), then if $V(n; l; l)$ is inhabited, then the formula having number n is not provable in T.

Let us summarize.
$\phi([m])$ denotes $\exists_{[n]}\forall_{[k]}([Q]([n]) \neq [S]([k]) \wedge [V]([n], [m], [m])$.
Let l be the number of formula $\phi([m])$, i.e. $\bar{G}_L(l)$ is $\phi([m])$.
Consider the formula $\phi([l])$, i.e. $\exists_{[n]}\forall_{[k]}([Q]([n]) \neq [S]([k]) \wedge [V]([n], [l], [l])$.
It is the formula resulting from replacing in formula l all occurrences of its free variables with $[l]$.
Let n be the number of the formula $\phi([l])$. Then, the type $V(n; l; l)$ is inhabited. Although it seems to be obvious, the construction of an object of type $V(n; l; l)$ requires some effort.

If $K(l)$ is inhabited (i.e. $\phi([l])$ is true), then the formula having number n (i.e. $\phi([l])$) is not provable.

If the type $K(l)$ is not inhabited ($\phi([l])$ is false), then the formula number n (i.e. $\phi([l])$) is provable. That is, the theory T is inconsistent.

The conclusion: If theory T is consistent, then the formula $\phi([l])$ is *true* and not provable. This is the Gödel's First Incompleteness Theorem.

Remark 1. The word *true* in the statement of the Gödel Theorem means that the type $K(l)$ is inhabited. Hence, to be precise the statement of the theorem is the following disjunction: $\neg(T\ is\ consistent) + K(l)$. Note that here the consistency means that no *false* formula is provable in T. This presupposes the interpretation of the language L in the Universe that implicitly was already done. The problem is how to provide the exact meaning for this very consistency as the type in the Universe. Since the paper is already too long, this will be done elsewhere.

Remark 2. For a formal language L, terms correspond to objects, formulas with free variables correspond to relations, propositions correspond to types. This very correspondence between a formal language

L and objects in the Universe is a grounding (concrete semantics) for the language L. Axioms of a formal theory T correspond to inhabited types, and to objects of these types; some of the objects may be primitive. Inference rules correspond to primitive constructors and primitive operations. As a whole, this correspondence is a construction in the Universe. Hence, from the model theoretic point of view, this very construction may be considered as an interpretation of the formal language L, and corresponds to Alfred Tarski semantics [37]. Natural deduction (introduced by Stanisław Jaśkowski [20] [21], and Gerhard Genzen [9]) correspond to general methods of object constructions in the Universe, i.e. to the primitive constructors and primitive operations.

12 Final conclusion

All objects constructed in this paper are parameterized finite structures. Each object and its type is determined and identified by its construction. Operation *Copy* applied to an object a returns a copy of the object, i.e. repetition of the construction of a must be done. However, this copy is different than its original object. Once an object is used in a construction of another object, it cannot be used again. It is clear because objects are interpreted by means of signals, links, and sockets. Since types and their constructors are interpreted as objects on the consecutive levels of the Universe, they also can be used only once. Operation *Copy* must be explicitly applied to produce as many copies as needed.

Universe is always open to qualitatively new constructions, and never completed. As its instance (i.e. all that has already been constructed by now), it is always a finite structure, because all its construction parameters are bounded, and there are finite many of such parameters.

The constructions in the Universe are monotonic and continuous. As far as computable functionals are considered (see the classic definitions by Kleene [24][25], Kreisel [26], Grzegorczyk [13] and [14]) they are operations in the Universe. Also correspondence to the work of Platek & Scott PCF^{++} [35, 34]) and to Scott Domain [36] is straightforward. Scott Domain may be viewed as a mathematical model (abstract description) of the Universe as a complete entity.

Correspondences to System F, ML TT, and HoTT are clear and were already discussed in the previous sections. Since they are formal theories, it seems that they may have grounding in the Universe.

The Curry-Howard (also the Brouwer-Heyting-Kolmogorov) interpretation should be revised. In the classic view it is interpretation of propositions as types, and proofs as objects of these types. If types and proofs are still syntactic terms and term rewriting rules, then it is an interpretation of one formal theory in another formal (type) theory. However, if types and objects are parameterized finite structures constructed as objects of the Universe, then this correspondence becomes interpretation (semantics) of the formal theory.

Comparison to Calculus of Inductive Constructions CoIC is of some importance, because the Universe resembles CoIC. However, the main difference is that CoIC is a formal type theory based on Lambda calculus. It satisfies the properties of confluence and strong normalization, so that constructions are denoted by terms whereas computation is identified with term reduction. Implementation of CoIC, as Coq proof assistant, provides an operational semantics. There is an infinite well-founded typing hierarchy of sorts whose base sorts are Prop and Set, i.e. the hierarchy of universes Type(i) for any natural number i. Prop and Set are objects of type Type(1). Type(i) is an object of Type(i+1). The sort Prop intends to be the type of logical propositions. The sort Set intends to be the type of small sets. This includes data types such as booleans and natural numbers, but also products, subsets, and function types over these data types. Set corresponds to $Types^0$, Type(i+1) correspond to higher levels of the Universe. Prop is an impredicative formal theory so that it is incompatible with the relations in the Universe.

It seems that the Universe is natural, simple and obvious. It is an attempt to understand what is grounding (semantics) of the (formal) theories. Although this concerns mainly Mathematics and Informatics, it is of some importance also for programming. It seems a bit strange, that development of high level programming languages (for example Haskell, Clojure and Scala) is far ahead of the theoretical investigations concerning computations on higher types.

The Universe was presented above in an informal way, that is, the

constructions were described in such a way that the idea and the essence could to be easily grasped by the Reader. Since the constructions as objects are claimed to be parametrized finite structures, these very structures must be realized. The descriptions and particularly the figures only suggest a way of implementing these objects as data structures in programming. An interface is needed to assist a user to construct new objects as well as to use them in computations and reasoning.

The work poses more problems than it solves. It seems that there are still a lot of new constructors and new primitive types to discover.

Acknowledgements. The work was supported by the grants:
RobREx - Autonomia dla robotów ratowniczo-eksploracyjnych. Grant NCBR Nr PBS1/A3/8/2012 w ramach Programu Badań Stosowanych w Obszarze Technologie informacyjne, elektronika, automatyka i robotyka, w Ścieżce A.
oraz *IT SOA - Nowe technologie informacyjne dla elektronicznej gospodarki i Społeczeństwa informacyjnego oparte na paradygmacie SOA*; Program Operacyjny Innowacyjna Gospodarka: Działanie 1.3.1.

References

[1] J. Adamek, S. Milius, and J. Velebil. Semantics of higher-order recursion schemes. *Logical Methods in Computer Science*, 7(1:15):1–43, 2011.

[2] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. Preprint, 2013.

[3] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Math. Struct. in Comp. Science*, 15:671–708, 2005.

[4] T. Coquand. Coq proof assistant. chapter 4 calculus of inductive constructions. www http://coq.inria.fr/doc/Reference-Manual006.html, 2014. Site on www.

[5] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[6] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988.

[7] Haskell B. Curry. Combinatory recursive objects of all finite types. *Bull. Amer. Math. Soc.*, 70(6):814–817, 1964. http://projecteuclid.org/euclid.bams/1183526340.

[8] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. Amsterdam: North Holland, 1958.

[9] Gerhard Gentzen. Untersuchungen uber das logische Schliessen I and II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934.

[10] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7. Cambridge University Press (Cambridge Tracts in Theoretical Computer Science), 1990.

[11] J.Y. Girard. Une extension de l'interpretation de godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et dans la theorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium 1971.*

[12] K. Gödel. Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. *Dialectica*, 10:280 – 287, 1958.

[13] A. Grzegorczyk. Computable functionals. *Fundamenta Mathematicae*, 42:168–202, 1955.

[14] A. Grzegorczyk. On the definition of computable functionals. *Fundamenta Mathematicae*, 42:232–239, 1955.

[15] A. Grzegorczyk. On the definitions of computable real continuous functions. *Fundamenta Mathematicae*, 44:61–71, 1957.

[16] A. Grzegorczyk. Recursive objects in all finite types. *Fundamenta Mathematicae*, 54:73–93, 1964.

[17] R. Harper. What is the big deal with HoTT? http://existentialtype.wordpress.com/2013/06/22/whats-the-big-deal-with-hott/.

[18] D. Hilbert. *Gesammelte Abhandlungen*, volume 3. Berlin, 1935.

[19] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Proceedings. 14th Symposium on Logic in Computer Science*, pages 204–204. IEEE Computer Society, 1999.

[20] Stanisław Jaśkowski. Teoria dedukcji oparta na regułach założeniowych (theory of deduction based on suppositional rules). In *Księga pamiątkowa pierwszego polskiego zjazdu matematycznego (Proceedings of the First Polish Mathematical Congress), 1927*, page 36. Krakow: Polish Mathematical Society, 1929.

[21] Stanisław Jaśkowski. On the rules of suppositions in formal logic. *Studia Logica*, 1:5–32, 1934.

[22] Tomasz Kaczynski, Konstantin Mischaikow, and Marian Mrozek. *Computational Homology.* Springer-Verlag, 2004.

[23] S. C. Kleene. Countable functionals. *Constructivity in Mathematics: Proceedings of the colloquium held at Amsterdam*, pages 81–100, 1959.

[24] S. C. Kleene. Recursive functionals and quantifiers of finite types i. *Transactions of the American Mathematical Society*, 91:1–52, 1959.

[25] S. C. Kleene. Recursive functionals and quantifiers of finite types ii. *Transactions of the American Mathematical Society*, 108:106–142, 1963.

[26] G. Kreisel. Interpretation of analysis by means of functionals of finite type,. *Constructivity in Mathematics: Proceedings of the colloquium held at Amsterdam*, pages 101–128, 1959.

[27] D. Lacombe. Remarques sur les operateurs recursifs et sur les fonctions recursives d'une variable reelle. *Comptes Rendus de l'Academie des Sciences, Paris*, 241:1250–1252, 1955.

[28] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloqium 1973.*

[29] L. C. Paulson. Constructing recursion operators in intuitionistic type theory. *J. Symbolic Computation*, 2:325–355, 1986.

[30] Rózsa Péter. *Recursive functions.* Elsevier Science Publishing Co Inc,, 1967.

[31] J. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation 1974*, pages 408–425. Paris, France, 1974.

[32] Barkley Rosser. An Informal Exposition of Proofs of Gödel's Theorems and Church's Theorem. *Journal of Symbolic Logic*, (2):53–60, 06.

[33] Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Fundamental study. primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266:1–57, 2001.

[34] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1969.

[35] D. S. Scott. A theory of computable functions of higher type. *University of Oxford*, 1969.

[36] D. S. Scott. Outline of a mathematical theory of computation. *Technical Monograph PRG-2*, 1970.

[37] A. Tarski. *Logic, Semantics, Metamathematics, papers from 1923 to 1938.* Indianapolis: Hackett Publishing Company, 1983.

[38] A.S. Troelstra. History of constructivism in the 20th century. in set theory, arithmetic, and foundations of mathematics. theorems, philosophies. *Lecture Notes in Logic*, 36:7–9, 2011.

[39] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study,http://homotopytypetheory.org/book, 2013.

[40] Vladimir Voevodsky. The origins and motivations of univalent foundations. *IAS - The Institute Letter. Summer 2014, Institute for Advanced Study, Princeton, NJ, USA*, pages 8–9, 2014.